Cloud Native PG

Release 1.23.1

The Cloud NativePG Contributors.

May 08, 2024

Contents

1 CloudNativePG **1**

1.1 Supported Kubernetes distributions . . . 2
1.2 Container images . . . 2
1.3 Main features . . . 2
1.4 About this guide . . . 4
1.4.1 Introduction . . . 4
1.4.2 Before You Start . . . 6
1.4.3 Use cases . . . 8
1.4.4 Architecture . . . 11
1.4.5 Installation and upgrades . . . 17
1.4.6 Quickstart . . . 26
1.4.7 Image Catalog . . . 32
1.4.8 Bootstrap . . . 33
1.4.9 Importing Postgres databases . . . 44
1.4.10 Security . . . 49
1.4.11 Postgres instance manager . . . 55
1.4.12 Scheduling . . . 57
1.4.13 Resource management . . . 59
1.4.14 Failure Modes . . . 61
1.4.15 Rolling Updates . . . 64
1.4.16 Replication . . . 66
1.4.17 Backup . . . 72
1.4.18 Backup on object stores . . . 78
1.4.19 WAL archiving . . . 81
1.4.20 Backup on volume snapshots . . . 82
1.4.21 Recovery . . . 88
1.4.22 PostgreSQL Configuration . . . 98
1.4.23 Database Role Management . . . 109
1.4.24 Tablespaces . . . 114
1.4.25 Operator configuration . . . 120
1.4.26 Instance pod configuration . . . 124
1.4.27 Volume Claim Template for Temporary Storage . . . 125
1.4.28 Storage . . . 127
1.4.29 Labels and annotations . . . 135
1.4.30 Monitoring . . . 138
1.4.31 Logging . . . 154

1.4.32 Certificates . . . 158
1.4.33 Client TLS/SSL connections . . . 168
1.4.34 Connecting from an application . . . 171
1.4.35 Connection pooling . . . 173
1.4.36 Replica clusters . . . 187
1.4.37 Kubernetes Upgrade and Maintenance . . . 192
1.4.38 Exposing Postgres Services . . . 195
1.4.39 Kubectl Plugin . . . 198
1.4.40 Automated failover . . . 221
1.4.41 Troubleshooting . . . 222
1.4.42 Fencing . . . 235
1.4.43 Declarative hibernation . . . 237
1.4.44 PostGIS . . . 239
1.4.45 End-to-End Tests . . . 242
1.4.46 Container Image Requirements . . . 245
1.4.47 Operator capability levels . . . 246
1.4.48 Custom Pod Controller . . . 258
1.4.49 Examples . . . 260
1.4.50 Networking . . . 277
1.4.51 Benchmarking . . . 279
1.4.52 Commercial support . . . 282
1.4.53 Frequently Asked Questions (FAQ) . . . 282
1.4.54 API Reference . . . 288
1.4.55 Supported releases . . . 333
1.4.56 Release notes . . . 337
1.4.57 Appendixes . . . 392
1.4.58 Appendixes . . . 392

CHAPTER 1

CloudNativePG

CloudNativePG is an open source operator

designed to manage PostgreSQL workloads on any supported
Kubernetes cluster running in private, public, hybrid, or multi-cloud

environments. CloudNativePG adheres to DevOps principles and concepts such as declarative configuration and immutable infrastructure.

It defines a new Kubernetes resource called `Cluster` representing a PostgreSQL cluster made up of a single primary and an optional number of replicas that co-exist in a chosen Kubernetes namespace for High Availability and offloading of read-only queries.

Applications that reside in the same Kubernetes cluster can access the PostgreSQL database using a service which is solely managed by the operator, without having to worry about changes of the primary role following a failover or a switchover. Applications that reside outside the Kubernetes cluster, need to configure a Service or Ingress object to expose the Postgres via TCP. Web applications can take advantage of the native connection pooler based on PgBouncer.

CloudNativePG was originally built by EDB , then released open source under Apache License 2.0 and submitted for CNCF Sandbox in April 2022. The source code repository is in Github .

Note: Based on the *Operator Capability Levels model* , users can expect a **"Level V - Auto Pilot"** set of capabilities from the CloudNativePG Operator.

1.1 Supported Kubernetes distributions

Each minor release of CloudNativePG is designed to work with a range of Kubernetes versions, usually the ones supported by the CNCF at the time the minor version was first released.

Please refer to the *Supported releases* page for details.

1.2 Container images

The CloudNativePG community maintains container images for both the operator and the operand, that is PostgreSQL.

The CloudNativePG operator container images are distroless

and available on the cloudnative-pg .

The PostgreSQL operand container images are available for all the
PGDG supported versions of PostgreSQL , on multiple architectures, directly from the postgres-containers .

Additionally, the Community provides images for the *PostGIS extension* .

1.3 Main features

- Direct integration with Kubernetes API server for High Availability, without requiring an external tool
- Self-Healing capability, through:
- failover of the primary instance by promoting the most aligned replica
- automated recreation of a replica
- Planned switchover of the primary instance by promoting a selected replica
- Scale up/down capabilities
- Definition of an arbitrary number of instances (minimum 1 - one primary server)
- Definition of the *read-write* service, to connect your applications to the only primary server of the cluster
- Definition of the *read-only* service, to connect your applications to any of the instances for reading workloads
- Declarative management of PostgreSQL configuration, including certain popular Postgres extensions through the cluster `spec` : `pgaudit` , `auto_explain` , `pg_stat_statements` , and `pg_failover_slots`
- Declarative management of Postgres roles, users and groups
- Support for Local Persistent Volumes with PVC templates
- Reuse of Persistent Volumes storage in Pods
- Separate volumes for WAL files and tablespaces
- Declarative management of Postgres tablespaces, including temporary tablespaces
- Rolling updates for PostgreSQL minor versions
- In-place or rolling updates for operator upgrades
- TLS connections and client certificate authentication
- Support for custom TLS certificates (including integration with cert-manager)

- Continuous WAL archiving to an object store (AWS S3 and S3-compatible, Azure Blob Storage, and Google Cloud Storage)
- Backups on volume snapshots (where supported by the underlying storage classes)
- Backups on object stores (AWS S3 and S3-compatible, Azure Blob Storage, and Google Cloud Storage)
- Full recovery and Point-In-Time recovery from an existing backup on volume snapshots or object stores
- Offline import of existing PostgreSQL databases, including major upgrades of PostgreSQL
- Online import of existing PostgreSQL databases, including major upgrades of PostgreSQL, through PostgreSQL native logical replication (imperative, via the `cnpg` plugin)
- Fencing of an entire PostgreSQL cluster, or a subset of the instances in a declarative way
- Hibernation of a PostgreSQL cluster in a declarative way
- Support for Synchronous Replicas
- Support for HA physical replication slots at cluster level
- Synchronization of user defined physical replication slots
- Backup from a standby
- Backup retention policies (based on recovery window, only on object stores)
- Parallel WAL archiving and restore to allow the database to keep up with WAL generation on high write systems
- Support tagging backup files uploaded to an object store to enable optional retention management at the object store layer
- Replica clusters for PostgreSQL deployments across multiple Kubernetes clusters, enabling private, public, hybrid, and multi-cloud architectures
- Delayed Replica clusters
- Connection pooling with PgBouncer
- Support for node affinity via `nodeSelector`
- Native customizable exporter of user defined metrics for Prometheus through the `metrics` port (9187)
- Standard output logging of PostgreSQL error messages in JSON format
- Automatically set `readOnlyRootFilesystem` security context for pods
- `cnpg` plugin for `kubectl`
- Simple bind and search+bind LDAP client authentication
- Multi-arch format container images
- OLM installation

Note: CloudNativePG does not use *StatefulSet* s for managing data persistence. Rather, it manages persistent volume claims (PVCs) directly. If you are curious, read *Custom Pod Controller* to know more.

1.4 About this guide

Follow the instructions in the *Quickstart* to test CloudNativePG on a local Kubernetes cluster using Kind, or Minikube.

In case you are not familiar with some basic terminology on Kubernetes and PostgreSQL, please consult the *Before You Start* .

- Postgres, PostgreSQL and the Slonik Logo

are trademarks or registered trademarks of the PostgreSQL Community Association of Canada, and used with their permission.*

1.4.1 Introduction

Supported Kubernetes distributions

Each minor release of CloudNativePG is designed to work with a range of Kubernetes versions, usually the ones supported by the CNCF at the time the minor version was first released.

Please refer to the *Supported releases* page for details.

Container images

The CloudNativePG community maintains container images for both the operator and the operand, that is PostgreSQL.

The CloudNativePG operator container images are distroless

and available on the cloudnative-pg .

The PostgreSQL operand container images are available for all the
PGDG supported versions of PostgreSQL , on multiple architectures, directly from the postgres-containers .

Additionally, the Community provides images for the *PostGIS extension* .

Main features

- Direct integration with Kubernetes API server for High Availability, without requiring an external tool
- Self-Healing capability, through:
- failover of the primary instance by promoting the most aligned replica
- automated recreation of a replica
- Planned switchover of the primary instance by promoting a selected replica
- Scale up/down capabilities
- Definition of an arbitrary number of instances (minimum 1 - one primary server)
- Definition of the *read-write* service, to connect your applications to the only primary server of the cluster
- Definition of the *read-only* service, to connect your applications to any of the instances for reading workloads
- Declarative management of PostgreSQL configuration, including certain popular Postgres extensions through the cluster `spec` : `pgaudit` , `auto_explain` , `pg_stat_statements` , and `pg_failover_slots`
- Declarative management of Postgres roles, users and groups
- Support for Local Persistent Volumes with PVC templates

- Reuse of Persistent Volumes storage in Pods
- Separate volumes for WAL files and tablespaces
- Declarative management of Postgres tablespaces, including temporary tablespaces
- Rolling updates for PostgreSQL minor versions
- In-place or rolling updates for operator upgrades
- TLS connections and client certificate authentication
- Support for custom TLS certificates (including integration with cert-manager)
- Continuous WAL archiving to an object store (AWS S3 and S3-compatible, Azure Blob Storage, and Google Cloud Storage)
- Backups on volume snapshots (where supported by the underlying storage classes)
- Backups on object stores (AWS S3 and S3-compatible, Azure Blob Storage, and Google Cloud Storage)
- Full recovery and Point-In-Time recovery from an existing backup on volume snapshots or object stores
- Offline import of existing PostgreSQL databases, including major upgrades of PostgreSQL
- Online import of existing PostgreSQL databases, including major upgrades of PostgreSQL, through PostgreSQL native logical replication (imperative, via the `cnpg` plugin)
- Fencing of an entire PostgreSQL cluster, or a subset of the instances in a declarative way
- Hibernation of a PostgreSQL cluster in a declarative way
- Support for Synchronous Replicas
- Support for HA physical replication slots at cluster level
- Synchronization of user defined physical replication slots
- Backup from a standby
- Backup retention policies (based on recovery window, only on object stores)
- Parallel WAL archiving and restore to allow the database to keep up with WAL generation on high write systems
- Support tagging backup files uploaded to an object store to enable optional retention management at the object store layer
- Replica clusters for PostgreSQL deployments across multiple Kubernetes clusters, enabling private, public, hybrid, and multi-cloud architectures
- Delayed Replica clusters
- Connection pooling with PgBouncer
- Support for node affinity via `nodeSelector`
- Native customizable exporter of user defined metrics for Prometheus through the `metrics` port (9187)
- Standard output logging of PostgreSQL error messages in JSON format
- Automatically set `readOnlyRootFilesystem` security context for pods
- `cnpg` plugin for `kubectl`
- Simple bind and search+bind LDAP client authentication
- Multi-arch format container images
- OLM installation

Note: CloudNativePG does not use *StatefulSet* s for managing data persistence. Rather, it manages persistent volume claims (PVCs) directly. If you are curious, read *Custom Pod Controller* to know more.

About this guide

Follow the instructions in the *Quickstart* to test CloudNativePG on a local Kubernetes cluster using Kind, or Minikube.

In case you are not familiar with some basic terminology on Kubernetes and PostgreSQL, please consult the *Before You Start* .

- Postgres, PostgreSQL and the Slonik Logo

are trademarks or registered trademarks of the PostgreSQL Community Association of Canada, and used with their permission.*

1.4.2 Before You Start

Before we get started, it is essential to go over some terminology that is specific to Kubernetes and PostgreSQL.

Kubernetes terminology

Node
A *node* is a worker machine in Kubernetes, either virtual or physical, where all services necessary to run pods are managed by the control plane node(s).

Pod
A *pod* is the smallest computing unit that can be deployed in a Kubernetes cluster and is composed of one or more containers that share network and storage.

Service
A *service* is an abstraction that exposes as a network service an application that runs on a group of pods and standardizes important features such as service discovery across applications, load balancing, failover, and so on.

Secret
A *secret* is an object that is designed to store small amounts of sensitive data such as passwords, access keys, or tokens, and use them in pods.

Storage Class
A *storage class* allows an administrator to define the classes of storage in a cluster, including provisioner (such as AWS EBS), reclaim policies, mount options, volume expansion, and so on.

Persistent Volume
A *persistent volume* (PV) is a resource in a Kubernetes cluster that represents storage that has been either manually provisioned by an administrator or dynamically provisioned by a *storage class* controller. A PV is associated with a pod using a *persistent volume claim* and its lifecycle is independent of any pod that uses it. Normally, a PV is a network volume, especially in the public cloud. A **local persistent volume* (LPV)* is a persistent volume that exists only on the particular node where the pod that uses it is running.

Persistent Volume Claim
A *persistent volume claim* (PVC) represents a request for storage, which might include size, access mode, or a particular storage class. Similar to how a pod consumes node resources, a PVC consumes the resources of a PV.

Namespace
A *namespace* is a logical and isolated subset of a Kubernetes cluster and can be seen as a *virtual cluster* within the wider physical cluster. Namespaces allow administrators to create separated environments based on projects, departments, teams, and so on.

RBAC
Role Based Access Control (RBAC), also known as *role-based security*, is a method used in computer systems security to restrict access to the network and resources of a system to authorized users only. Kubernetes has a native API to control roles at the namespace and cluster level and associate them with specific resources and individuals.

CRD
A *custom resource definition* (CRD) is an extension of the Kubernetes API and allows developers to create new data types and objects, *called custom resources*.

Operator
An *operator* is a custom resource that automates those steps that are normally performed by a human operator when managing one or more applications or given services. An operator assists Kubernetes in making sure that the resource' s defined state always matches the observed one.

kubectl
`kubectl` is the command-line tool used to manage a Kubernetes cluster.

CloudNativePG requires a Kubernetes version supported by the community. Please refer to the *Supported releases* page for details.

PostgreSQL terminology

Instance : A Postgres server process running and listening on a pair "IP address(es)" and "TCP port" (usually 5432).

Primary : A PostgreSQL instance that can accept both read and write operations.

Replica : A PostgreSQL instance replicating from the only primary instance in a cluster and is kept updated by reading a stream of Write-Ahead Log (WAL) records. A replica is also known as *standby* or *secondary* server. PostgreSQL relies on physical streaming replication (async/sync) and file-based log shipping (async).

Hot Standby : PostgreSQL feature that allows a *replica* to accept read-only workloads.

Cluster : To be intended as High Availability (HA) Cluster: a set of PostgreSQL instances made up by a single primary and an optional arbitrary number of replicas.

Replica Cluster : A CloudNativePG `Cluster` that is in continuous recovery mode from a selected PostgreSQL cluster, normally residing outside the Kubernetes cluster. It is a feature that enables multi-cluster deployments in private, public, hybrid, and multi-cloud contexts.

Designated Primary : A PostgreSQL standby instance in a replica cluster that is in continuous recovery from another PostgreSQL cluster and that is designated to become primary in case the replica cluster becomes primary.

Superuser : In PostgreSQL a *superuser* is any role with both `LOGIN` and `SUPERUSER` privileges. For security reasons, CloudNativePG performs administrative tasks by connecting to the `postgres` database as the `postgres` user via `peer` authentication over the local Unix Domain Socket.

WAL
Write-Ahead Logging (WAL) is a standard method for ensuring data integrity in database management systems.

PVC group : A PVC group in CloudNativePG' s terminology is a group of related PVCs belonging to the same PostgreSQL instance, namely the main volume containing the PGDATA (`storage`) and the volume for WALs (`walStorage`).|

Cloud terminology

Region : A *region* in the Cloud is an isolated and independent geographic area organized in *availability zones*. Zones within a region have very little round-trip network latency.

Zone : An *availability zone* in the Cloud (also known as *zone*) is an area in a region where resources can be deployed. Usually, an availability zone corresponds to a data center or an isolated building of the same data center.

What to do next

Now that you have familiarized with the terminology, you can decide to

test CloudNativePG on your laptop using a local cluster before deploying the operator in your selected cloud environment.

1.4.3 Use cases

CloudNativePG has been designed to work with applications that reside in the same Kubernetes cluster, for a full cloud native experience.

However, it might happen that, while the database can be hosted inside a Kubernetes cluster, applications cannot be containerized at the same time and need to run in a *traditional environment* such as a VM.

Case 1: Applications inside Kubernetes

In a typical situation, the application and the database run in the same namespace inside a Kubernetes cluster.

The application, normally stateless, is managed as a standard `Deployment` , with multiple replicas spread over different Kubernetes node, and internally exposed through a `ClusterIP` service.

The service is exposed externally to the end user through an `Ingress` and the provider' s load balancer facility, via HTTPS.

The application uses the backend PostgreSQL database to keep track of the state in a reliable and persistent way. The application refers to the read-write service exposed by the `Cluster` resource defined by CloudNativePG, which points to the current primary instance, through a TLS connection. The `Cluster` resource embeds the logic of single primary and multiple standby architecture, hiding the complexity of managing a high availability cluster in Postgres.

Case 2: Applications outside Kubernetes

Another possible use case is to manage your PostgreSQL database inside Kubernetes, while having your applications outside of it (for example in a virtualized environment). In this case, PostgreSQL is represented by an IP address (or host name) and a TCP port, corresponding to the defined Ingress resource in Kubernetes.

The application can still benefit from a TLS connection to PostgreSQL.

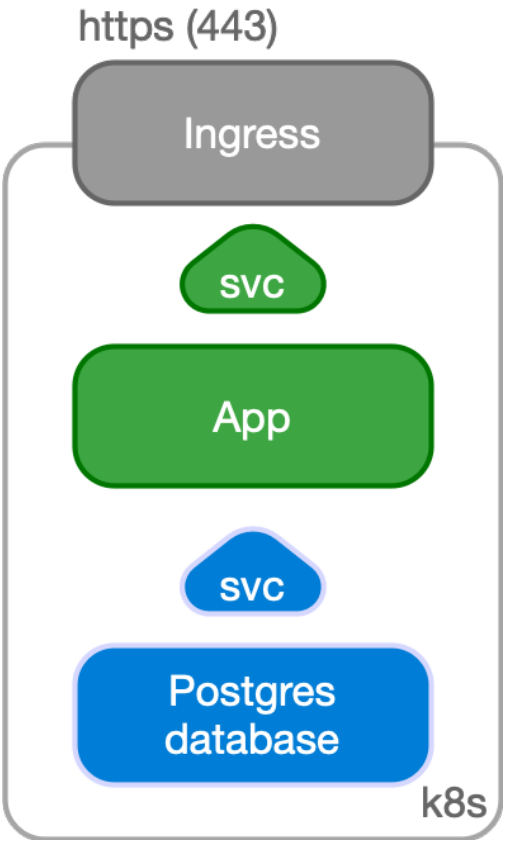


Fig. 1: Application and Database inside Kubernetes

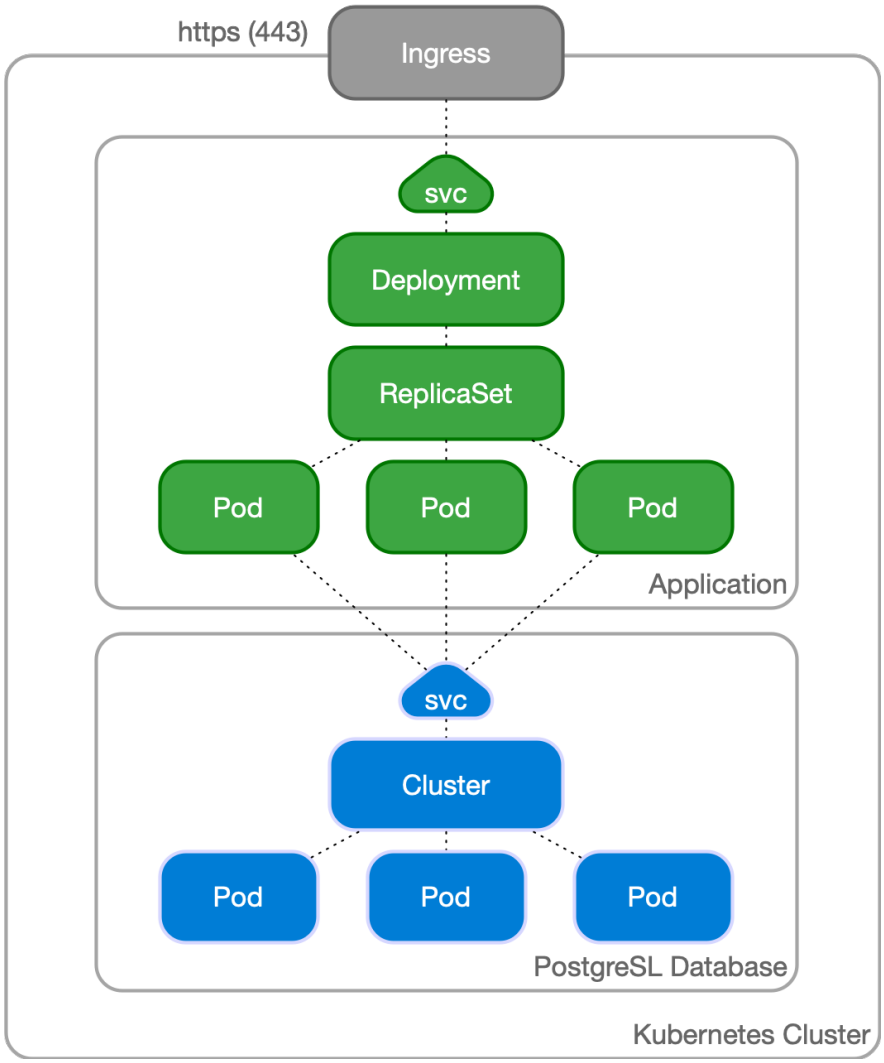


Fig. 2: Close-up view of application and database inside Kubernetes

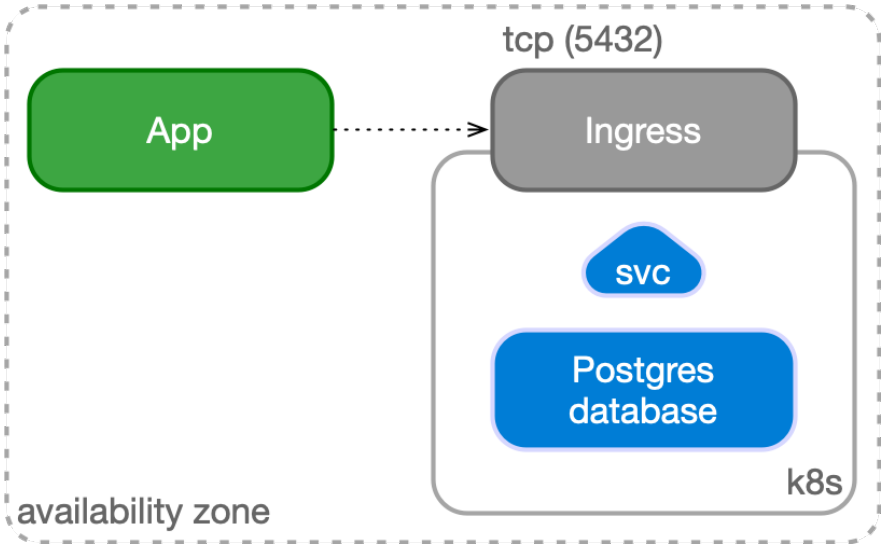


Fig. 3: Application outside Kubernetes

1.4.4 Architecture

This section covers the main architectural aspects you need to consider when deploying PostgreSQL in Kubernetes.

Important: We encourage you to read an article that we've written for the CNCF blog with title *Recommended Architectures for PostgreSQL in Kubernetes* .

Important: If you are deploying PostgreSQL in a self-managed Kubernetes environment, please make sure you read the *Kubernetes architecture*

below when you start planning your journey to the Cloud Native world.

Synchronizing the state

PostgreSQL is a database management system and, as such, it needs to be treated as a **stateful workload** in Kubernetes. While stateless applications mainly use traffic redirection to achieve High Availability (HA) and Disaster Recovery (DR), in the case of a database, state must be replicated in multiple locations, preferably in a continuous and instantaneous way, by adopting either of the following two strategies:

- *storage-level replication*, normally persistent volumes
- *application-level replication*, in this specific case PostgreSQL

CloudNativePG relies on application-level replication, for a simple reason: the PostgreSQL database management system comes with robust and reliable built-in **physical replication** capabilities based on **Write Ahead Log (WAL) shipping**, which have been used in production by millions of users all over the world for over a decade.

PostgreSQL supports both asynchronous and synchronous streaming replication over the network, as well as asynchronous file-based log shipping (normally used as a fallback option, for example, to store WAL files in an object store). Replicas are usually called *standby servers* and can also be used for read-only workloads, thanks to the *Hot Standby* feature.

Important: **We recommend against storage-level replication with PostgreSQL**, although CloudNativePG allows you to adopt that strategy. For more information, please refer to the talk given by Chris Milsted and Gabriele Bartolini at KubeCon NA 2022 entitled Data On Kubernetes, Deploying And Running PostgreSQL And Patterns For Databases In a Kubernetes Cluster

where this topic was covered in detail.

Kubernetes architecture

Kubernetes natively provides the possibility to span separate physical locations - also known as data centers, failure zones, or more frequently **availability zones** - connected to each other via redundant, low-latency, private network connectivity.

Being a distributed system, the recommended minimum number of availability zones for a Kubernetes cluster is three (3), in order to make the control plane resilient to the failure of a single zone. For details, please refer to Running in multiple zones . This means that **each data center is active at any time** and can run workloads simultaneously.

Note: Most of the public Cloud Providers' managed Kubernetes services already provide 3 or more availability zones in each region.

Multi-availability zone Kubernetes clusters

The multi-availability zone Kubernetes architecture with three (3) or more zones is the one that we recommend for PostgreSQL usage. This scenario is typical of Kubernetes services managed by Cloud Providers.

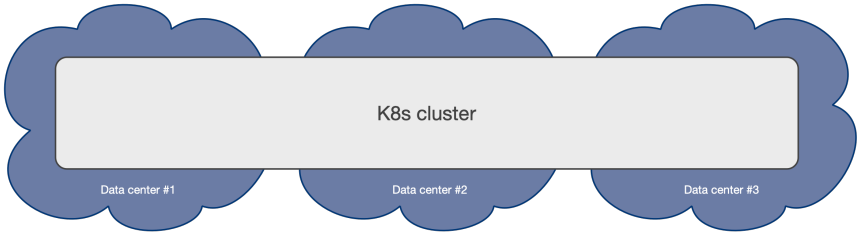


Fig. 4: Kubernetes cluster spanning over 3 independent data centers

Such an architecture enables the CloudNativePG operator to control the full lifecycle of a `Cluster` resource across the zones within a single Kubernetes cluster, by treating all the availability zones as active: this includes, among other operations, *Scheduling* the workloads in a declarative manner (based on affinity rules, tolerations and node selectors), automated failover, self-healing, and updates. All will work seamlessly across the zones in a single Kubernetes cluster.

Please refer to the *PostgreSQL architecture*

section below for details on how you can design your PostgreSQL clusters within the same Kubernetes cluster through shared-nothing deployments at the storage, worker node, and availability zone levels.

Moreover, you can take advantage of additional *Multi-availability zone Kubernetes clusters* , by using them to host "passive" PostgreSQL replica clusters. This should be used primarily for DR, read-only operations, or cross-region availability, even though failovers and promotions in this case must be done manually.

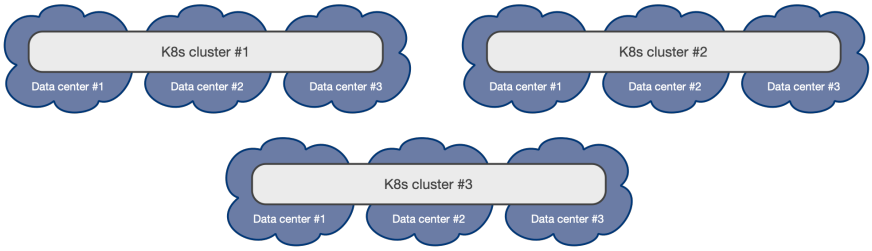


Fig. 5: Example of a multiple Kubernetes cluster architecture distributed over 3 regions each with 3 independent data centers

Single availability zone Kubernetes clusters

If your Kubernetes cluster has only one availability zone, CloudNativePG still provides you with a lot of features to improve HA and DR outcomes for your PostgreSQL databases, pushing the single point of failure (SPoF) to the level of the zone as much as possible - i.e. the zone must have an outage before your CloudNativePG clusters suffer a failure.

This scenario is typical of self-managed on-premise Kubernetes clusters, where only one data center is available.

Single availability zone Kubernetes is unfortunately the only viable option where just **two (2) data centers** are available within reach of a low latency connection (normally in the same metropolitan area): having only two zones precludes users from creating a multi-availability zone Kubernetes cluster (as the minimum number of 3 zones is not reached) and forces them to create two different Kubernetes clusters in an active/passive configuration, where the second cluster is used primarily for Disaster Recovery.

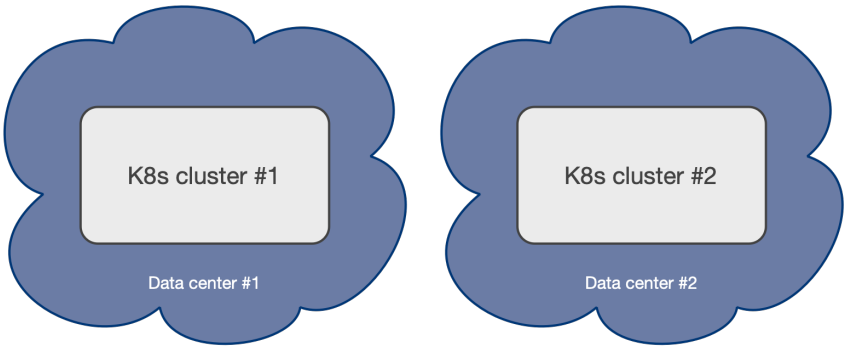


Fig. 6: Example of a Kubernetes architecture with only 2 data centers

Hint: If you are at en early stage of your Kubernetes journey, please share this document with your infrastructure team. The two data centers setup might be simply the result of a "lift-and-shift" transition to Kubernetes from a traditional bare-metal or VM based infrastructure, and the benefits that Kubernetes offers in a 3+ zone scenario might not have been known, or addressed at the time the infrastructure architecture was designed. Ultimately, a third physical location connected to the other two might represent a valid option to consider for organization, as it reduces the overall costs of the infrastructure by moving the day-to-day complexity from the application level down to the physical infrastructure level.

Please refer to the *PostgreSQL architecture*

section below for details on how you can design your PostgreSQL clusters within your single availability zone Kubernetes cluster through shared-nothing deployments at the storage and worker node levels only. For HA, in such a scenario it becomes even more important that the PostgreSQL instances be located on different worker nodes and do not share

the same storage.

For DR, you can push the SPoF above the single zone, by using additional *Multi-availability zone Kubernetes clusters* to host "passive" PostgreSQL replica clusters. As with other Kubernetes workloads in this scenario, promotion of a Kubernetes cluster as primary must be done manually. As explained below, no automated failover across Kubernetes clusters is available for PostgreSQL at the moment with CloudNativePG, as the operator can only work within a single Kubernetes cluster.

PostgreSQL architecture

CloudNativePG supports clusters based on asynchronous and synchronous streaming replication to manage multiple hot standby replicas within the same Kubernetes cluster, with the following specifications:

- One primary, with optional multiple hot standby replicas for HA
- Available services for applications:
  - `-rw` : applications connect only to the primary instance of the cluster
  - `-ro` : applications connect only to hot standby replicas for read-only-workloads
  - `-r` : applications connect to any of the instances for read-only workloads
- Shared-nothing architecture recommended for better resilience of the PostgreSQL cluster:
  - PostgreSQL instances should reside on different Kubernetes worker nodes and share only the network - as a result, instances should not share the storage and preferably use local volumes attached to the node they run on
  - PostgreSQL instances should reside in different availability zones within the same Kubernetes cluster / region

The below diagram provides a simplistic view of the recommended shared-nothing architecture for a PostgreSQL cluster spanning across 3 different availability zones, running on separate nodes, each with dedicated local storage for PostgreSQL data.

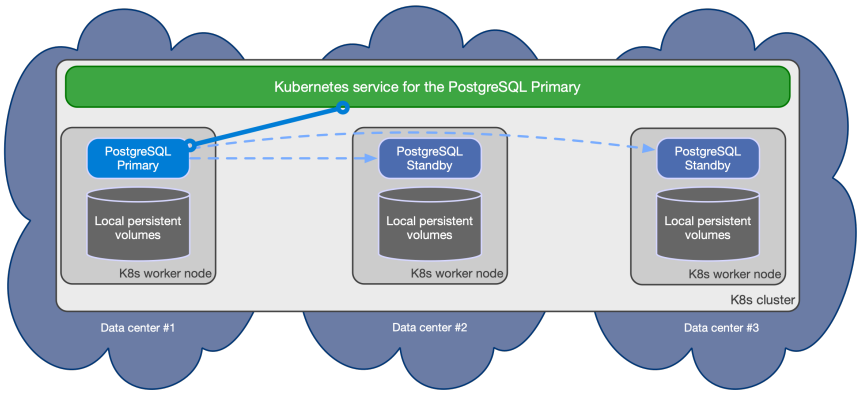


Fig. 7: Bird-eye view of the recommended shared nothing architecture for PostgreSQL in Kubernetes

CloudNativePG automatically takes care of updating the above services if the topology of the cluster changes. For example, in case of failover, it automatically updates the `-rw` service to point to the promoted primary, making sure that traffic from the applications is seamlessly redirected.

See also:

Please refer to the *About the replication user* for more information about how CloudNativePG relies on PostgreSQL replication, including synchronous settings.

See also:

Please refer to the *Connecting from an application* for information about how to connect to CloudNativePG from a stateless application within the same Kubernetes cluster.

See also:

Please refer to the *Connection pooling* for information about how to take advantage of PgBouncer as a connection pooler, and create an access layer between your applications and the PostgreSQL clusters.

Read-write workloads

Applications can decide to connect to the PostgreSQL instance elected as *current primary* by the Kubernetes operator, as depicted in the following diagram:

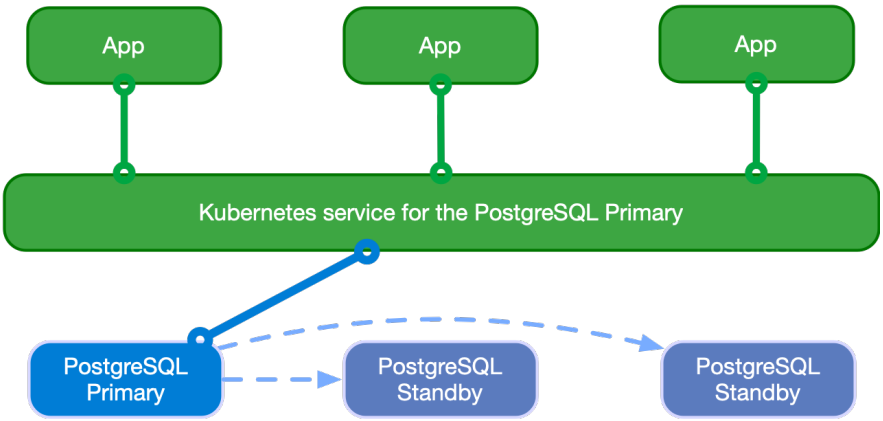


Fig. 8: Applications writing to the single primary

Applications can use the `-rw` suffix service.

In case of temporary or permanent unavailability of the primary, for High Availability purposes CloudNativePG will trigger a failover, pointing the `-rw` service to another instance of the cluster.

Read-only workloads

Important: Applications must be aware of the limitations that Hot Standby

presents and familiar with the way PostgreSQL operates when dealing with these workloads.

Applications can access hot standby replicas through the `-ro` service made available by the operator. This service enables the application to offload read-only queries from the primary node.

The following diagram shows the architecture:

Applications can also access any PostgreSQL instance through the `-r` service.

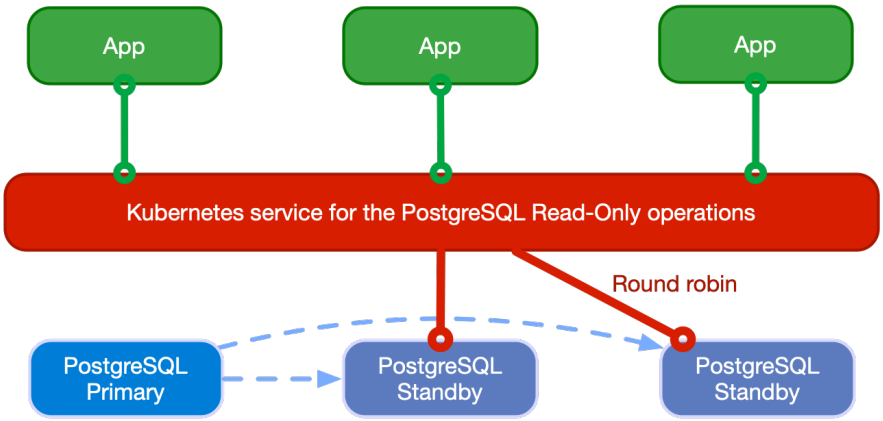


Fig. 9: Applications reading from hot standby replicas in round robin

Deployments across Kubernetes clusters

Note: CloudNativePG supports deploying PostgreSQL across multiple Kubernetes clusters through a feature called **Replica Cluster**, which is described in this section.

In a distributed PostgreSQL cluster there can only be a single PostgreSQL instance acting as a primary at all times. This means that applications can only write inside a single Kubernetes cluster, at any time.

However, for business continuity objectives it is fundamental to:

- reduce global **recovery point objectives** (RPO) by storing PostgreSQL backup data in multiple locations, regions and possibly using different providers (Disaster Recovery)
- reduce global **recovery time objectives** (RTO) by taking advantage of PostgreSQL replication beyond the primary Kubernetes cluster (High Availability)

In order to address the above concerns, CloudNativePG introduces the concept of a *PostgreSQL Replica Cluster*. Replica clusters are the CloudNativePG way to enable multi-cluster deployments in private, public, hybrid, and multi-cloud contexts.

A replica cluster is a separate `Cluster` resource:

1. having either `pg_basebackup` or full `recovery` as the `bootstrap` option from a defined external source cluster
2. having the `replica.enabled` option set to `true`
3. replicating from a defined external cluster identified by `replica.source` , normally located outside the Kubernetes cluster
4. replaying WAL information received from the recovery object store (using PostgreSQL' s `restore_command` parameter), or via streaming replication (using PostgreSQL' s `primary_conninfo` parameter), or any of the two (in case both the `barmanObjectStore` and `connectionParameters` are defined in the external cluster)
5. accepting only read connections, as supported by PostgreSQL' s Hot Standby

See also:

Please refer to the *Bootstrap* for more information about cloning a PostgreSQL cluster from another one (defined in the *externalClusters* section).

The diagram below depicts a PostgreSQL cluster spanning over two different Kubernetes clusters, where the primary cluster is in the first Kubernetes cluster and the replica cluster is in the second. The second Kubernetes cluster acts as the company' s disaster recovery cluster, ready to be activated in case of disaster and unavailability of the first one.

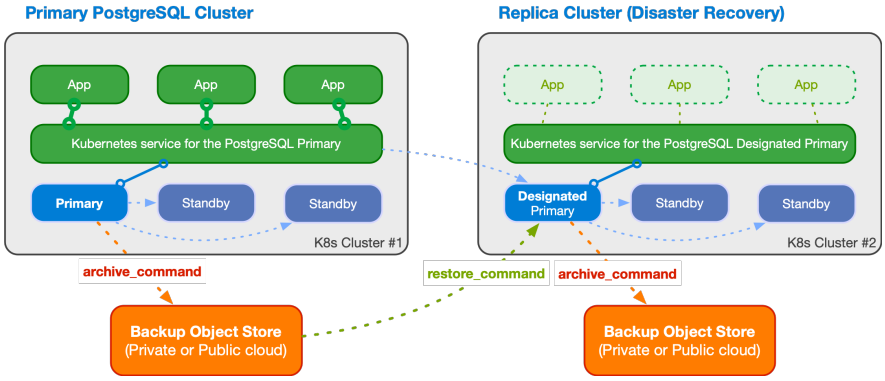


Fig. 10: An example of multi-cluster deployment with a primary and a replica cluster

A replica cluster can have the same architecture of the primary cluster. In place of the primary instance, a replica cluster has a **designated primary** instance, which is a standby server with an arbitrary number of cascading standby servers in streaming replication (symmetric architecture).

The designated primary can be promoted at any time, making the replica cluster a primary cluster capable of accepting write connections.

> **Warning:** CloudNativePG does not perform any cross-cluster switchover or failover at the moment. Such operation must be performed manually or delegated to a multi-cluster/federated cluster aware authority. Each PostgreSQL cluster is independent from any other.

The designated primary in the above example is fed via WAL streaming (`primary_conninfo`), with fallback option for file-based WAL shipping through the `restore_command` and `barman-cloud-wal-restore` .

CloudNativePG allows you to define multiple replica clusters. You can also define replica clusters with a lower number of replicas, and then increase this number when the cluster is promoted to primary.

See also:

Please refer to the *Replica clusters* for more information about physical replica clusters work and how you can configure read-only clusters in different Kubernetes cluster to improve your global disaster recovery and HA strategy.

1.4.5 Installation and upgrades

Installation on Kubernetes

Directly using the operator manifest

The operator can be installed like any other resource in Kubernetes, through a YAML manifest applied via `kubectl` .

You can install the latest operator manifest

for this minor release as follows:

```
kubectl apply --server-side -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-1.23/releases/
↪cnpg-1.23.1.yaml
```

You can verify that with:

```
kubectl get deployment -n cnpg-system cnpg-controller-manager
```

Using the `cnpg` plugin for `kubectl`

You can use the `cnpg` plugin to override the default configuration options that are in the static manifests.

For example, to generate the default latest manifest but change the watch namespaces to only be a specific namespace, you could run:

```
kubectl cnpg install generate \
  --watch-namespace "specific-namespace" \
  > cnpg_for_specific_namespace.yaml
```

Please refer to Backup {postgresql-cnpg-io-v1-Backup} documentation for a more comprehensive example.

> **Warning:** If you are deploying CloudNativePG on GKE and get an error (... *failed to call webhook...*), be aware that by default traffic between worker nodes and control plane is blocked by the firewall except for a few specific ports, as explained in the official docs

and by this issue . You' ll need to either change the `targetPort` in the webhook service, to be one of the allowed ones, or open the webhooks' port (`9443`) on the firewall.

Testing the latest development snapshot

If you want to test or evaluate the latest development snapshot of CloudNativePG before the next official patch release, you can download the manifests from the cloudnative-pg/artifacts

which provides easy access to the current trunk (main) as well as to each supported release.

For example, you can install the latest snapshot of the operator with:

```
curl -sSfL \
  https://raw.githubusercontent.com/cloudnative-pg/artifacts/main/manifests/operator-
↪manifest.yaml | \
  kubectl apply --server-side -f -
```

If you are instead looking for the latest snapshot of the operator for this specific minor release, you can just run:

```
curl -sSfL \
  https://raw.githubusercontent.com/cloudnative-pg/artifacts/release-1.23/manifests/
↪operator-manifest.yaml | \
  kubectl apply --server-side -f -
```

Important: Snapshots are not supported by the CloudNativePG and not intended for production usage.

Using the Helm Chart

The operator can be installed using the provided Helm chart .

Using OLM

CloudNativePG can also be installed using the Operator Lifecycle Manager (OLM)

directly from OperatorHub.io .

Details about the deployment

In Kubernetes, the operator is by default installed in the `cnpg-system` namespace as a Kubernetes `Deployment` . The name of this deployment depends on the installation method. When installed through the manifest or the `cnpg` plugin, it is called `cnpg-controller-manager` by default. When installed via Helm, the default name is `cnpg-cloudnative-pg` .

Note: With Helm you can customize the name of the deployment via the *fullnameOverride* field in the *"values.yaml"* file .

You can get more information using the `describe` command in `kubectl` :

```
$ kubectl get deployments -n cnpg-system
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE
<deployment-name>    1/1     1            1           18m
```

```
kubectl describe deploy \
  -n cnpg-system \
  <deployment-name>
```

As with any Deployment, it sits on top of a ReplicaSet and supports rolling upgrades. The default configuration of the CloudNativePG operator comes with a Deployment of a single replica, which is suitable for most installations. In case the node where the pod is running is not reachable anymore, the pod will be rescheduled on another node.

If you require high availability at the operator level, it is possible to specify multiple replicas in the Deployment configuration - given that the operator supports leader election. Also, you can take advantage of taints and tolerations to make sure that the operator does not run on the same nodes where the actual PostgreSQL clusters are running (this might even include the control plane for self-managed Kubernetes installations).

See also:

You can change the default behavior of the operator by overriding some default options. For more information, please refer to the *Operator configuration* section.

Upgrades

Important: Please carefully read the *Release notes for 1.14.0 and earlier*

before performing an upgrade as some versions might require extra steps.

Upgrading CloudNativePG operator is a two-step process:

1. upgrade the controller and the related Kubernetes resources
2. upgrade the instance manager running in every PostgreSQL pod

Unless differently stated in the release notes, the first step is normally done by applying the manifest of the newer version for plain Kubernetes installations, or using the native package manager of the used distribution (please follow the instructions in the above sections).

The second step is automatically executed after having updated the controller, by default triggering a rolling update of every deployed PostgreSQL instance to use the new instance manager. The rolling update procedure culminates with a switchover, which is controlled by the `primaryUpdateStrategy` option, by default set to `unsupervised` . When set to `supervised` , users need to complete the rolling update by manually promoting a new instance through the `cnpg` plugin for `kubectl` .

See also:

This process is discussed in-depth on the *Rolling Updates* page.

Important: In case *primaryUpdateStrategy* is set to the default value of *unsupervised* , an upgrade of the operator will trigger a switchover on your PostgreSQL cluster, causing a (normally negligible) downtime.

Since version 1.10.0, the rolling update behavior can be replaced with in-place updates of the instance manager. The latter don' t require a restart of the PostgreSQL instance and, as a result, a switchover in the cluster. This behavior, which is disabled by default, is described below.

In-place updates of the instance manager

By default, CloudNativePG issues a rolling update of the cluster every time the operator is updated. The new instance manager shipped with the operator is added to each PostgreSQL pod via an init container.

However, this behavior can be changed via configuration to enable in-place updates of the instance manager, which is the PID 1 process that keeps the container alive.

Internally, any instance manager from version 1.10 of CloudNativePG supports injection of a new executable that will replace the existing one, once the integrity verification phase is completed, as well as graceful termination of all the internal processes. When the new instance manager restarts using the new binary, it adopts the already running *postmaster*.

As a result, the PostgreSQL process is unaffected by the update, refraining from the need to perform a switchover. The other side of the coin, is that the Pod is changed after the start, breaking the pure concept of immutability.

You can enable this feature by setting the `ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES` environment variable to `'true'` in the *Operator configuration* .

The in-place upgrade process will not change the init container image inside the Pods. Therefore, the Pod definition will not reflect the current version of the operator.

Compatibility among versions

CloudNativePG follows semantic versioning. Every release of the operator within the same API version is compatible with the previous one. The current API version is v1, corresponding to versions 1.x.y of the operator.

In addition to new features, new versions of the operator contain bug fixes and stability enhancements. Because of this, **we strongly encourage users to upgrade to the latest version of the operator**, as each version is released in order to maintain the most secure and stable Postgres environment.

CloudNativePG currently releases new versions of the operator at least monthly. If you are unable to apply updates as each version becomes available, we recommend upgrading through each version in sequential order to come current periodically and not skipping versions.

The *Release notes for 1.14.0 and earlier* page contains a detailed list of the changes introduced in every released version of CloudNativePG, and it must be read before upgrading to a newer version of the software.

Most versions are directly upgradable and in that case, applying the newer manifest for plain Kubernetes installations or using the native package manager of the chosen distribution is enough.

When versions are not directly upgradable, the old version needs to be removed before installing the new one. This won't affect user data but only the operator itself.

Upgrading to 1.23.0, 1.22.3 or 1.21.5

Important: We encourage all existing users of CloudNativePG to upgrade to version 1.23.0 or at least to the latest stable version of the minor release you are currently using (namely 1.22.2 or 1.21.4).

Warning: Every time you are upgrading to a higher minor release, make sure you go through the release notes and upgrade instructions of all the intermediate minor releases. For example, if you want to move from 1.21.x to 1.23, make sure you go through the release notes and upgrade instructions for 1.22 and 1.23.

User defined replication slots

CloudNativePG now offers automated synchronization of all replication slots defined on the primary to any standby within the High Availability (HA) cluster.

If you manually manage replication slots on a standby, it is essential to exclude those replication slots from synchronization. Failure to do so may result in CloudNativePG removing them from the standby. To implement this exclusion, utilize the following YAML configuration. In this example, replication slots with a name starting with 'foo' are prevented from synchronization:

```yaml
...
  replicationSlots:
    synchronizeReplicas:
      enabled: true
      excludePatterns:
      - "^foo"
```

Alternatively, if you prefer to disable the synchronization mechanism entirely, use the following configuration:

```
...
  replicationSlots:
    synchronizeReplicas:
      enabled: false
```

Server-side apply of manifests

To ensure compatibility with Kubernetes 1.29 and upcoming versions, CloudNativePG now mandates the utilization of Server-side apply

when deploying the operator manifest.

While employing this installation method poses no challenges for new deployments, updating existing operator manifests using the `--server-side` option may result in errors resembling the example below:

```
Apply failed with 1 conflict: conflict with "kubectl-client-side-apply" using..
```

If such errors arise, they can be resolved by explicitly specifying the `--force-conflicts` option to enforce conflict resolution:

```
kubectl apply --server-side --force-conflicts -f <OPERATOR_MANIFEST>
```

Henceforth, `kube-apiserver` will be automatically acknowledged as a recognized manager for the CRDs, eliminating the need for any further manual intervention on this matter.

Upgrading to 1.22 from a previous minor version

CloudNativePG continues to adhere to the security-by-default approach. As of version 1.22, the usage of the `ALTER SYSTEM` command is now disabled by default.

The reason behind this choice is to ensure that, by default, changes to the PostgreSQL configuration in a database cluster controlled by CloudNativePG are allowed only through the Kubernetes API.

At the same time, we are providing an option to enable `ALTER SYSTEM` if you need to use it, even temporarily, from versions 1.22.0, 1.21.2, and 1.20.5, by setting `.spec.postgresql.enableAlterSystem` to `true` , as in the following excerpt:

```
...
  postgresql:
    enableAlterSystem: true
...
```

Clusters in 1.22 will have `enableAlterSystem` set to `false` by default. If you want to retain the existing behavior, in 1.22, you need to explicitly set `enableAlterSystem` to `true` as shown above.

In versions 1.21.2 and 1.20.5, and later patch releases in the 1.20 and 1.21 branches, `enableAlterSystem` will be set to `true` by default, keeping with the existing behavior. If you don't need to use `ALTER SYSTEM` , we recommend that you set `enableAlterSystem` explicitly to `false` .

Important: You can set the desired value for *enableAlterSystem* immediately following your upgrade to version 1.22.0, 1.21.2, or 1.20.5, as shown in the example above.

Upgrading to 1.21 from a previous minor version

With the goal to keep improving out-of-the-box the *convention over configuration* behavior of the operator, CloudNativePG changes the default value of several knobs in the following areas:

- startup and shutdown control of the PostgreSQL instance
- self-healing
- security
- labels

Warning: Please read carefully the list of changes below, and how to modify the *Cluster* manifests to retain the existing behavior if you don't want to disrupt your existing workloads. Alternatively, postpone the upgrade to until you are sure. In general, we recommend adopting these default values unless you have valid reasons not to.

Superuser access disabled

Pushing towards *security-by-default*, CloudNativePG now disables access `postgres` superuser access via the network in all new clusters, unless explicitly enabled.

If you want to ensure superuser access to the PostgreSQL cluster, regardless which version of CloudNativePG you are running, we advise you to explicitly declare it by setting:

```
spec:
   ...
   enableSuperuserAccess: true
```

Replication slots for HA

Replication slots for High Availability are enabled by default.

If you want to ensure replication slots are disabled, regardless of which version of CloudNativePG you are running, we advise you to explicitly declare it by setting:

```
spec:
   ...
   replicationSlots:
     highAvailability:
       enabled: false
```

Delay for PostgreSQL shutdown

Up to 1.20.2, the `stopDelay parameter <Postgres instance manager>`

was set to 30 seconds. Despite the recommendations to change and tune this value, almost all the cases we have examined during support incidents or community issues show that this value is left unchanged.

The new default value is 1800 seconds , the equivalent of 30 minutes.

The new `smartShutdownTimeout` parameter has been introduced to define the maximum time window within the `stopDelay` value reserved to complete the `smart` shutdown procedure in PostgreSQL. During this time, the Postgres server rejects any new connections while waiting for all regular sessions to terminate.

Once elapsed, the remaining time up to `stopDelay` will be reserved for PostgreSQL to complete its duties regarding WAL commitments with both the archive and the streaming replicas to ensure the cluster doesn't lose any data.

If you want to retain the old behavior, you need to set explicitly:

```
spec:
   ...
   stopDelay: 30
```

And, **after** the upgrade has completed, specify `smartShutdownTimeout` :

```
spec:
   ...
   stopDelay: 30
   smartShutdownTimeout: 15
```

Delay for PostgreSQL startup

Up to 1.20.2, the `startDelay parameter <Postgres instance manager>`

was set to 30 seconds, and CloudNativePG used this parameter as `initialDelaySeconds` for the Kubernetes liveness probe. Given that all the supported Kubernetes releases provide startup probes , `startDelay` is now automatically divided into periods of 10 seconds of duration each.

Important: In order to add the *startupProbe* , each pod needs to be restarted. As a result, when you upgrade the operator, a one-time rolling update of the cluster will be executed even in the online update case.

Despite the recommendations to change and tune this value, almost all the cases we have examined during support incidents or community issues show that this value is left unchanged. Given that this parameter influences the startup of a PostgreSQL instance, a low value of `startDelay` would cause Postgres never to reach a consistent recovery state and be restarted indefinitely.

For this reason, `startDelay` has been raised by default to 3600 seconds , the equivalent of 1 hour.

If you want to retain the existing behavior using the new implementation, you can do that by explicitly setting:

```
spec:
   ...
   startDelay: 30
```

Delay for PostgreSQL switchover

Up to 1.20.2, the `switchoverDelay parameter <Postgres instance manager>`

was set by default to 40000000 seconds (over 15 months) to simulate a very long interval.

The default value has been lowered to 3600 seconds , the equivalent of 1 hour.

If you want to retain the old behavior, you need to set explicitly:

```
spec:
   ...
   switchoverDelay: 40000000
```

Labels

In version 1.18, we deprecated the `postgresql` label in pods to identify the name of the cluster, and replaced it with the more canonical `cnpg.io/cluster` label. The `postgresql` label is no longer maintained.

Similarly, from this version, the `role` label is deprecated. The new label `cnpg.io/instanceRole` is now used, and will entirely replace the `role` label in a future release.

Shortcut for keeping the existing behavior

If you want to explicitly keep the behavior of CloudNativePG up to version 1.20.2 (we advise not to), you need to set these values in all your `Cluster` definitions **before upgrading** to a higher version:

```
spec:
   ...
   # Changed in 1.21.0, 1.20.3 and 1.19.5
   startDelay: 30
   stopDelay: 30
   switchoverDelay: 40000000
   # Changed in 1.21.0 only
   enableSuperuserAccess: true
   replicationSlots:
     highAvailability:
       enabled: false
```

Once the upgrade is completed, also add:

```
spec:
   ...
   smartShutdownTimeout: 15
```

1.4.6 Quickstart

This section describes how to test a PostgreSQL cluster on your laptop/computer using CloudNativePG on a local Kubernetes cluster in

Kind or Minikube .

Warning: The instructions contained in this section are for demonstration, testing, and practice purposes only and must not be used in production.

Like any other Kubernetes application, CloudNativePG is deployed using regular manifests written in YAML.

By following the instructions on this page you should be able to start a PostgreSQL cluster on your local Kubernetes installation and experiment with it.

Important: Make sure that you have *kubectl* installed on your machine in order to connect to the Kubernetes cluster. Please follow the Kubernetes documentation on *how to install `kubectl* <https://kubernetes.io/docs/tasks/tools/install-kubectl/>`_ .

Part 1: Setup the local Kubernetes playground

The first part is about installing Minikube or Kind. Please spend some time reading about the systems and decide which one to proceed with. After setting up one of them, please proceed with part 2.

We also provide instructions for setting up monitoring with Prometheus and Grafana for local testing/evaluation, in *Part 4: Monitor clusters with Prometheus and Grafana*

Minikube

Minikube is a tool that makes it easy to run Kubernetes locally. Minikube runs a single-node Kubernetes cluster inside a Virtual Machine (VM) on your laptop for users looking to try out Kubernetes or develop with it day-to-day. Normally, it is used in conjunction with VirtualBox.

You can find more information in the official Kubernetes documentation on how to install Minikube in your local personal environment. When you installed it, run the following command to create a minikube cluster:

```
minikube start
```

This will create the Kubernetes cluster, and you will be ready to use it. Verify that it works with the following command:

```
kubectl get nodes
```

You will see one node called `minikube` .

Kind

If you do not want to use a virtual machine hypervisor, then Kind is a tool for running local Kubernetes clusters using Docker container "nodes" (Kind stands for "Kubernetes IN Docker" indeed).

Install `kind` on your environment following the instructions in the
Quickstart , then create a Kubernetes cluster with:

```
kind create cluster --name pg
```

Part 2: Install CloudNativePG

Now that you have a Kubernetes installation up and running on your laptop, you can proceed with CloudNativePG installation.

Please refer to the *Installation and upgrades* section and then proceed with the deployment of a PostgreSQL cluster.

Part 3: Deploy a PostgreSQL cluster

As with any other deployment in Kubernetes, to deploy a PostgreSQL cluster you need to apply a configuration file that defines your desired `Cluster` .

The sample file defines a simple `Cluster` using the default storage class to allocate disk space:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3

  storage:
    size: 1Gi
```

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3

  storage:
    size: 1Gi
```

In order to create the 3-node PostgreSQL cluster, you need to run the following command:

```
kubectl apply -f cluster-example.yaml
```

You can check that the pods are being created with the `get pods` command:

```
kubectl get pods
```

That will look for pods in the default namespace. To separate your cluster from other workloads on your Kubernetes installation, you could always create a new namespace to deploy clusters on. Alternatively, you can use labels. The operator will apply the `cnpg.io/cluster` label on all objects relevant to a particular cluster. For example:

```
kubectl get pods -l cnpg.io/cluster=<CLUSTER>
```

Important: Note that we are using *cnpg.io/cluster* as the label. In the past you may have seen or used *postgresql* . This label is being deprecated, and will be dropped in the future. Please use *cngp.io/cluster* .

By default, the operator will install the latest available minor version of the latest major version of PostgreSQL when the operator was released. You can override this by setting the `imageName` key in the `spec` section of the `Cluster` definition. For example, to install PostgreSQL 13.6:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
   # [...]
spec:
   # [...]
   imageName: ghcr.io/cloudnative-pg/postgresql:13.6
   #[...]
```

Important: The immutable infrastructure paradigm requires that you always point to a specific version of the container image. Never use tags like *latest* or *13* in a production environment as it might lead to unpredictable scenarios in terms of update policies and version consistency in the cluster. For strict deterministic and repeatable deployments, you can add the digests to the image name, through the *<image>:<tag>@sha256:<digestValue>* format.

Part 4: Monitor clusters with Prometheus and Grafana

Important: Installing Prometheus and Grafana is beyond the scope of this project. The instructions in this section are provided for experimentation and illustration only.

In this section we show how to deploy Prometheus and Grafana for observability, and how to create a Grafana Dashboard to monitor CloudNativePG clusters, and a set of Prometheus Rules defining alert conditions.

We leverage the Kube-Prometheus stack , Helm chart, which is maintained by the
Prometheus Community . Please refer to the project website for additional

documentation and background.

The Kube-Prometheus-stack Helm chart installs the Prometheus Operator , including the Alert Manager , and a Grafana deployment.

We include a configuration file for the deployment of this Helm chart that will provide useful initial settings for observability of CloudNativePG clusters.

Installation

If you don't have Helm installed yet, please follow the
instructions to install it in your system.

We need to add the `prometheus-community` helm chart repository, and then install the *Kube Prometheus stack* using the sample configuration we provide:

We can accomplish this with the following commands:

```
helm repo add prometheus-community \
  https://prometheus-community.github.io/helm-charts

helm upgrade --install \
  -f https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/main/docs/src/
↪samples/monitoring/kube-stack-config.yaml \
  prometheus-community \
  prometheus-community/kube-prometheus-stack
```

After completion, you will have Prometheus, Grafana and Alert Manager installed with values from the `kube-stack-config.yaml` file:

- From the Prometheus installation, you will have the Prometheus Operator watching for **any** `PodMonitor` (see **monitoring**).
- The Grafana installation will be watching for a Grafana dashboard `ConfigMap` .

See also:

For further information about the above command see the helm install documentation.

You can see several Custom Resources have been created:

```
% kubectl get crds
NAME                                          CREATED AT
...
alertmanagers.monitoring.coreos.com           <timestamp>
...
prometheuses.monitoring.coreos.com            <timestamp>
prometheusrules.monitoring.coreos.com         <timestamp>
...
```

as well as a series of Services:

```
% kubectl get svc
NAME                                        TYPE        PORT(S)
...                                          ...          ...
prometheus-community-grafana                ClusterIP   80/TCP
prometheus-community-kube-alertmanager      ClusterIP   9093/TCP
prometheus-community-kube-operator          ClusterIP   443/TCP
prometheus-community-kube-prometheus        ClusterIP   9090/TCP
```

Viewing with Prometheus

At this point, a CloudNativePG cluster deployed with Monitoring activated would be observable via Prometheus.

For example, you could deploy a simple cluster with `PodMonitor` enabled:

```
kubectl apply -f - <<EOF
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-with-metrics
spec:
  instances: 3

  storage:
    size: 1Gi

  monitoring:
    enablePodMonitor: true
EOF
```

To access Prometheus, port-forward the Prometheus service:

```
kubectl port-forward svc/prometheus-community-kube-prometheus 9090
```

Then access the Prometheus console locally at: http://localhost:9090/

Assuming that the monitoring stack was successfully deployed, and you have a Cluster with `enablePodMonitor: true` you should find a series of metrics relating to CloudNativePG clusters. Again, please refer to the **monitoring section** for more information.

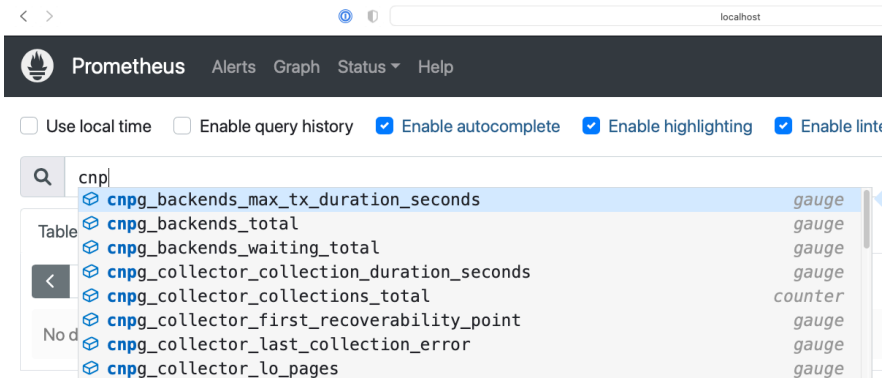


Fig. 11: local prometheus

You can now define some alerts by creating a `prometheusRule` :

```
kubectl apply -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/main/docs/src/samples/
↪monitoring/prometheusrule.yaml
```

You should see the default alerts now:

```
% kubectl get prometheusrules
NAME                                                    AGE
cnpg-default-alerts                                     3m27s
```

In the Prometheus console, you can click on the *Alerts* menu to see the alerts we just installed.

Grafana Dashboard

In our "plain" installation, Grafana is deployed with no predefined dashboards.

You can port-forward:

```
kubectl port-forward svc/prometheus-community-grafana 3000:80
```

And access Grafana locally at http://localhost:3000/

providing the credentials `admin` as username, `prom-operator` as password (defined in `kube-stack-config.yaml`).

CloudNativePG provides a default dashboard for Grafana as part of the official Helm chart . You can also download the grafana-dashboard.json

file and manually importing it via the GUI.

> **Warning:** Some graphs in the previous dashboard make use of metrics that are in alpha stage by the time this was created, like *kubelet_volume_stats_available_bytes* and *kubelet_volume_stats_capacity_bytes* producing some graphs to show *No data* .

Fig. 12: local grafana

Note that in our example setup, both Prometheus and Grafana will pick up any other CloudNativePG clusters deployed with Monitoring activated.

1.4.7 Image Catalog

`ImageCatalog` and `ClusterImageCatalog` are essential resources that empower you to define images for creating a `Cluster` .

The key distinction lies in their scope: an `ImageCatalog` is namespaced, while a `ClusterImageCatalog` is cluster-scoped.

Both share a common structure, comprising a list of images, each equipped with a `major` field indicating the major version of the image.

> **Warning:** The operator places trust in the user-defined major version and refrains from conducting any PostgreSQL version detection. It is the user's responsibility to ensure alignment between the declared major version in the catalog and the PostgreSQL image.

The `major` field' s value must remain unique within a catalog, preventing duplication across images. Distinct catalogs, however, may expose different images under the same `major` value.

Example of a Namespaced ``ImageCatalog`` :

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: ImageCatalog
metadata:
  name: postgresql
  namespace: default
spec:
  images:
    - major: 15
      image: ghcr.io/cloudnative-pg/postgresql:15.6
    - major: 16
      image: ghcr.io/cloudnative-pg/postgresql:16.2
```

Example of a Cluster-Wide Catalog using ``ClusterImageCatalog`` Resource:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: ClusterImageCatalog
metadata:
  name: postgresql
spec:
  images:
    - major: 15
      image: ghcr.io/cloudnative-pg/postgresql:15.6
    - major: 16
      image: ghcr.io/cloudnative-pg/postgresql:16.2
```

A `Cluster` resource has the flexibility to reference either an `ImageCatalog` or a `ClusterImageCatalog` to precisely specify the desired image.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
```

(continues on next page)

(continued from previous page)

```
  instances: 3
  imageCatalogRef:
    apiGroup: postgresql.cnpg.io
    kind: ImageCatalog
    name: postgresql
    major: 16
  storage:
    size: 1Gi
```

Clusters utilizing these catalogs maintain continuous monitoring. Any alterations to the images within a catalog trigger automatic updates for **all associated clusters** referencing that specific entry.

CloudNativePG Catalogs

The CloudNativePG project maintains `ClusterImageCatalogs` for the images it provides. These catalogs are regularly updated with the latest images for each major version. By applying the `ClusterImageCatalog.yaml` file from the CloudNativePG project' s GitHub repositories, cluster administrators can ensure that their clusters are automatically updated to the latest version within the specified major release.

PostgreSQL Container Images

You can install the latest version of the cluster catalog for the PostgreSQL Container Images

(cloudnative-pg/postgres-containers repository) with:

```
kubectl apply \
  -f https://raw.githubusercontent.com/cloudnative-pg/postgres-containers/main/Debian/
↪ClusterImageCatalog.yaml
```

PostGIS Container Images

You can install the *latest version of the cluster catalog for the PostGIS Container Images*

(*cloudnative-pg/postgis-containers* repository) with:

```
kubectl apply \
  -f https://raw.githubusercontent.com/cloudnative-pg/postgis-containers/main/PostGIS/
↪ClusterImageCatalog.yaml
```

1.4.8 Bootstrap

This section describes the options you have to create a new PostgreSQL cluster and the design rationale behind them. There are primarily two ways to bootstrap a new cluster:

- from scratch (`initdb`)
- from an existing PostgreSQL cluster, either directly (`pg_basebackup`) or indirectly through a physical base backup (`recovery`)

The `initdb` bootstrap also offers the possibility to import one or more databases from an existing Postgres cluster, even outside Kubernetes, and having a different major version of Postgres. For more detailed information about this feature, please refer to the *Importing Postgres databases* section.

Important: Bootstrapping from an existing cluster opens up the possibility to create a **replica cluster**, that is an independent PostgreSQL cluster which is in continuous recovery, synchronized with the source and that accepts read-only connections.

Warning: CloudNativePG requires both the *postgres* user and database to always exists. Using the local Unix Domain Socket, it needs to connect as *postgres* user to the *postgres* database via *peer* authentication in order to perform administrative tasks on the cluster. **DO NOT DELETE** the *postgres* user or the *postgres* database!!!

Note: CloudNativePG is gradually introducing support for *Kubernetes' native `VolumeSnapshot* API <https://github.com/cloudnative-pg/cloudnative-pg/issues/2081>`_

for both incremental and differential copy in backup and recovery operations - if supported by the underlying storage classes. Please see *Recovery from Volume Snapshot objects*

for details.

The `bootstrap` section

The *bootstrap* method can be defined in the `bootstrap` section of the cluster specification. CloudNativePG currently supports the following bootstrap methods:

- `initdb` : initialize a new PostgreSQL cluster (default)
- `recovery` : create a PostgreSQL cluster by restoring from a base backup of an existing cluster and, if needed, replaying all the available WAL files or up to a given *point in time*
- `pg_basebackup` : create a PostgreSQL cluster by cloning an existing one of the same major version using `pg_basebackup` via streaming replication protocol - useful if you want to migrate databases to CloudNativePG, even from outside Kubernetes.

Differently from the `initdb` method, both `recovery` and `pg_basebackup` create a new cluster based on another one (either offline or online) and can be used to spin up replica clusters. They both rely on the definition of external clusters.

Given that there are several possible backup methods and combinations of backup storage that the CloudNativePG operator provides, please refer to the Bootstrap from a backup (`recovery ) <Bootstrap from a backup (*recovery* )>` for guidance on each method.

See also:

Please refer to the "API reference for the `bootstrap section <Resource Types>`

for more information.

The `externalClusters` section

The `externalClusters` section provides a mechanism for specifying one or more PostgreSQL clusters associated with the current configuration. Its primary use cases include:

1. **Importing Databases:** Specify an external source to be utilized during the *importation of databases* via logical backup and restore, as part of the `initdb` bootstrap method.
2. **Cross-Region Replication:** Define a cross-region PostgreSQL cluster employing physical replication, capable of extending across distinct Kubernetes clusters or traditional VM/bare-metal environments.
3. **Recovery from Physical Base Backup:** Recover, fully or at a given Point-In-Time, a PostgreSQL cluster by referencing a physical base backup.

Note: Ongoing development will extend the functionality of *externalClusters* to accommodate additional use cases, such as logical replication and foreign servers in future releases.

As far as bootstrapping is concerned, `externalClusters` can be used to define the source PostgreSQL cluster for either the `pg_basebackup` method or the `recovery` one. An external cluster needs to have:

- a name that identifies the origin cluster, to be used as a reference via the `source` option
- at least one of the following:
  - information about streaming connection
  - information about the **recovery object store**, which is a Barman Cloud compatible object store that contains:
    - the WAL archive (required for Point In Time Recovery)
    - the catalog of physical base backups for the Postgres cluster

---

Note: A recovery object store is normally an AWS S3, or an Azure Blob Storage, or a Google Cloud Storage source that is managed by Barman Cloud.

When only the streaming connection is defined, the source can be used for the `pg_basebackup` method. When only the recovery object store is defined, the source can be used for the `recovery` method. When both are defined, any of the two bootstrap methods can be chosen.

Furthermore, in case of `pg_basebackup` or full `recovery` point in time, the cluster is eligible for replica cluster mode. This means that the cluster is continuously fed from the source, either via streaming, via WAL shipping through the PostgreSQL' s `restore_command` , or any of the two.

See also:

Please refer to the "API reference for the `externalClusters section <Resource Types>`

for more information.

Password files

Whenever a password is supplied within an `externalClusters` entry, CloudNativePG autonomously manages a PostgreSQL password file

for it, residing at `/controller/external/NAME/pgpass` in each instance.

This approach empowers CloudNativePG to securely establish connections with an external server without exposing any passwords in the connection string. Instead, the connection safely references the aforementioned file through the `passfile` connection parameter.

Bootstrap an empty cluster (`initdb`)

The `initdb` bootstrap method is used to create a new PostgreSQL cluster from scratch. It is the default one unless specified differently.

The following example contains the full structure of the `initdb` configuration:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    initdb:
      database: app
      owner: app
      secret:
        name: app-secret

  storage:
    size: 1Gi
```

The above example of bootstrap will:

1. create a new `PGDATA` folder using PostgreSQL' s native `initdb` command
2. create an *unprivileged* user named `app`
3. set the password of the latter (`app`) using the one in the `app-secret` secret (make sure that `username` matches the same name of the `owner`)
4. create a database called `app` owned by the `app` user.

Thanks to the *convention over configuration paradigm*, you can let the operator choose a default database name (`app`) and a default application user name (same as the database name), as well as randomly generate a secure password for both the superuser and the application user in PostgreSQL.

Alternatively, you can generate your password, store it as a secret, and use it in the PostgreSQL cluster - as described in the above example.

The supplied secret must comply with the specifications of the kubernetes.io/basic-auth . As a result, the `username` in the secret must match

the one of the `owner` (for the application secret) and `postgres` for the superuser one.

The following is an example of a `basic-auth` secret:

```yaml
apiVersion: v1
data:
  username: YXBw
  password: cGFzc3dvcmQ=
kind: Secret
metadata:
  name: app-secret
type: kubernetes.io/basic-auth
```

The application database is the one that should be used to store application data. Applications should connect to the cluster with the user that owns the application database.

Important: If you need to create additional users, please refer to *Declarative database role management* .

In case you don' t supply any database name, the operator will proceed by convention and create the `app` database, and adds it to the cluster definition using a *defaulting webhook*. The user that owns the database defaults to the database name instead.

The application user is not used internally by the operator, which instead relies on the superuser to reconcile the cluster with the desired status.

Passing options to `initdb`

The actual PostgreSQL data directory is created via an invocation of the `initdb` PostgreSQL command. If you need to add custom options to that command (i.e., to change the `locale` used for the template databases or to add data checksums), you can use the following parameters:

dataChecksums : When `dataChecksums` is set to `true` , CNPG invokes the `-k` option in `initdb` to enable checksums on data pages and help detect corruption by the I/O system - that would otherwise be silent (default: `false`).

encoding : When `encoding` set to a value, CNPG passes it to the `--encoding` option in `initdb` , which selects the encoding of the template database (default: `UTF8`).

localeCollate : When `localeCollate` is set to a value, CNPG passes it to the `--lc-collate` option in `initdb` . This option controls the collation order (`LC_COLLATE` subcategory), as defined in Locale Support

from the PostgreSQL documentation (default: `C`).

localeCType : When `localeCType` is set to a value, CNPG passes it to the `--lc-ctype` option in `initdb` . This option controls the collation order (`LC_CTYPE` subcategory), as defined in Locale Support

from the PostgreSQL documentation (default: `C`).

walSegmentSize : When `walSegmentSize` is set to a value, CNPG passes it to the `--wal-segsize` option in `initdb` (default: not set - defined by PostgreSQL as 16 megabytes).

Note: The only two locale options that CloudNativePG implements during the *initdb* bootstrap refer to the *LC_COLLATE* and *LC_TYPE* subcategories. The remaining locale subcategories can be configured directly in the PostgreSQL configuration, using the *lc_messages* , *lc_monetary* , *lc_numeric* , and *lc_time* parameters.

The following example enables data checksums and sets the default encoding to `LATIN1` :

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    initdb:
      database: app
      owner: app
      dataChecksums: true
      encoding: LATIN1
  storage:
    size: 1Gi
```

Warning: CloudNativePG supports another way to customize the behavior of the *initdb* invocation, using the *options* subsection. However, given that there are options that can break the behavior of the operator (such as *--auth* or *-d*), this technique is deprecated and will be removed from future versions of the API.

You can also specify a custom list of queries that will be executed once, just after the database is created and configured. These queries will be executed as the *superuser* (`postgres`), connected to the `postgres` database:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    initdb:
      database: app
      owner: app
      dataChecksums: true
      localeCollate: en_US
      localeCType: en_US
      postInitSQL:
        - CREATE DATABASE angus
  storage:
    size: 1Gi
```

Warning: Please use the *postInitSQL* , *postInitApplicationSQL* and *postInitTemplateSQL* options with extreme care, as queries are run as a superuser and can disrupt the entire cluster. An error in any of those queries interrupts the bootstrap phase, leaving the cluster incomplete.

Executing queries after initialization

Moreover, you can specify a list of Secrets and/or ConfigMaps which contains SQL script that will be executed after the database is created and configured. These SQL script will be executed using the **superuser** role (`postgres`), connected to the database specified in the `initdb` section:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    initdb:
      database: app
      owner: app
      postInitApplicationSQLRefs:
        secretRefs:
        - name: my-secret
          key: secret.sql
        configMapRefs:
        - name: my-configmap
          key: configmap.sql
  storage:
    size: 1Gi
```

Note: The SQL scripts referenced in *secretRefs* will be executed before the ones referenced in *configMapRefs* . For both sections the SQL scripts will be executed respecting the order in the list. Inside SQL scripts, each SQL statement is executed in a single exec on the server according to the PostgreSQL semantics , comments can be included, but internal command like *psql* cannot.

Warning: Please make sure the existence of the entries inside the ConfigMaps or Secrets specified in *postInitApplicationSQLRefs* , otherwise the bootstrap will fail. Errors in any of those SQL files will prevent the bootstrap phase to complete successfully.

Bootstrap from another cluster

CloudNativePG enables the bootstrap of a cluster starting from another one of the same major version. This operation can happen by connecting directly to the source cluster via streaming replication (`pg_basebackup`), or indirectly via an existing physical *base backup* (`recovery`).

The source cluster must be defined in the `externalClusters` section, identified by `name` (our recommendation is to use the same `name` of the origin cluster).

Important: By default the *recovery* method strictly uses the *name* of the cluster in the *externalClusters* section to locate the main folder of the backup data within the object store, which is normally reserved for the name of the server. You can specify a different one with the *barmanObjectStore.serverName* property (by default assigned to the value of

name in the external cluster definition).

Bootstrap from a backup (`recovery`)

Given the several possibilities, methods, and combinations that the CloudNativePG operator provides in terms of backup and recovery, please refer to the Bootstrap from a backup (`recovery ) <Bootstrap from a backup (*recovery* )>` .

Bootstrap from a live cluster (`pg_basebackup`)

The `pg_basebackup` bootstrap mode lets you create a new cluster (*target*) as an exact physical copy of an existing and **binary compatible** PostgreSQL instance (*source*), through a valid *streaming replication* connection. The source instance can be either a primary or a standby PostgreSQL server.

The primary use case for this method is represented by **migrations** to CloudNativePG, either from outside Kubernetes or within Kubernetes (e.g., from another operator).

Warning: The current implementation creates a *snapshot* of the origin PostgreSQL instance when the cloning process terminates and immediately starts the created cluster. See *Current limitations* below for details.

Similar to the case of the `recovery` bootstrap method, once the clone operation completes, the operator will take ownership of the target cluster, starting from the first instance. This includes overriding some configuration parameters, as required by CloudNativePG, resetting the superuser password, creating the `streaming_replica` user, managing the replicas, and so on. The resulting cluster will be completely independent of the source instance.

Important: Configuring the network between the target instance and the source instance goes beyond the scope of CloudNativePG documentation, as it depends on the actual context and environment.

The streaming replication client on the target instance, which will be transparently managed by `pg_basebackup` , can authenticate itself on the source instance in any of the following ways:

1. via *Username/Password authentication*
2. via TLS client certificate

The latter is the recommended one if you connect to a source managed by CloudNativePG or configured for TLS authentication. The first option is, however, the most common form of authentication to a PostgreSQL server in general, and might be the easiest way if the source instance is on a traditional environment outside Kubernetes. Both cases are explained below.

Requirements

The following requirements apply to the `pg_basebackup` bootstrap method:

- target and source must have the same hardware architecture
- target and source must have the same major PostgreSQL version
- target and source must have the same tablespaces
- source must be configured with enough `max_wal_senders` to grant access from the target for this one-off operation by providing at least one *walsender* for the backup plus one for WAL streaming

- the network between source and target must be configured to enable the target instance to connect to the PostgreSQL port on the source instance
- source must have a role with `REPLICATION LOGIN` privileges and must accept connections from the target instance for this role in `pg_hba.conf` , preferably via TLS (see *About the replication user* below)
- target must be able to successfully connect to the source PostgreSQL instance using a role with `REPLICATION LOGIN` privileges

See also:

For further information, please refer to the Planning , the pg_basebackup

and the High Availability, Load Balancing, and Replication

in the PostgreSQL documentation.

About the replication user

As explained in the requirements section, you need to have a user with either the `SUPERUSER` or, preferably, just the `REPLICATION` privilege in the source instance.

If the source database is created with CloudNativePG, you can reuse the `streaming_replica` user and take advantage of client TLS certificates authentication (which, by default, is the only allowed connection method for `streaming_replica`).

For all other cases, including outside Kubernetes, please verify that you already have a user with the `REPLICATION` privilege, or create a new one by following the instructions below.

As `postgres` user on the source system, please run:

```
createuser -P --replication streaming_replica
```

Enter the password at the prompt and save it for later, as you will need to add it to a secret in the target instance.

Note: Although the name is not important, we will use *streaming_replica* for the sake of simplicity. Feel free to change it as you like, provided you adapt the instructions in the following sections.

Username/Password authentication

The first authentication method supported by CloudNativePG with the `pg_basebackup` bootstrap is based on username and password matching.

Make sure you have the following information before you start the procedure:

- location of the source instance, identified by a hostname or an IP address and a TCP port
- replication username (`streaming_replica` for simplicity)
- password

You might need to add a line similar to the following to the `pg_hba.conf` file on the source PostgreSQL instance:

```
#  A more restrictive rule for TLS and IP of origin is recommended

host replication streaming_replica all md5
```

The following manifest creates a new PostgreSQL 16.2 cluster, called `target-db` , using the `pg_basebackup` bootstrap method to clone an external PostgreSQL cluster defined as `source-db` (in the `externalClusters` array). As you can see, the `source-db` definition points to the `source-db.foo.com` host and connects as the `streaming_replica` user, whose password is stored in the `password` key of the `source-db-replica-user` secret.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: target-db
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:16.2

  bootstrap:
    pg_basebackup:
      source: source-db

  storage:
    size: 1Gi

  externalClusters:
  - name: source-db
    connectionParameters:
      host: source-db.foo.com
      user: streaming_replica
    password:
      name: source-db-replica-user
      key: password
```

All the requirements must be met for the clone operation to work, including the same PostgreSQL version (in our case 16.2).

TLS certificate authentication

The second authentication method supported by CloudNativePG with the `pg_basebackup` bootstrap is based on TLS client certificates. This is the recommended approach from a security standpoint.

The following example clones an existing PostgreSQL cluster (`cluster-example`) in the same Kubernetes cluster.

Note: This example can be easily adapted to cover an instance that resides outside the Kubernetes cluster.

The manifest defines a new PostgreSQL 16.2 cluster called `cluster-clone-tls` , which is bootstrapped using the `pg_basebackup` method from the `cluster-example` external cluster. The host is identified by the read/write service in the same cluster, while the `streaming_replica` user is authenticated thanks to the provided keys, certificate, and certification authority information (respectively in the `cluster-example-replication` and `cluster-example-ca` secrets).

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-clone-tls
spec:
```

(continues on next page)

(continued from previous page)

```
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:16.2

  bootstrap:
    pg_basebackup:
      source: cluster-example

  storage:
    size: 1Gi

  externalClusters:
  - name: cluster-example
    connectionParameters:
      host: cluster-example-rw.default.svc
      user: streaming_replica
      sslmode: verify-full
    sslKey:
      name: cluster-example-replication
      key: tls.key
    sslCert:
      name: cluster-example-replication
      key: tls.crt
    sslRootCert:
      name: cluster-example-ca
      key: ca.crt
```  #### Configure the application database

We also support to configure the application database for cluster which bootstrap from a␣
↪live cluster, just like the case of `initdb` and `recovery` bootstrap method. If␣
↪the new cluster is created as a replica cluster (with replica mode enabled),␣
↪application database configuration will be skipped.

The following example configure the application database `app` with password in␣
↪supplied secret `app-secret` after bootstrap from a live cluster.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  bootstrap:
    pg_basebackup:
      database: app
      owner: app
      secret:
        name: app-secret
      source: cluster-example
```

With the above configuration, the following will happen after recovery is completed:

1. if database app does not exist, a new database app will be created.
2. if user app does not exist, a new user app will be created.

3. if user `app` is not the owner of database, user `app` will be granted as owner of database `app` .
4. If value of `username` match value of `owner` in secret, the password of application database will be changed to the value of `password` in secret.

Important: For a replica cluster with replica mode enabled, the operator will not create any database or user in the PostgreSQL instance, as these will be recovered from the original cluster.

Current limitations

Snapshot copy

The `pg_basebackup` method takes a snapshot of the source instance in the form of a PostgreSQL base backup. All transactions written from the start of the backup to the correct termination of the backup will be streamed to the target instance using a second connection (see the `--wal-method=stream` option for `pg_basebackup`).

Once the backup is completed, the new instance will be started on a new timeline and diverge from the source. For this reason, it is advised to stop all write operations to the source database before migrating to the target database in Kubernetes.

Important: Before you attempt a migration, you must test both the procedure and the applications. In particular, it is fundamental that you run the migration procedure as many times as needed to systematically measure the downtime of your applications in production.

1.4.9 Importing Postgres databases

This section describes how to import one or more existing PostgreSQL databases inside a brand new CloudNativePG cluster.

The import operation is based on the concept of online logical backups in PostgreSQL, and relies on `pg_dump` via a network connection to the origin host, and `pg_restore` . Thanks to native Multi-Version Concurrency Control (MVCC) and snapshots, PostgreSQL enables taking consistent backups over the network, in a concurrent manner, without stopping any write activity.

Logical backups are also the most common, flexible and reliable technique to perform major upgrades of PostgreSQL versions.

As a result, the instructions in this section are suitable for both:

- importing one or more databases from an existing PostgreSQL instance, even outside Kubernetes
- importing the database from any PostgreSQL version to one that is either the same or newer, enabling *major upgrades* of PostgreSQL (e.g. from version 11.x to version 15.x)

Warning: When performing major upgrades of PostgreSQL you are responsible for making sure that applications are compatible with the new version and that the upgrade path of the objects contained in the database (including extensions) is feasible.

In both cases, the operation is performed on a consistent **snapshot** of the origin database.

Important: For this reason we suggest to stop write operations on the source before the final import in the *Cluster* resource, as changes done to the source database after the start of the backup will not be in the destination cluster - hence why this feature is referred to as "offline import" or "offline major upgrade".

How it works

Conceptually, the import requires you to create a new cluster from scratch (*destination cluster*), using the Bootstrap an empty cluster (`initdb ) <Bootstrap an empty cluster (*initdb* )>` , and then complete the `initdb.import` subsection to import objects from an existing Postgres cluster (*source cluster*). As per PostgreSQL recommendation, we suggest that the PostgreSQL major version of the *destination cluster* is greater or equal than the one of the *source cluster*.

CloudNativePG provides two main ways to import objects from the source cluster into the destination cluster:

- **microservice approach**: the destination cluster is designed to host a single application database owned by the specified application user, as recommended by the CloudNativePG project
- **monolith approach**: the destination cluster is designed to host multiple databases and different users, imported from the source cluster

The first import method is available via the `microservice` type, while the latter by the `monolith` type.

Warning: It is your responsibility to ensure that the destination cluster can access the source cluster with a superuser or a user having enough privileges to take a logical backup with *pg_dump* . Please refer to the PostgreSQL documentation on "SQL Dump"

for further information.

The `microservice` type

With the microservice approach, you can specify a single database you want to import from the source cluster into the destination cluster. The operation is performed in 4 steps:

- `initdb` bootstrap of the new cluster
- export of the selected database (in `initdb.import.databases`) using `pg_dump -Fc`
- import of the database using `pg_restore --no-acl --no-owner` into the `initdb.database` (application database) owned by the `initdb.owner` user
- cleanup of the database dump file
- optional execution of the user defined SQL queries in the application database via the `postImportApplicationSQL` parameter
- execution of `ANALYZE VERBOSE` on the imported database

For example, the YAML below creates a new 3 instance PostgreSQL cluster (latest available major version at the time the operator was released) called `cluster-microservice` that imports the `angus` database from the `cluster-pg96` cluster (with the unsupported PostgreSQL 9.6), by connecting to the `postgres` database using the `postgres` user, via the password stored in the `cluster-pg96-superuser` secret.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
```

(continues on next page)

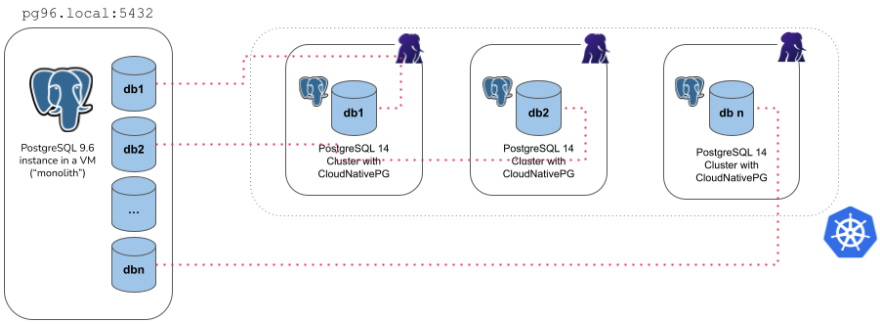


Fig. 13: Example of microservice import type

(continued from previous page)

```
  name: cluster-microservice
spec:
  instances: 3

  bootstrap:
    initdb:
      import:
        type: microservice
        databases:
          - angus
        source:
          externalCluster: cluster-pg96
        #postImportApplicationSQL:
        #- |
        #  INSERT YOUR SQL QUERIES HERE
  storage:
    size: 1Gi
  externalClusters:
    - name: cluster-pg96
      connectionParameters:
        # Use the correct IP or host name for the source database
        host: pg96.local
        user: postgres
        dbname: postgres
      password:
        name: cluster-pg96-superuser
        key: password
```

Warning: The example above deliberately uses a source database running a version of PostgreSQL that is not supported anymore by the Community, and consequently by CloudNativePG. Data export from the source instance is performed using the version of *pg_dump* in the destination cluster, which must be a supported one, and equal or greater than the source one. Based on our experience, this way of exporting data should work on older and unsupported versions of Postgres too, giving you the chance to move your legacy data to a better system, inside Kubernetes. This is the main reason why we used 9.6 in the examples of this section. We'd be interested to hear from you should you experience any issues in this area.

There are a few things you need to be aware of when using the `microservice` type:

- It requires an `externalCluster` that points to an existing PostgreSQL instance containing the data to import (for more information, please refer to The `externalClusters section <The *externalClusters* section>`)
- Traffic must be allowed between the Kubernetes cluster and the `externalCluster` during the operation
- Connection to the source database must be granted with the specified user that needs to run `pg_dump` and read roles information (*superuser* is OK)
- Currently, the `pg_dump -Fc` result is stored temporarily inside the `dumps` folder in the `PGDATA` volume, so there should be enough available space to temporarily contain the dump result on the assigned node, as well as the restored data and indexes. Once the import operation is completed, this folder is automatically deleted by the operator.
- Only one database can be specified inside the `initdb.import.databases` array
- Roles are not imported - and as such they cannot be specified inside `initdb.import.roles`

The `monolith` type

With the monolith approach, you can specify a set of roles and databases you want to import from the source cluster into the destination cluster. The operation is performed in the following steps:

- `initdb` bootstrap of the new cluster
- export and import of the selected roles
- export of the selected databases (in `initdb.import.databases`), one at a time, using `pg_dump -Fc`
- create each of the selected databases and import data using `pg_restore`
- run `ANALYZE` on each imported database
- cleanup of the database dump files

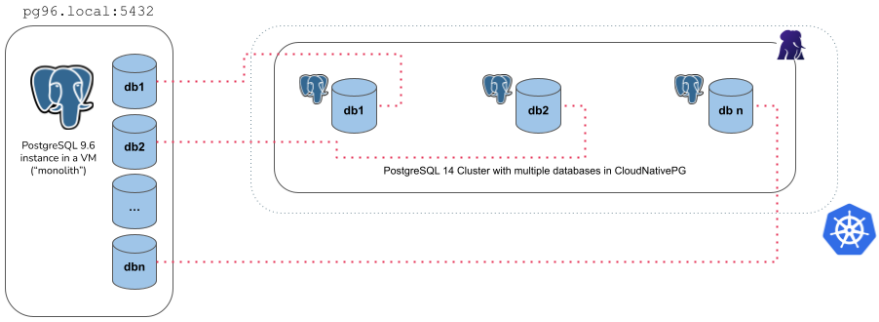


Fig. 14: Example of monolith import type

For example, the YAML below creates a new 3 instance PostgreSQL cluster (latest available major version at the time the operator was released) called `cluster-monolith` that imports the `accountant` and the `bank_user` roles, as well as the `accounting` , `banking` , `resort` databases from the `cluster-pg96` cluster (with the unsupported PostgreSQL 9.6), by connecting to the `postgres` database using the `postgres` user, via the password stored in the `cluster-pg96-superuser` secret.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-monolith
spec:
```

(continues on next page)

(continued from previous page)

```yaml
instances: 3
bootstrap:
  initdb:
    import:
      type: monolith
      databases:
        - accounting
        - banking
        - resort
      roles:
        - accountant
        - bank_user
      source:
        externalCluster: cluster-pg96
storage:
  size: 1Gi
externalClusters:
  - name: cluster-pg96
    connectionParameters:
      # Use the correct IP or host name for the source database
      host: pg96.local
      user: postgres
      dbname: postgres
      sslmode: require
    password:
      name: cluster-pg96-superuser
      key: password
```

There are a few things you need to be aware of when using the `monolith` type:

- It requires an `externalCluster` that points to an existing PostgreSQL instance containing the data to import (for more information, please refer to The `externalClusters section <The *externalClusters* section>`)
- Traffic must be allowed between the Kubernetes cluster and the `externalCluster` during the operation
- Connection to the source database must be granted with the specified user that needs to run `pg_dump` and retrieve roles information (*superuser* is OK)
- Currently, the `pg_dump -Fc` result is stored temporarily inside the `dumps` folder in the `PGDATA` volume, so there should be enough available space to temporarily contain the dump result on the assigned node, as well as the restored data and indexes. Once the import operation is completed, this folder is automatically deleted by the operator.
- At least one database to be specified in the `initdb.import.databases` array
- Any role that is required by the imported databases must be specified inside `initdb.import.roles` , with the limitations below:
  - The following roles, if present, are not imported: `postgres` , `streaming_replica` , `cnp_pooler_pgbouncer`
  - The `SUPERUSER` option is removed from any imported role
- Wildcard `"*"` can be used as the only element in the `databases` and/or `roles` arrays to import every object of the kind; When matching databases the wildcard will ignore the `postgres` database, template databases, and those databases not allowing connections

- After the clone procedure is done, `ANALYZE VERBOSE` is executed for every database.
- `postImportApplicationSQL` field is not supported

Import optimizations

During the logical import of a database, CloudNativePG optimizes the configuration of PostgreSQL in order to prioritize speed versus data durability, by forcing:

- `archive_mode` to `off`
- `fsync` to `off`
- `full_page_writes` to `off`
- `max_wal_senders` to `0`
- `wal_level` to `minimal`

Before completing the import job, CloudNativePG restores the expected configuration, then runs `initdb --sync-only` to ensure that data is permanently written on disk.

Important: WAL archiving, if requested, and WAL level will be honored after the database import process has completed. Similarly, replicas will be cloned after the bootstrap phase, when the actual cluster resource starts.

There are other optimizations you can do during the import phase. Although this topic is beyond the scope of CloudNativePG, we recommend that you reduce unnecessary writes in the checkpoint area by tuning Postgres GUCs like `shared_buffers` , `max_wal_size` , `checkpoint_timeout` directly in the `Cluster` configuration.

1.4.10 Security

This section contains information about security for CloudNativePG, that are analyzed at 3 different layers: Code, Container and Cluster.

Warning: The information contained in this page must not exonerate you from performing regular InfoSec duties on your Kubernetes cluster. Please familiarize yourself with the *Overview of Cloud Native Security*

page from the Kubernetes documentation.

See also:

Please refer to *The 4C' s Security Model in Kubernetes*

blog article to get a better understanding and context of the approach EDB has taken with security in CloudNativePG.

Code

Source code of CloudNativePG is *systematically scanned* for static analysis purposes, including **security problems**, using a popular open-source linter for Go called GolangCI-Lint directly in the CI/CD pipeline. GolangCI-Lint can run several *linters* on the same source code.

One of these is Golang Security Checker , or simply `gosec` , a linter that scans the abstract syntactic tree of the source against a set of rules aimed at the discovery of well-known vulnerabilities, threats, and weaknesses hidden in the code such as hard-coded credentials, integer overflows and SQL injections - to name a few.

Important: A failure in the static code analysis phase of the CI/CD pipeline is a blocker for the entire delivery of CloudNativePG, meaning that each commit is validated against all the linters defined by GolangCI-Lint.

Container

Every container image that is part of CloudNativePG is automatically built via CI/CD pipelines following every commit. Such images include not only the operator' s, but also the operands' - specifically every supported PostgreSQL version. Within the pipelines, images are scanned with:

- Dockle : for best practices in terms of the container build process

---

Important: All operand images are automatically rebuilt once a day by our pipelines in case of security updates at the base image and package level, providing **patch level updates** for the container images that the community distributes.

The following guidelines and frameworks have been taken into account for container-level security:

- the Container Image Creation and Deployment Guide , developed by the Defense Information Systems Agency (DISA) of the United States Department of Defense (DoD)
- the *CIS Benchmark for Docker* , developed by the Center for Internet Security (CIS)

See also:

Please refer to *Security and Containers in CloudNativePG*

blog article for more information about the approach that EDB has taken on security at the container level in CloudNativePG.

Cluster

Security at the cluster level takes into account all Kubernetes components that form both the control plane and the nodes, as well as the applications that run in the cluster (PostgreSQL included).

Role Based Access Control (RBAC)

The operator interacts with the Kubernetes API server with a dedicated service account called `cnpg-manager` . In Kubernetes this is installed by default in the `cnpg-system` namespace, with a cluster role binding between this service account and the `cnpg-manager` cluster role which defines the set of rules/resources/verbs granted to the operator.

Important: The above permissions are exclusively reserved for the operator's service account to interact with the Kubernetes API server. They are not directly accessible by the users of the operator that interact only with *Cluster* , *Pooler* , *Backup* , *ScheduledBackup* , *ImageCatalog* and *ClusterImageCatalog* resources.

Below we provide some examples and, most importantly, the reasons why CloudNativePG requires full or partial management of standard Kubernetes namespaced resources.

`configmaps` : The operator needs to create and manage default config maps for the Prometheus exporter monitoring metrics.

`deployments` : The operator needs to manage a PgBouncer connection pooler using a standard Kubernetes `Deployment` resource.

`jobs` : The operator needs to handle jobs to manage different `Cluster` ' s phases.

`persistentvolumeclaims` : The volume where the `PGDATA` resides is the central element of a PostgreSQL `Cluster` resource; the operator needs to interact with the selected storage class to dynamically provision the requested volumes, based on the defined scheduling policies.

`pods` : The operator needs to manage `Cluster` ' s instances.

`secrets` : Unless you provide certificates and passwords to your `Cluster` objects, the operator adopts the "convention over configuration" paradigm by self-provisioning random generated passwords and TLS certificates, and by storing them in secrets.

`serviceaccounts` : The operator needs to create a service account that enables the instance manager (which is the *PID 1* process of the container that controls the PostgreSQL server) to safely communicate with the Kubernetes API server to coordinate actions and continuously provide a reliable status of the `Cluster` .

`services` : The operator needs to control network access to the PostgreSQL cluster (or the connection pooler) from applications, and properly manage failover/switchover operations in an automated way (by assigning, for example, the correct end-point of a service to the proper primary PostgreSQL instance).

`validatingwebhookconfigurations` and `mutatingwebhookconfigurations` : The operator injects its self-signed webhook CA into both webhook configurations, which are needed to validate and mutate all the resources it manages. For more details, please see the *Kubernetes documentation* .

`volumesnapshots` : The operator needs to generate `VolumeSnapshots` objects in order to take backups of a PostgreSQL server. VolumeSnapshots are read too in order to validate them before starting the restore process.

`nodes` : The operator needs to get the labels for Affinity and AntiAffinity, so it can decide in which nodes a pod can be scheduled preventing the replicas to be in the same node, specially if nodes are in different availability zones. This permission is also used to determine if a node is schedule or not, avoiding the creation of pods that cannot be created at all.

To see all the permissions required by the operator, you can run `kubectl describe clusterrole cnpg-manager` .

Calls to the API server made by the instance manager

The instance manager, which is the entry point of the operand container, needs to make some calls to the Kubernetes API server to ensure that the status of some resources is correctly updated and to access the config maps and secrets that are associated with that Postgres cluster. Such calls are performed through a dedicated `ServiceAccount` created by the operator that shares the same PostgreSQL `Cluster` resource name.

Important: The operand can only access a specific and limited subset of resources through the API server. A service account is the recommended way to access the API server from within a Pod .

For transparency, the permissions associated with the service account are defined in the roles.go

file. For example, to retrieve the permissions of a generic `mypg` cluster in the `myns` namespace, you can type the following command:

```
kubectl get role -n myns mypg -o yaml
```

Then verify that the role is bound to the service account:

```
kubectl get rolebinding -n myns mypg -o yaml
```

Important: Remember that **roles are limited to a given namespace**.

Below we provide a quick summary of the permissions associated with the service account for generic Kubernetes resources.

`configmaps` : The instance manager can only read config maps that are related to the same cluster, such as custom monitoring queries

`secrets` : The instance manager can only read secrets that are related to the same cluster, namely: streaming replication user, application user, super user, LDAP authentication user, client CA, server CA, server certificate, backup credentials, custom monitoring queries

`events` : The instance manager can create an event for the cluster, informing the API server about a particular aspect of the PostgreSQL instance lifecycle

Here instead, we provide the same summary for resources specific to CloudNativePG.

`clusters` : The instance manager requires read-only permissions, namely `get` , `list` and `watch` , just for its own `Cluster` resource

`clusters/status` : The instance manager requires to `update` and `patch` the status of just its own `Cluster` resource

`backups` : The instance manager requires `get` and `list` permissions to read any `Backup` resource in the namespace. Additionally, it requires the `delete` permission to clean up the Kubernetes cluster by removing the `Backup` objects that do not have a counterpart in the object store - typically because of retention policies

`backups/status` : The instance manager requires to `update` and `patch` the status of any `Backup` resource in the namespace

Pod Security Policies

Important: Starting from Kubernetes v1.21, the use of *PodSecurityPolicy* has been deprecated, and as of Kubernetes v1.25, it has been completely removed. Despite this deprecation, we acknowledge that the operator is currently undergoing testing in older and unsupported versions of Kubernetes. Therefore, this section is retained for those specific scenarios.

A *Pod Security Policy*

is the Kubernetes way to define security rules and specifications that a pod needs to meet to run in a cluster. For InfoSec reasons, every Kubernetes platform should implement them.

CloudNativePG does not require *privileged* mode for containers execution. The PostgreSQL containers run as `postgres` system user. No component whatsoever requires running as `root` .

Likewise, Volumes access does not require *privileges* mode or `root` privileges either. Proper permissions must be properly assigned by the Kubernetes platform and/or administrators. The PostgreSQL containers run with a read-only root filesystem (i.e. no writable layer).

The operator explicitly sets the required security contexts.

Restricting Pod access using AppArmor

You can assign an AppArmor profile to the `postgres` , `initdb` , `join` , `full-recovery` and `bootstrap-controller` containers inside every `Cluster` pod through the `container.apparmor.security.beta.kubernetes.io` annotation.

See also:

edb_notranlate_2

Warning: Using this kind of annotations can result in your cluster to stop working. If this is the case, the annotation can be safely removed from the *Cluster* .

The AppArmor configuration must be at Kubernetes node level, meaning that the underlying operating system must have this option enable and properly configured.

In case this is not the situation, and the annotations were added at the `Cluster` creation time, pods will not be created. On the other hand, if you add the annotations after the `Cluster` was created the pods in the cluster will be unable to start and you will get an error like this:

```
metadata.annotations[container.apparmor.security.beta.kubernetes.io/postgres]:␣
↪Forbidden: may not add AppArmor annotations]
```

In such cases, please refer to your Kubernetes administrators and ask for the proper AppArmor profile to use.

Network Policies

The pods created by the `Cluster` resource can be controlled by Kubernetes network policies

to enable/disable inbound and outbound network access at IP and TCP level. You can find more information in the *networking document* .

Important: The operator needs to communicate to each instance on TCP port 8000 to get information about the status of the PostgreSQL server. Please make sure you keep this in mind in case you add any network policy, and refer to the "Exposed Ports" section below for a list of ports used by CloudNativePG for finer control.

Network policies are beyond the scope of this document. Please refer to the Network policies

section of the Kubernetes documentation for further information.

Exposed Ports

CloudNativePG exposes ports at operator, instance manager and operand levels, as listed in the table below:

System	Port number	Exposing	Name	Certificates	Authentication
operator	9443	webhook server	*webhook-server*	TLS	Yes
operator	8080	metrics	*metrics*	no TLS	No
instance manager	9187	metrics	*metrics*	no TLS	No
instance manager	8000	status	*status*	no TLS	No
operand	5432	PostgreSQL instance	*postgresql*	optional TLS	Yes

PostgreSQL

The current implementation of CloudNativePG automatically creates passwords and `.pgpass` files for the the database owner and, only if requested by setting `enableSuperuserAccess` to `true` , for the `postgres` superuser.

Warning: Prior to CloudNativePG 1.21, *enableSuperuserAccess* was set to *true* by default. This change has been implemented to improve the security-by-default posture of the operator, fostering a microservice approach where changes to PostgreSQL are performed in a declarative way through the *spec* of the *Cluster* resource, while providing developers with full powers inside the database through the database owner user.

As far as password encryption is concerned, CloudNativePG follows the default behavior of PostgreSQL: starting from PostgreSQL 14, `password_encryption` is by default set to `scram-sha-256` , while on earlier versions it is set to `md5` .

Important: Please refer to the Password authentication

section in the PostgreSQL documentation for details.

Note: The operator supports toggling the *enableSuperuserAccess* option. When you disable it on a running cluster, the operator will ignore the content of the secret, remove it (if previously generated by the operator) and set the password

of the *postgres* user to *NULL* (de facto disabling remote access through password authentication).

See the *Secrets* for more information.

You can use those files to configure application access to the database.

By default, every replica is automatically configured to connect in **physical async streaming replication** with the current primary instance, with a special user called `streaming_replica` . The connection between nodes is **encrypted** and authentication is via **TLS client certificates** (please refer to the *Client TLS/SSL connections* page for details). By default, the operator requires TLS v1.3 connections.

Currently, the operator allows administrators to add `pg_hba.conf` lines directly in the manifest as part of the `pg_hba` section of the `postgresql` configuration. The lines defined in the manifest are added to a default `pg_hba.conf` .

For further detail on how `pg_hba.conf` is managed by the operator, see the *PostgreSQL Configuration* of the documentation.

The administrator can also customize the content of the `pg_ident.conf` file that by default only maps the local postgres user to the postgres user in the database.

For further detail on how `pg_ident.conf` is managed by the operator, see the *PostgreSQL Configuration* of the documentation.

Important: Examples assume that the Kubernetes cluster runs in a private and secure network.

Storage

CloudNativePG delegates encryption at rest to the underlying storage class. For data protection in production environments, we highly recommend that you choose a storage class that supports encryption at rest.

1.4.11 Postgres instance manager

CloudNativePG does not rely on an external tool for failover management. It simply relies on the Kubernetes API server and a native key component called: the **Postgres instance manager**.

The instance manager takes care of the entire lifecycle of the PostgreSQL leading process (also known as `postmaster`).

When you create a new cluster, the operator makes a Pod per instance. The field `.spec.instances` specifies how many instances to create.

Each Pod will start the instance manager as the parent process (PID 1) for the main container, which in turn runs the PostgreSQL instance. During the lifetime of the Pod, the instance manager acts as a backend to handle the startup, liveness and readiness probes .

Startup, liveness and readiness probes

The startup and liveness probes rely on `pg_isready` , while the readiness probe checks if the database is up and able to accept connections using the superuser credentials.

The readiness probe is positive when the Pod is ready to accept traffic. The liveness probe controls when to restart the container once the startup probe interval has elapsed.

Important: The liveness and readiness probes will report a failure if the probe command fails three times with a 10-second interval between each check.

The liveness probe detects if the PostgreSQL instance is in a broken state and needs to be restarted. The value in `startDelay` is used to delay the probe' s execution, preventing an instance with a long startup time from being restarted.

The interval (in seconds) after the Pod has started before the liveness probe starts working is expressed in the `.spec.startDelay` parameter, which defaults to 3600 seconds. The correct value for your cluster is related to the time needed by PostgreSQL to start.

Warning: If *.spec.startDelay* is too low, the liveness probe will start working before the PostgreSQL startup is complete, and the Pod could be restarted prematurely.

Shutdown control

When a Pod running Postgres is deleted, either manually or by Kubernetes following a node drain operation, the kubelet will send a termination signal to the instance manager, and the instance manager will take care of shutting down PostgreSQL in an appropriate way. The `.spec.smartShutdownTimeout` and `.spec.stopDelay` options, expressed in seconds, control the amount of time given to PostgreSQL to shut down. The values default to 180 and 1800 seconds, respectively.

The shutdown procedure is composed of two steps:

1. The instance manager requests a **smart** shut down, disallowing any new connection to PostgreSQL. This step will last for up to `.spec.smartShutdownTimeout` seconds.
2. If PostgreSQL is still up, the instance manager requests a **fast** shut down, terminating any existing connection and exiting promptly. If the instance is archiving and/or streaming WAL files, the process will wait for up to the remaining time set in `.spec.stopDelay` to complete the operation and then forcibly shut down. Such a timeout needs to be at least 15 seconds.

Important: In order to avoid any data loss in the Postgres cluster, which impacts the database RPO, don't delete the Pod where the primary instance is running. In this case, perform a switchover to another instance first.

Shutdown of the primary during a switchover

During a switchover, the shutdown procedure is slightly different from the general case. Indeed, the operator requires the former primary to issue a **fast** shut down before the selected new primary can be promoted, in order to ensure that all the data are available on the new primary.

For this reason, the `.spec.switchoverDelay` , expressed in seconds, controls the time given to the former primary to shut down gracefully and archive all the WAL files. By default it is set to `3600` (1 hour).

> **Warning:** The *.spec.switchoverDelay* option affects the RPO and RTO of your PostgreSQL database. Setting it to a low value, might favor RTO over RPO but lead to data loss at cluster level and/or backup level. On the contrary, setting it to a high value, might remove the risk of data loss while leaving the cluster without an active primary for a longer time during the switchover.

Failover

In case of primary pod failure, the cluster will go into failover mode. Please refer to the *Automated failover* for details.

1.4.12 Scheduling

Scheduling, in Kubernetes, is the process responsible for placing a new pod on the best node possible, based on several criteria.

See also:

Please refer to the *Kubernetes documentation*

for more information on scheduling, including all the available policies. On this page we assume you are familiar with concepts like affinity, anti-affinity, node selectors, and so on.

You can control how the CloudNativePG cluster' s instances should be scheduled through the AffinityConfiguration {postgresql-cnpg-io-v1-AffinityConfiguration}

section in the definition of the cluster, which supports:

- pod affinity/anti-affinity
- node selectors
- tolerations

Note: CloudNativePG does not support pod templates for finer control on the scheduling of workloads. While they were part of the initial concept, the development team decided to postpone their introduction in a newer version of the API (most likely v2 of CNPG).

Pod affinity and anti-affinity

Kubernetes allows you to control which nodes a pod should (*affinity*) or should not (*anti-affinity*) be scheduled, based on the actual workloads already running in those nodes. This is technically known as **inter-pod affinity/anti-affinity**.

CloudNativePG by default will configure the cluster' s instances preferably on different nodes, resulting in the following `affinity` definition:

```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: postgresql
                operator: In
                values:
                  - cluster-example
          topologyKey: kubernetes.io/hostname
        weight: 100
```

As a result of the following Cluster spec:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:16.2

  affinity:
    enablePodAntiAffinity: true #default value
    topologyKey: kubernetes.io/hostname #defaul value
    podAntiAffinityType: preferred #default value

  storage:
    size: 1Gi
```

Therefore, Kubernetes will *prefer* to schedule a 3-node PostgreSQL cluster over 3 different nodes - resources permitting.

The aforementioned default behavior can be changed by tweaking the above settings.

`podAntiAffinityType` can be set to `required` : resulting in `requiredDuringSchedulingIgnoredDuringExecution` being used instead of `preferredDuringSchedulingIgnoredDuringExecution` . Please, be aware that such a strong requirement might result in pending instances in case resources are not available (which is an expected condition when using Cluster Autoscaler for automated horizontal scaling of a Kubernetes cluster).

See also:

More information on this topic is in the *Kubernetes documentation* .

Another possible value for `topologyKey` in a cloud environment can be `topology.kubernetes.io/zone` , to be sure pods will be spread across availability zones and not just nodes. Please refer to Well-Known Labels, Annotations and Taints

for more options.

You can disable the operator' s generated anti-affinity policies by setting `enablePodAntiAffinity` to false.

Additionally, in case a more fine-grained control is needed, you can specify a list of custom pod affinity or anti-affinity rules via the `additionalPodAffinity` and `additionalPodAntiAffinity` configuration attributes. These rules will be added to the ones generated by the operator, if enabled, or passed transparently otherwise.

Note: You have to pass to *additionalPodAntiAffinity* or *additionalPodAffinity* the whole content of *podAntiAffinity* or *podAffinity* that is expected by the Pod spec (please look at the following YAML as an example of having only one instance of PostgreSQL running on every worker node, regardless of which PostgreSQL cluster they belong to).

```yaml
additionalPodAntiAffinity:
  requiredDuringSchedulingIgnoredDuringExecution:
  - labelSelector:
      matchExpressions:
      - key: postgresql
        operator: Exists
        values: []
    topologyKey: "kubernetes.io/hostname"
```

Node selection through `nodeSelector`

Kubernetes allows `nodeSelector` to provide a list of labels (defined as key-value pairs) to select the nodes on which a pod can run. Specifically, the node must have each indicated key-value pair as labels for the pod to be scheduled and run.

Similarly, CloudNativePG consents you to define a `nodeSelector` in the `affinity` section, so that you can request a PostgreSQL cluster to run only on nodes that have those labels.

Tolerations

Kubernetes allows you to specify (through `taints`) whether a node should repel all pods not explicitly tolerating (through `tolerations`) their `taints` .

So, by setting a proper set of `tolerations` for a workload matching a specific node' s `taints` , Kubernetes scheduler will now take into consideration the tainted node, while deciding on which node to schedule the workload. Tolerations can be configured for all the pods of a Cluster through the `.spec.affinity.tolerations` section, which accepts the usual Kubernetes syntax for tolerations.

See also:

More information on taints and tolerations can be found in the *Kubernetes documentation* .

1.4.13 Resource management

In a typical Kubernetes cluster, pods run with unlimited resources. By default, they might be allowed to use as much CPU and RAM as needed.

CloudNativePG allows administrators to control and manage resource usage by the pods of the cluster, through the `resources` section of the manifest, with two knobs:

- `requests` : initial requirement
- `limits` : maximum usage, in case of dynamic increase of resource needs

For example, you can request an initial amount of RAM of 32MiB (scalable to 128MiB) and 50m of CPU (scalable to 100m) as follows:

```
resources:
  requests:
    memory: "32Mi"
    cpu: "50m"
  limits:
    memory: "128Mi"
    cpu: "100m"
```

Memory requests and limits are associated with containers, but it is useful to think of a pod as having a memory request and limit. The pod' s memory request is the sum of the memory requests for all the containers in the pod.

Pod scheduling is based on requests and not on limits. A pod is scheduled to run on a Node only if the Node has enough available memory to satisfy the pod' s memory request.

For each resource, we divide containers into 3 Quality of Service (QoS) classes, in decreasing order of priority:

- *Guaranteed*
- *Burstable*
- *Best-Effort*

For more details, please refer to the Configure Quality of Service for Pods

section in the Kubernetes documentation.

For a PostgreSQL workload it is recommended to set a "Guaranteed" QoS.

To avoid resources related issues in Kubernetes, we can refer to the best practices for "out of resource" handling while creating a cluster:

- Specify your required values for memory and CPU in the resources section of the manifest file. This way, you can avoid the `OOM Killed` (where "OOM" stands for Out Of Memory) and `CPU throttle` or any other resource-related issues on running instances.
- For your cluster' s pods to get assigned to the "Guaranteed" QoS class, you must set limits and requests for both memory and CPU to the same value.
- Specify your required PostgreSQL memory parameters consistently with the pod resources (as you would do in a VM or physical machine scenario - see below).
- Set up database server pods on a dedicated node using nodeSelector. See the "nodeSelector" and "tolerations" fields of the "*affinityconfiguration*" resource on the API reference page.

You can refer to the following example manifest:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-resources
spec:

  instances: 3

  postgresql:
    parameters:
      shared_buffers: "256MB"
```

(continues on next page)

(continued from previous page)

```
resources:
  requests:
    memory: "1024Mi"
    cpu: 1
  limits:
    memory: "1024Mi"
    cpu: 1

storage:
  size: 1Gi
```

In the above example, we have specified `shared_buffers` parameter with a value of `256MB` - i.e., how much memory is dedicated to the PostgreSQL server for caching data (the default value for this parameter is `128MB` in case it' s not defined).

A reasonable starting value for `shared_buffers` is 25% of the memory in your system. For example: if your `shared_buffers` is 256 MB, then the recommended value for your container memory size is 1 GB, which means that within a pod all the containers will have a total of 1 GB memory that Kubernetes will always preserve, enabling our containers to work as expected. For more details, please refer to the Resource Consumption

section in the PostgreSQL documentation.

See also:

For more details on resource management, please refer to the Managing Compute Resources for Containers

page from the Kubernetes documentation.

1.4.14 Failure Modes

This section provides an overview of the major failure scenarios that PostgreSQL can face on a Kubernetes cluster during its lifetime.

Important: In case the failure scenario you are experiencing is not covered by this section, please immediately seek for professional support .

See also:

Please refer to the *Postgres instance manager*

for more information the liveness and readiness probes implemented by CloudNativePG.

Storage space usage

The operator will instantiate one PVC for every PostgreSQL instance to store the `PGDATA` content. A second PVC dedicated to the WAL storage will be provisioned in case `.spec.walStorage` is specified during cluster initialization.

Such storage space is set for reuse in two cases:

- when the corresponding Pod is deleted by the user (and a new Pod will be recreated)
- when the corresponding Pod is evicted and scheduled on another node

If you want to prevent the operator from reusing a certain PVC you need to remove the PVC before deleting the Pod. For this purpose, you can use the following command:

```
kubectl delete -n [namespace] pvc/[cluster-name]-[serial] pod/[cluster-name]-[serial]
```

Note: If you specified a dedicated WAL volume, it will also have to be deleted during this process.

```
kubectl delete -n [namespace] pvc/[cluster-name]-[serial] pvc/[cluster-name]-[serial]-
↪wal pod/[cluster-name]-[serial]
```

For example:

```
$ kubectl delete -n default pvc/cluster-example-1 pvc/cluster-example-1-wal pod/cluster-
↪example-1
persistentvolumeclaim "cluster-example-1" deleted
persistentvolumeclaim "cluster-example-1-wal" deleted
pod "cluster-example-1" deleted
```

Failure modes

A pod belonging to a `Cluster` can fail in the following ways:

- the pod is explicitly deleted by the user;
- the readiness probe on its `postgres` container fails;
- the liveness probe on its `postgres` container fails;
- the Kubernetes worker node is drained;
- the Kubernetes worker node where the pod is scheduled fails.

Each one of these failures has different effects on the `Cluster` and the services managed by the operator.

Pod deleted by the user

The operator is notified of the deletion. A new pod belonging to the `Cluster` will be automatically created reusing the existing PVC, if available, or starting from a physical backup of the *primary* otherwise.

Important: In case of deliberate deletion of a pod, *PodDisruptionBudget* policies will not be enforced.

Self-healing will happen as soon as the *apiserver* is notified.

You can trigger a sudden failure on a given pod of the cluster using the following generic command:

```
kubectl delete -n [namespace] \
  pod/[cluster-name]-[serial] --grace-period=1
```

For example, if you want to simulate a real failure on the primary and trigger the failover process, you can run:

```
kubectl delete pod [primary pod] --grace-period=1
```

Warning: Never use *--grace-period=0* in your failover simulation tests, as this might produce misleading results with your PostgreSQL cluster. A grace period of 0 guarantees that the pod is immediately removed from the Kubernetes API server, without first ensuring that the PID 1 process of the *postgres* container (the instance manager) is shut down - contrary to what would happen in case of a real failure (e.g. unplug the power cord cable or network partitioning). As a result, the operator doesn't see the pod of the primary anymore, and triggers a failover promoting the most aligned standby, without the guarantee that the primary had been shut down.

Readiness probe failure

After 3 failures, the pod will be considered *not ready*. The pod will still be part of the `Cluster` , no new pod will be created.

If the cause of the failure can' t be fixed, it is possible to delete the pod manually. Otherwise, the pod will resume the previous role when the failure is solved.

Self-healing will happen after three failures of the probe.

Liveness probe failure

After 3 failures, the `postgres` container will be considered failed. The pod will still be part of the `Cluster` , and the *kubelet* will try to restart the container. If the cause of the failure can' t be fixed, it is possible to delete the pod manually.

Self-healing will happen after three failures of the probe.

Worker node drained

The pod will be evicted from the worker node and removed from the service. A new pod will be created on a different worker node from a physical backup of the *primary* if the `reusePVC` option of the `nodeMaintenanceWindow` parameter is set to `off` (default: `on` during maintenance windows, `off` otherwise).

The `PodDisruptionBudget` may prevent the pod from being evicted if there is at least another pod that is not ready.

Note: Single instance clusters prevent node drain when *reusePVC* is set to *false* . Refer to the *Kubernetes Upgrade section* .

Self-healing will happen as soon as the *apiserver* is notified.

Worker node failure

Since the node is failed, the *kubelet* won' t execute the liveness and the readiness probes. The pod will be marked for deletion after the toleration seconds configured by the Kubernetes cluster administrator for that specific failure cause. Based on how the Kubernetes cluster is configured, the pod might be removed from the service earlier.

A new pod will be created on a different worker node from a physical backup of the *primary*. The default value for that parameter in a Kubernetes cluster is 5 minutes.

Self-healing will happen after `tolerationSeconds` .

Self-healing

If the failed pod is a standby, the pod is removed from the `-r` service and from the `-ro` service. The pod is then restarted using its PVC if available; otherwise, a new pod will be created from a backup of the current primary. The pod will be added again to the `-r` service and to the `-ro` service when ready.

If the failed pod is the primary, the operator will promote the active pod with status ready and the lowest replication lag, then point the `-rw` service to it. The failed pod will be removed from the `-r` service and from the `-rw` service. Other standbys will start replicating from the new primary. The former primary will use `pg_rewind` to synchronize itself with the new one if its PVC is available; otherwise, a new standby will be created from a backup of the current primary.

Manual intervention

In the case of undocumented failure, it might be necessary to intervene to solve the problem manually.

Important: In such cases, please do not perform any manual operation without professional support .

From version 1.11.0 of the operator, you can use the `cnpg.io/reconciliationLoop` annotation to temporarily disable the reconciliation loop on a selected PostgreSQL cluster, as follows:

```
metadata:
  name: cluster-example-no-reconcile
  annotations:
    cnpg.io/reconciliationLoop: "disabled"
spec:
  # ...
```

The `cnpg.io/reconciliationLoop` must be used with extreme care and for the sole duration of the extraordinary/emergency operation.

Warning: Please make sure that you use this annotation only for a limited period of time and you remove it when the emergency has finished. Leaving this annotation in a cluster will prevent the operator from issuing any self-healing operation, such as a failover.

1.4.15 Rolling Updates

The operator allows changing the PostgreSQL version used in a cluster while applications are running against it.

Important: Only upgrades for PostgreSQL minor releases are supported.

Rolling upgrades are started when:

- the user changes the `imageName` attribute of the cluster specification;
- the *Image Catalog* is updated with a new image for the major used by the cluster;
- a change in the PostgreSQL configuration requires a restart to be applied;
- a change on the `Cluster .spec.resources` values
- a change in size of the persistent volume claim on AKS

- after the operator is updated, to ensure the Pods run the latest instance manager (unless *in-place updates are enabled*).

The operator starts upgrading all the replicas, one Pod at a time, and begins from the one with the highest serial.

The primary is the last node to be upgraded.

Rolling updates are configurable and can be either entirely automated (`unsupervised`) or requiring human intervention (`supervised`).

The upgrade keeps the CloudNativePG identity, without re-cloning the data. Pods will be deleted and created again with the same PVCs and a new image, if required.

During the rolling update procedure, each service endpoints move to reflect the cluster' s status, so that applications can ignore the node that is being updated.

Automated updates (`unsupervised`)

When `primaryUpdateStrategy` is set to `unsupervised` , the rolling update process is managed by Kubernetes and is entirely automated. Once the replicas have been upgraded, the selected `primaryUpdateMethod` operation will initiate on the primary. This is the default behavior.

The `primaryUpdateMethod` option accepts one of the following values:

- `restart` : if possible, perform an automated restart of the pod where the primary instance is running. Otherwise, the restart request is ignored and a switchover issued. This is the default behavior.
- `switchover` : a switchover operation is automatically performed, setting the most aligned replica as the new target primary, and shutting down the former primary pod.

There' s no one-size-fits-all configuration for the update method, as that depends on several factors like the actual workload of your database, the requirements in terms of RPO and RTO, whether your PostgreSQL architecture is shared or shared nothing, and so on.

Indeed, being PostgreSQL a primary/standby architecture database management system, the update process inevitably generates a downtime for your applications. One important aspect to consider for your context is the time it takes for your pod to download the new PostgreSQL container image, as that depends on your Kubernetes cluster settings and specifications. The `switchover` method makes sure that the promoted instance already runs the target image version of the container. The `restart` method instead might require to download the image from the origin registry after the primary pod has been shut down. It is up to you to determine whether, for your database, it is best to use `restart` or `switchover` as part of the rolling update procedure.

Manual updates (`supervised`)

When `primaryUpdateStrategy` is set to `supervised` , the rolling update process is suspended immediately after all replicas have been upgraded.

This phase can only be completed with either a manual switchover or an in-place restart. Keep in mind that image upgrades can not be applied with an in-place restart, so a switchover is required in such cases.

You can trigger a switchover with:

```
kubectl cnpg promote [cluster] [new_primary]
```

You can trigger a restart with:

```
kubectl cnpg restart [cluster] [current_primary]
```

You can find more information in the Backup {postgresql-cnpg-io-v1-Backup} .

1.4.16 Replication

Physical replication is one of the strengths of PostgreSQL and one of the reasons why some of the largest organizations in the world have chosen it for the management of their data in business continuity contexts. Primarily used to achieve high availability, physical replication also allows scale-out of read-only workloads and offloading of some work from the primary.

Important: This section is about replication within the same *Cluster* resource managed in the same Kubernetes cluster. For information about how to replicate with another Postgres *Cluster* resource, even across different Kubernetes clusters, please refer to the *Replica clusters*

---

section.

Application-level replication

Having contributed throughout the years to the replication feature in PostgreSQL, we have decided to build high availability in CloudNativePG on top of the native physical replication technology, and integrate it directly in the Kubernetes API.

In Kubernetes terms, this is referred to as **application-level replication**, in contrast with *storage-level replication*.

A very mature technology

PostgreSQL has a very robust and mature native framework for replicating data from the primary instance to one or more replicas, built around the concept of transactional changes continuously stored in the WAL (Write Ahead Log).

Started as the evolution of crash recovery and point in time recovery technologies, physical replication was first introduced in PostgreSQL 8.2 (2006) through WAL shipping from the primary to a warm standby in continuous recovery.

PostgreSQL 9.0 (2010) enhanced it with WAL streaming and read-only replicas via *hot standby*, while 9.1 (2011) introduced synchronous replication at the transaction level (for RPO=0 clusters). Cascading replication was released with PostgreSQL 9.2 (2012). The foundations of logical replication were laid in PostgreSQL 9.4, while version 10 (2017) introduced native support for the publisher/subscriber pattern to replicate data from an origin to a destination.

Streaming replication support

At the moment, CloudNativePG natively and transparently manages physical streaming replicas within a cluster in a declarative way, based on the number of provided `instances` in the `spec` :

```
replicas = instances - 1 (where  instances > 0)
```

Immediately after the initialization of a cluster, the operator creates a user called `streaming_replica` as follows:

```
CREATE USER streaming_replica WITH REPLICATION;
   -- NOSUPERUSER INHERIT NOCREATEROLE NOCREATEDB NOBYPASSRLS
```

Out of the box, the operator automatically sets up streaming replication within the cluster over an encrypted channel and enforces TLS client certificate authentication for the `streaming_replica` user - as highlighted by the following excerpt taken from `pg_hba.conf` :

```
#  Require client certificate authentication for the streaming_replica user

hostssl postgres streaming_replica all cert
hostssl replication streaming_replica all cert
```

See also:

For details on how CloudNativePG manages certificates, please refer to the *Certificates*

in the documentation.

If configured, the operator manages replication slots for all the replicas in the HA cluster, ensuring that WAL files required by each standby are retained on the primary' s storage, even after a failover or switchover.

See also:

For details on how CloudNativePG automatically manages replication slots for the High Availability replicas, please refer to the *Replication slots for High Availability*

below.

Continuous backup integration

In case continuous backup is configured in the cluster, CloudNativePG transparently configures replicas to take advantage of `restore_command` when in continuous recovery. As a result, PostgreSQL can use the WAL archive as a fallback option whenever pulling WALs via streaming replication fails.

Synchronous replication

CloudNativePG supports the configuration of **quorum-based synchronous streaming replication** via two configuration options called `minSyncReplicas` and `maxSyncReplicas` , which are the minimum and the maximum number of expected synchronous standby replicas available at any time. For self-healing purposes, the operator always compares these two values with the number of available replicas to determine the quorum.

Important: By default, synchronous replication selects among all the available replicas indistinctively. You can limit on which nodes your synchronous replicas can be scheduled, by working on node labels through the *syncReplicaElectionConstraint* option as described in the next section.

Synchronous replication is disabled by default (`minSyncReplicas` and `maxSyncReplicas` are not defined). In case both `minSyncReplicas` and `maxSyncReplicas` are set, CloudNativePG automatically updates the `synchronous_standby_names` option in PostgreSQL to the following value:

```
ANY q (pod1, pod2, ...)
```

Where:

- `q` is an integer automatically calculated by the operator to be: `1 <= minSyncReplicas <= q <= maxSyncReplicas <= readyReplicas`
- `pod1, pod2, ...` is the list of all PostgreSQL pods in the cluster

Warning: To provide self-healing capabilities, the operator can ignore *minSyncReplicas* if such value is higher than the currently available number of replicas. Synchronous replication is automatically disabled when *readyReplicas* is *0* .

As stated in the PostgreSQL documentation , the *method ``ANY`` specifies a quorum-based synchronous replication and makes transaction commits wait until their WAL records are replicated to at least the requested number of synchronous standbys in the list*.

Important: Even though the operator chooses self-healing over enforcement of synchronous replication settings, our recommendation is to plan for synchronous replication only in clusters with 3+ instances or, more generally, when *maxSyncReplicas < (instances - 1)* .

Select nodes for synchronous replication

CloudNativePG enables you to select which PostgreSQL instances are eligible to participate in a quorum-based synchronous replication set through anti-affinity rules based on the node labels where the PVC holding the PGDATA and the Postgres pod are.

See also:

For more information on the general pod affinity and anti-affinity rules, please check the *Scheduling* .

As an example use-case for this feature: in a cluster with a single sync replica, we would be able to ensure the sync replica will be in a different availability zone from the primary instance, usually identified by the `topology.kubernetes.io/zone` label on a node . This would increase the robustness of the cluster in case of an outage in a single availability zone, especially in terms of recovery point objective (RPO).

The idea of anti-affinity is to ensure that sync replicas that participate in the quorum are chosen from pods running on nodes that have different values for the selected labels (in this case, the availability zone label) then the node where the primary is currently in execution. If no node matches such criteria, the replicas are eligible for synchronous replication.

Important: The self-healing enforcement still applies while defining additional constraints for synchronous replica election (see *Zero-data-loss clusters through synchronous replication*).

The example below shows how this can be done through the `syncReplicaElectionConstraint` section within `.spec.postgresql` . `nodeLabelsAntiAffinity` allows you to specify those node labels that need to be evaluated to make sure that synchronous replication will be dynamically configured by the operator between the current primary and the replicas which are located on nodes having a value of the availability zone label different from that of the node where the primary is:

```
spec:
  instances: 3
  postgresql:
    syncReplicaElectionConstraint:
      enabled: true
      nodeLabelsAntiAffinity:
      - topology.kubernetes.io/zone
```

As you can imagine, the availability zone is just an example, but you could customize this behavior based on other labels that describe the node, such as storage, CPU, or memory.

Replication slots

Replication slots

are a native PostgreSQL feature introduced in 9.4 that provides an automated way to ensure that the primary does not remove WAL segments until all the attached streaming replication clients have received them, and that the primary does not remove rows which could cause a recovery conflict even when the standby is (temporarily) disconnected.

A replication slot exists solely on the instance that created it, and PostgreSQL does not replicate it on the standby servers. As a result, after a failover or a switchover, the new primary does not contain the replication slot from the old primary. This can create problems for the streaming replication clients that were connected to the old primary and have lost their slot.

CloudNativePG provides a turn-key solution to synchronize the content of physical replication slots from the primary to each standby, addressing two use cases:

- the replication slots automatically created for the High Availability of the Postgres cluster (see *Replication slots for High Availability* for details)
- *User-Defined Replication slots* created on the primary

Replication slots for High Availability

CloudNativePG fills this gap by introducing the concept of cluster-managed replication slots, starting with high availability clusters. This feature automatically manages physical replication slots for each hot standby replica in the High Availability cluster, both in the primary and the standby.

In CloudNativePG, we use the terms:

- **Primary HA slot**: a physical replication slot whose lifecycle is entirely managed by the current primary of the cluster and whose purpose is to map to a specific standby in streaming replication. Such a slot lives on the primary only.
- **Standby HA slot**: a physical replication slot for a standby whose lifecycle is entirely managed by another standby in the cluster, based on the content of the `pg_replication_slots` view in the primary, and updated at regular intervals using `pg_replication_slot_advance()` .

This feature is enabled by default and can be disabled via configuration. For details, please refer to the ReplicationSlotsConfiguration {postgresql-cnpg-io-v1-ReplicationSlotsConfiguration} . Here follows a brief description of the main options:

`.spec.replicationSlots.highAvailability.enabled` : if `true` , the feature is enabled (`true` is the default since 1.21)

`.spec.replicationSlots.highAvailability.slotPrefix` : the prefix that identifies replication slots managed by the operator for this feature (default: `_cnpg_`)

`.spec.replicationSlots.updateInterval` : how often the standby synchronizes the position of the local copy of the replication slots with the position on the current primary, expressed in seconds (default: 30)

Important: This capability requires PostgreSQL 11 or higher, as it relies on the pg_replication_slot_advance()

to directly manipulate the position of a replication slot.

> **Warning:** In PostgreSQL 11, enabling replication slots if initially disabled, or conversely disabling them if initially enabled, will require a rolling update of the cluster (due to the presence of the *recovery.conf* file that is only read at startup).

Although it is not recommended, if you desire a different behavior, you can customize the above options.

For example, the following manifest will create a cluster with replication slots disabled.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  # Disable replication slots for HA in the cluster
  replicationSlots:
    highAvailability:
      enabled: false

  storage:
    size: 1Gi
```

User-Defined Replication slots

Although CloudNativePG doesn't support a way to declaratively define physical replication slots, you can still create your own slots via SQL .

> **Note:** At the moment, we don't have any plans to manage replication slots in a declarative way, but it might change depending on the feedback we receive from users. The reason is that replication slots exist for a specific purpose and each should be managed by a specific application the oversees the entire lifecycle of the slot on the primary.

CloudNativePG can manage the synchronization of any user managed physical replication slots between the primary and standbys, similarly to what it does for the HA replication slots explained above (the only difference is that you need to create the replication slot).

This feature is enabled by default (meaning that any replication slot is synchronized), but you can disable it or further customize its behavior (for example by excluding some slots using regular expressions) through the `synchronizeReplicas` stanza. For example:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  replicationSlots:
    synchronizeReplicas:
      enabled: true
      excludePatterns:
      - "^foo"
```

For details, please refer to the ReplicationSlotsConfiguration {postgresql-cnpg-io-v1-ReplicationSlotsConfiguration} . Here follows a brief description of the main options:

`.spec.replicationSlots.synchronizeReplicas.enabled` : When true or not specified, every user-defined replication slot on the primary is synchronized on each standby. If changed to false, the operator will remove any replication slot previously created by itself on each standby.

`.spec.replicationSlots.synchronizeReplicas.excludePatterns` : A list of regular expression patterns to match the names of user-defined replication slots to be excluded from synchronization. This can be useful to exclude specific slots based on naming conventions.

> **Warning:** Users utilizing this feature should carefully monitor user-defined replication slots to ensure they align with their operational requirements and do not interfere with the failover process.

Synchronization frequency

You can also control the frequency with which a standby queries the `pg_replication_slots` view on the primary, and updates its local copy of the replication slots, like in this example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  # Reduce the frequency of standby HA slots updates to once every 5 minutes
  replicationSlots:
    updateInterval: 300

  storage:
    size: 1Gi
```

Capping the WAL size retained for replication slots

When replication slots is enabled, you might end up running out of disk space due to PostgreSQL trying to retain WAL files requested by a replication slot. This might happen due to a standby that is (temporarily?) down, or lagging, or simply an orphan replication slot.

Starting with PostgreSQL 13, you can take advantage of the
max_slot_wal_keep_size

configuration option controlling the maximum size of WAL files that replication slots are allowed to retain in the `pg_wal` directory at checkpoint time. By default, in PostgreSQL `max_slot_wal_keep_size` is set to `-1` , meaning that replication slots may retain an unlimited amount of WAL files. As a result, our recommendation is to explicitly set `max_slot_wal_keep_size` when replication slots support is enabled. For example:

```
# ...
postgresql:
  parameters:
    max_slot_wal_keep_size: "10GB"
# ...
```

Monitoring replication slots

Replication slots must be carefully monitored in your infrastructure. By default, we provide the `pg_replication_slots` metric in our Prometheus exporter with key information such as the name of the slot, the type, whether it is active, the lag from the primary.

See also:

Please refer to the MonitoringConfiguration {postgresql-cnpg-io-v1-MonitoringConfiguration} for details on how to monitor a CloudNativePG deployment.

1.4.17 Backup

Important: With version 1.21, backup and recovery capabilities in CloudNativePG have sensibly changed due to the introduction of native support for *Kubernetes Volume Snapshots* . Up to that point, backup and recovery were available only for object stores. Please carefully read this section and the Bootstrap from a backup (`recovery ) <Bootstrap from a backup (*recovery* )>`

---

one if you have been a user of CloudNativePG 1.15 through 1.20.

PostgreSQL natively provides first class backup and recovery capabilities based on file system level (physical) copy. These have been successfully used for more than 15 years in mission critical production databases, helping organizations all over the world achieve their disaster recovery goals with Postgres.

Note: There's another way to backup databases in PostgreSQL, through the *pg_dump* utility - which relies on logical backups instead of physical ones. However, logical backups are not suitable for business continuity use cases and as such are not covered by CloudNativePG (yet, at least). If you want to use the *pg_dump* utility, let yourself be inspired by the *Troubleshooting / Emergency backup* .

In CloudNativePG, the backup infrastructure for each PostgreSQL cluster is made up of the following resources:

- **WAL archive**: a location containing the WAL files (transactional logs) that are continuously written by Postgres and archived for data durability
- **Physical base backups**: a copy of all the files that PostgreSQL uses to store the data in the database (primarily the `PGDATA` and any tablespace)

The WAL archive can only be stored on object stores at the moment.

On the other hand, CloudNativePG supports two ways to store physical base backups:

- on *Appendix A - Common object stores for backups* , as tarballs - optionally compressed
- on *Kubernetes Volume Snapshots* , if supported by the underlying storage class

---

Important: Before choosing your backup strategy with CloudNativePG, it is important that you take some time to familiarize with some basic concepts, like WAL archive, hot and cold backups.

Important: Please refer to the official Kubernetes documentation for a list of all the supported Container Storage Interface (CSI) drivers

---

that provide snapshotting capabilities.

WAL archive

The WAL archive in PostgreSQL is at the heart of **continuous backup**, and it is fundamental for the following reasons:

- **Hot backups**: the possibility to take physical base backups from any instance in the Postgres cluster (either primary or standby) without shutting down the server; they are also known as online backups
- **Point in Time recovery** (PITR): to possibility to recover at any point in time from the first available base backup in your system

Warning: WAL archive alone is useless. Without a physical base backup, you cannot restore a PostgreSQL cluster.

In general, the presence of a WAL archive enhances the resilience of a PostgreSQL cluster, allowing each instance to fetch any required WAL file from the archive if needed (normally the WAL archive has higher retention periods than any Postgres instance that normally recycles those files).

This use case can also be extended to *Replica clusters* , as they can simply rely on the WAL archive to synchronize across long distances, extending disaster recovery goals across different regions.

When you *configure a WAL archive* , CloudNativePG provides out-of-the-box an RPO <= 5 minutes for disaster recovery, even across regions.

Important: Our recommendation is to always setup the WAL archive in production. There are known use cases - normally involving staging and development environments - where none of the above benefits are needed and the WAL archive is not necessary. RPO in this case can be any value, such as 24 hours (daily backups) or infinite (no backup at all).

Cold and Hot backups

Hot backups have already been defined in the previous section. They require the presence of a WAL archive and they are the norm in any modern database management system.

Cold backups, also known as offline backups, are instead physical base backups taken when the PostgreSQL instance (standby or primary) is shut down. They are consistent per definition and they represent a snapshot of the database at the time it was shut down.

As a result, PostgreSQL instances can be restarted from a cold backup without the need of a WAL archive, even though they can take advantage of it, if available (with all the benefits on the recovery side highlighted in the previous section).

In those situations with a higher RPO (for example, 1 hour or 24 hours), and shorter retention periods, cold backups represent a viable option to be considered for your disaster recovery plans.

Object stores or volume snapshots: which one to use?

In CloudNativePG, object store based backups:

- always require the WAL archive
- support hot backup only
- don’t support incremental copy
- don’t support differential copy

VolumeSnapshots instead:

- don' t require the WAL archive, although in production it is always recommended
- support incremental copy, depending on the underlying storage classes
- support differential copy, depending on the underlying storage classes
- also support cold backup

Which one to use depends on your specific requirements and environment, including:

- availability of a viable object store solution in your Kubernetes cluster
- availability of a trusted storage class that supports volume snapshots
- size of the database: with object stores, the larger your database, the longer backup and, most importantly, recovery procedures take (the latter impacts RTO); in presence of Very Large Databases (VLDB), the general advice is to rely on Volume Snapshots as, thanks to copy-on-write, they provide faster recovery
- data mobility and possibility to store or relay backup files on a secondary location in a different region, or any subsequent one
- other factors, mostly based on the confidence and familiarity with the underlying storage solutions

The summary table below highlights some of the main differences between the two available methods for storing physical base backups.

	Object store	Volume Snapshots
WAL archiving	Required	Recommended (1)
Cold backup	☒	✓
Hot backup	✓	✓
Incremental copy	☒	✓ (2)
Differential copy	☒	✓ (2)
Backup from a standby	✓	✓
Snapshot recovery	☒ (3)	✓
Point In Time Recovery (PITR)	✓	Requires WAL archive
Underlying technology	Barman Cloud	Kubernetes API
See the explanation below for the notes in the above table: > > 1.		
WAL archive must be on an object store at the moment > 2. If		
supported by the underlying storage classes of the PostgreSQL volumes		
> 3. Snapshot recovery can be emulated using the >		
`bootstrap.recovery.recoveryTarget.targetImmediate` option		

Scheduled backups

Scheduled backups are the recommended way to configure your backup strategy in CloudNativePG. They are managed by the `ScheduledBackup` resource.

Note: Please refer to ScheduledBackupSpec {postgresql-cnpg-io-v1-ScheduledBackupSpec}

in the API reference for a full list of options.

The `schedule` field allows you to define a *six-term cron schedule* specification, which includes seconds, as expressed in the *Go `cron* package format <https://pkg.go.dev/github.com/robfig/cron#hdr-CRON_Expression_Format>`_ .

> **Warning:** Beware that this format accepts also the *seconds* field, and it is different from the *crontab* format in Unix/Linux systems.

This is an example of a scheduled backup:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: ScheduledBackup
metadata:
  name: backup-example
spec:
  schedule: "0 0 0 * * *"
  backupOwnerReference: self
  cluster:
    name: pg-backup
```

The above example will schedule a backup every day at midnight because the schedule specifies zero for the second, minute, and hour, while specifying wildcard, meaning all, for day of the month, month, and day of the week.

In Kubernetes CronJobs, the equivalent expression is `0 0 * * *` because seconds are not included.

Hint: Backup frequency might impact your recovery time object (RTO) after a disaster which requires a full or Point-In-Time recovery operation. Our advice is that you regularly test your backups by recovering them, and then measuring the time it takes to recover from scratch so that you can refine your RTO predictability. Recovery time is influenced by the size of the base backup and the amount of WAL files that need to be fetched from the archive and replayed during recovery (remember that WAL archiving is what enables continuous backup in PostgreSQL!). Based on our experience, a weekly base backup is more than enough for most cases - while it is extremely rare to schedule backups more frequently than once a day.

You can choose whether to schedule a backup on a defined object store or a volume snapshot via the `.spec.method` attribute, by default set to `barmanObjectStore` . If you have properly defined *Object stores or volume snapshots: which one to use?*

in the `backup` stanza of the cluster, you can set `method: volumeSnapshot` to start scheduling base backups on volume snapshots.

ScheduledBackups can be suspended, if needed, by setting `.spec.suspend: true` . This will stop any new backup from being scheduled until the option is removed or set back to `false` .

In case you want to issue a backup as soon as the ScheduledBackup resource is created you can set `.spec.immediate: true` .

Note: *.spec.backupOwnerReference* indicates which ownerReference should be put inside the created backup resources.

- *none:* no owner reference for created backup objects (same behavior as before the field was introduced) - *self:* sets the Scheduled backup object as owner of the backup - *cluster:* set the cluster as owner of the backup

On-demand backups

Note: Please refer to BackupSpec {postgresql-cnpg-io-v1-BackupSpec}

in the API reference for a full list of options.

To request a new backup, you need to create a new `Backup` resource like the following one:

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: backup-example
spec:
  method: barmanObjectStore
  cluster:
    name: pg-backup
```

In this case, the operator will start to orchestrate the cluster to take the required backup on an object store, using `barman-cloud-backup` . You can check the backup status using the plain `kubectl describe backup <name>` command:

```
Name:         backup-example
Namespace:    default
Labels:       <none>
Annotations:  API Version:  postgresql.cnpg.io/v1
Kind:         Backup
Metadata:
  Creation Timestamp:  2020-10-26T13:57:40Z
  Self Link:         /apis/postgresql.cnpg.io/v1/namespaces/default/backups/backup-
↪example
  UID:               ad5f855c-2ffd-454a-a157-900d5f1f6584
Spec:
  Cluster:
    Name:  pg-backup
Status:
  Phase:       running
  Started At:  2020-10-26T13:57:40Z
Events:        <none>
```

When the backup has been completed, the phase will be `completed` like in the following example:

```
Name:         backup-example
Namespace:    default
Labels:       <none>
Annotations:  API Version:  postgresql.cnpg.io/v1
Kind:         Backup
Metadata:
  Creation Timestamp:  2020-10-26T13:57:40Z
  Self Link:         /apis/postgresql.cnpg.io/v1/namespaces/default/backups/backup-
↪example
  UID:               ad5f855c-2ffd-454a-a157-900d5f1f6584
Spec:
```

(continues on next page)

(continued from previous page)

```
  Cluster:
    Name:  pg-backup
Status:
  Backup Id:         20201026T135740
  Destination Path:  s3://backups/
  Endpoint URL:      http://minio:9000
  Phase:             completed
  s3Credentials:
    Access Key Id:
      Key:   ACCESS_KEY_ID
      Name:  minio
    Secret Access Key:
      Key:      ACCESS_SECRET_KEY
      Name:     minio
  Server Name:  pg-backup
  Started At:   2020-10-26T13:57:40Z
  Stopped At:   2020-10-26T13:57:44Z
Events:         <none>
```

!!!Important This feature will not backup the secrets for the superuser and the application user. The secrets are supposed to be backed up as part of the standard backup procedures for the Kubernetes cluster.

Backup from a standby

Taking a base backup requires to scrape the whole data content of the PostgreSQL instance on disk, possibly resulting in I/O contention with the actual workload of the database.

For this reason, CloudNativePG allows you to take advantage of a feature which is directly available in PostgreSQL: **backup from a standby**.

By default, backups will run on the most aligned replica of a `Cluster` . If no replicas are available, backups will run on the primary instance.

Note: Although the standby might not always be up to date with the primary, in the time continuum from the first available backup to the last archived WAL this is normally irrelevant. The base backup indeed represents the starting point from which to begin a recovery operation, including PITR. Similarly to what happens with pg_basebackup , when backing up from an online standby we do not force a switch of the WAL on the primary. This might produce unexpected results in the short term (before *archive_timeout* kicks in) in deployments with low write activity.

If you prefer to always run backups on the primary, you can set the backup target to `primary` as outlined in the example below:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  [...]
spec:
  backup:
    target: "primary"
```

> **Warning:** Beware of setting the target to primary when performing a cold backup with volume snapshots, as this will shut down the primary for the time needed to take the snapshot, impacting write operations. This also applies to taking a cold backup in a single-instance cluster, even if you did not explicitly set the primary as the target.

When the backup target is set to `prefer-standby` , such policy will ensure backups are run on the most up-to-date available secondary instance, or if no other instance is available, on the primary instance.

By default, when not otherwise specified, target is automatically set to take backups from a standby.

The backup target specified in the `Cluster` can be overridden in the `Backup` and `ScheduledBackup` types, like in the following example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  [...]
spec:
  cluster:
    name: [...]
  target: "primary"
```

In the previous example, CloudNativePG will invariably choose the primary instance even if the `Cluster` is set to prefer replicas.

1.4.18 Backup on object stores

CloudNativePG natively supports **online/hot backup** of PostgreSQL clusters through continuous physical backup and WAL archiving on an object store. This means that the database is always up (no downtime required) and that Point In Time Recovery is available.

The operator can orchestrate a continuous backup infrastructure that is based on the Barman Cloud tool. Instead of using the classical architecture with a Barman server, which backs up many PostgreSQL instances, the operator relies on the `barman-cloud-wal-archive` , `barman-cloud-check-wal-archive` , `barman-cloud-backup` , `barman-cloud-backup-list` , and `barman-cloud-backup-delete` tools. As a result, base backups will be *tarballs*. Both base backups and WAL files can be compressed and encrypted.

For this, it is required to use an image with `barman-cli-cloud` included. You can use the image `ghcr.io/cloudnative-pg/postgresql` for this scope, as it is composed of a community PostgreSQL image and the latest `barman-cli-cloud` package.

> **Important:** Always ensure that you are running the latest version of the operands in your system to take advantage of the improvements introduced in Barman cloud (as well as improve the security aspects of your cluster).

A backup is performed from a primary or a designated primary instance in a `Cluster` (please refer to *Replica clusters* for more information about designated primary instances), or alternatively on a *Backup from a standby* .

Common object stores

If you are looking for a specific object store such as *AWS S3* **,**
Microsoft Azure Blob Storage , *Google Cloud Storage* , or *MinIO Gateway* , or a compatible

provider, please refer to *Appendix A - Common object stores for backups* .

Retention policies

Important: Retention policies are not currently available on volume snapshots.

CloudNativePG can manage the automated deletion of backup files from the backup object store, using **retention policies** based on the recovery window.

Internally, the retention policy feature uses `barman-cloud-backup-delete` with `--retention-policy "RECOVERY WINDOW OF {{ retention policy value }} {{ retention policy unit }}"` .

For example, you can define your backups with a retention policy of 30 days as follows:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path here>"
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
    retentionPolicy: "30d"
```

Compression algorithms

CloudNativePG by default archives backups and WAL files in an uncompressed fashion. However, it also supports the following compression algorithms via `barman-cloud-backup` (for backups) and `barman-cloud-wal-archive` (for WAL files):

- bzip2
- gzip
- snappy

The compression settings for backups and WALs are independent. See the
DataBackupConfiguration {postgresql-cnpg-io-v1-DataBackupConfiguration} and WalBackupConfiguration {postgresql-cnpg-io-v1-WalBackupConfiguration} sections in the API reference.

It is important to note that archival time, restore time, and size change between the algorithms, so the compression algorithm should be chosen according to your use case.

The Barman team has performed an evaluation of the performance of the supported algorithms for Barman Cloud. The following table summarizes a scenario where a backup is taken on a local MinIO deployment. The Barman GitHub project includes a deeper analysis .

Compression	Backup Time (ms)	Restore Time (ms)	Uncompressed size (MB)	Compressed size (MB)	Approx ratio
None	10927	7553	395	395	1:1
bzip2	25404	13886	395	67	5.9:1
gzip	116281	3077	395	91	4.3:1
snappy	8134	8341	395	166	2.4:1

Tagging of backup objects

Barman 2.18 introduces support for tagging backup resources when saving them in object stores via `barman-cloud-backup` and `barman-cloud-wal-archive` . As a result, if your PostgreSQL container image includes Barman with version 2.18 or higher, CloudNativePG enables you to specify tags as key-value pairs for backup objects, namely base backups, WAL files and history files.

You can use two properties in the `.spec.backup.barmanObjectStore` definition:

- `tags` : key-value pair tags to be added to backup objects and archived WAL file in the backup object store
- `historyTags` : key-value pair tags to be added to archived history files in the backup object store

The excerpt of a YAML manifest below provides an example of usage of this feature:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      tags:
        backupRetentionPolicy: "expire"
      historyTags:
        backupRetentionPolicy: "keep"
```

Extra options for the backup command

You can append additional options to the `barman-cloud-backup` command by using the `additionalCommandArgs` property in the `.spec.backup.barmanObjectStore.data` section. This property is a list of strings that will be appended to the `barman-cloud-backup` command. For example, you can use the `--read-timeout=60` to customize the connection reading timeout. For additional options supported by `barman-cloud-backup` you can refer to the official barman documentation here .

If an option provided in `additionalCommandArgs` is already present in the declared options in the `barmanObjectStore` section, the extra option will be ignored.

The following is an example of how to use this property:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      data:
        additionalCommandArgs:
        - "--min-chunk-size=5MB"
        - "--read-timeout=60"
```

1.4.19 WAL archiving

WAL archiving is the process that feeds a *WAL archive*

in CloudNativePG.

Important: CloudNativePG currently only supports WAL archives on object stores. Such WAL archives serve for both object store backups and volume snapshot backups.

The WAL archive is defined in the `.spec.backup.barmanObjectStore` stanza of a `Cluster` resource. Please proceed with the same instructions you find in the *Backup on object stores* to set up the WAL archive.

Note: Please refer to BarmanObjectStoreConfiguration {postgresql-cnpg-io-v1-BarmanObjectStoreConfiguration}

in the API reference for a full list of options.

If required, you can choose to compress WAL files as soon as they are uploaded and/or encrypt them:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      wal:
        compression: gzip
        encryption: AES256
```

You can configure the encryption directly in your bucket, and the operator will use it unless you override it in the cluster configuration.

PostgreSQL implements a sequential archiving scheme, where the `archive_command` will be executed sequentially for every WAL segment to be archived.

Important: By default, CloudNativePG sets *archive_timeout* to *5min* , ensuring that WAL files, even in case of low workloads, are closed and archived at least every 5 minutes, providing a deterministic time-based value for your

Recovery Point Objective (RPO). Even though you change the value of the archive_timeout , our experience suggests that the default value set by the operator is suitable for most use cases.

When the bandwidth between the PostgreSQL instance and the object store allows archiving more than one WAL file in parallel, you can use the parallel WAL archiving feature of the instance manager like in the following example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      wal:
        compression: gzip
        maxParallel: 8
        encryption: AES256
```

In the previous example, the instance manager optimizes the WAL archiving process by archiving in parallel at most eight ready WALs, including the one requested by PostgreSQL.

When PostgreSQL will request the archiving of a WAL that has already been archived by the instance manager as an optimization, that archival request will be just dismissed with a positive status.

1.4.20 Backup on volume snapshots

Warning: The initial release of volume snapshots (version 1.21.0) only supported cold backups, which required fencing of the instance. This limitation has been waived starting with version 1.21.1. Given the minimal impact of the change on the code, maintainers have decided to backport this feature immediately instead of waiting for version 1.22.0 to be out, and make online backups the default behavior on volume snapshots too. If you are planning to rely instead on cold backups, make sure you follow the instructions below.

Warning: As noted in the *backup document* , a cold snapshot explicitly set to target the primary will result in the primary being fenced for the duration of the backup, rendering the cluster read-only during that For safety, in a cluster already containing fenced instances, a cold snapshot is rejected.

CloudNativePG is one of the first known cases of database operators that directly leverages the Kubernetes native Volume Snapshot API for both backup and recovery operations, in an entirely declarative way.

About standard Volume Snapshots

Volume snapshotting was first introduced in Kubernetes 1.12 (2018) as alpha , promoted to beta in 1.17 (2019) , and moved to GA in 1.20 (2020) . It' s now stable, widely available,

and standard, providing 3 custom resource definitions: `VolumeSnapshot` , `VolumeSnapshotContent` and `VolumeSnapshotClass` .

This Kubernetes feature defines a generic interface for:

- the creation of a new volume snapshot, starting from a PVC
- the deletion of an existing snapshot
- the creation of a new volume from a snapshot

Kubernetes delegates the actual implementation to the underlying CSI drivers (not all of them support volume snapshots). Normally, storage classes that provide volume snapshotting support **incremental and differential block level backup in a transparent way for the application**, which can delegate the complexity and the independent management down the stack, including cross-cluster availability of the snapshots.

Requirements

For Volume Snapshots to work with a CloudNativePG cluster, you need to ensure that each storage class used to dynamically provision the PostgreSQL volumes (namely, `storage` and `walStorage` sections) support volume snapshots.

Given that instructions vary from storage class to storage class, please refer to the documentation of the specific storage class and related CSI drivers you have deployed in your Kubernetes system.

Normally, it is the *VolumeSnapshotClass*

that is responsible to ensure that snapshots can be taken from persistent volumes of a given storage class, and managed as `VolumeSnapshot` and `VolumeSnapshotContent` resources.

Important: It is your responsibility to verify with the third party vendor that volume snapshots are supported. CloudNativePG only interacts with the Kubernetes API on this matter and we cannot support issues at the storage level for each specific CSI driver.

How to configure Volume Snapshot backups

CloudNativePG allows you to configure a given Postgres cluster for Volume Snapshot backups through the `backup.volumeSnapshot` stanza.

Note: Please refer to VolumeSnapshotConfiguration {postgresql-cnpg-io-v1-VolumeSnapshotConfiguration}

---

in the API reference for a full list of options.

A generic example with volume snapshots (assuming that PGDATA and WALs share the same storage class) is the following:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: snapshot-cluster
```

(continues on next page)

(continued from previous page)

```
spec:
  instances: 3

  storage:
    storageClass: @STORAGE_CLASS@
    size: 10Gi
  walStorage:
    storageClass: @STORAGE_CLASS@
    size: 10Gi

  backup:
    # Volume snapshot backups
    volumeSnapshot:
       className: @VOLUME_SNAPSHOT_CLASS_NAME@
    # WAL archive
    barmanObjectStore:
       # ...
```

As you can see, the `backup` section contains both the `volumeSnapshot` stanza (controlling physical base backups on volume snapshots) and the `barmanObjectStore` one (controlling the *WAL archive*).

Note: Once you have defined the *barmanObjectStore* , you can decide to use both volume snapshot and object store backup strategies simultaneously to take physical backups.

The `volumeSnapshot.className` option allows you to reference the default `VolumeSnapshotClass` object used for all the storage volumes you have defined in your PostgreSQL cluster.

Note: In case you are using a different storage class for *PGDATA* and WAL files, you can specify a separate *VolumeSnapshotClass* for that volume through the *walClassName* option (which defaults to the same value as *className*).

Once a cluster is defined for volume snapshot backups, you need to define a `ScheduledBackup` resource that requests such backups on a periodic basis.

Hot and cold backups

By default, CloudNativePG requests an online/hot backup on volume snapshots, using the PostgreSQL defaults of the low-level API for base backups :

- it doesn’t request an immediate checkpoint when starting the backup procedure
- it waits for the WAL archiver to archive the last segment of the backup when terminating the backup procedure

Important: The default values are suitable for most production environments. Hot backups are consistent and can be used to perform snapshot recovery, as we ensure WAL retention from the start of the backup through a temporary replication slot. However, our recommendation is to rely on cold backups for that purpose.

You can explicitly change the default behavior through the following options in the `.spec.backup.volumeSnapshot` stanza of the `Cluster` resource:

- `online` : accepting `true` (default) or `false` as a value
- `onlineConfiguration.immediateCheckpoint` : whether you want to request an immediate checkpoint before you start the backup procedure or not; technically, it corresponds to the `fast` argument you pass to the `pg_backup_start` /`pg_start_backup()` function in PostgreSQL, accepting `true` (default) or `false`
- `onlineConfiguration.waitForArchive` : whether you want to wait for the archiver to process the last segment of the backup or not; technically, it corresponds to the `wait_for_archive` argument you pass to the `pg_backup_stop` /`pg_stop_backup()` function in PostgreSQL, accepting `true` (default) or `false`

If you want to change the default behavior of your Postgres cluster to take cold backups by default, all you need to do is add the `online: false` option to your manifest, as follows:

```
# ...
backup:
  volumeSnapshot:
     online: false
     # ...
```

If you are instead requesting an immediate checkpoint as the default behavior, you can add this section:

```
# ...
backup:
  volumeSnapshot:
     online: true
     onlineConfiguration:
       immediateCheckpoint: true
     # ...
```

Overriding the default behavior

You can change the default behavior defined in the cluster resource by setting different values for `online` and, if needed, `onlineConfiguration` in the `Backup` or `ScheduledBackup` objects.

For example, in case you want to issue an on-demand cold backup, you can create a `Backup` object with `.spec.online: false` :

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: snapshot-cluster-cold-backup-example
spec:
  cluster:
    name: snapshot-cluster
  method: volumeSnapshot
  online: false
```

Similarly, for the ScheduledBackup:

```
apiVersion: postgresql.cnpg.io/v1
kind: ScheduledBackup
metadata:
  name: snapshot-cluster-cold-backup-example
spec:
```

(continues on next page)

(continued from previous page)

```
  schedule: "0 0 0 * * *"
  backupOwnerReference: self
  cluster:
    name: snapshot-cluster
  method: volumeSnapshot
  online: false
```

Persistence of volume snapshot objects

By default, `VolumeSnapshot` objects created by CloudNativePG are retained after deleting the `Backup` object that originated them, or the `Cluster` they refer to. Such behavior is controlled by the `.spec.backup.volumeSnapshot.snapshotOwnerReference` option which accepts the following values:

- `none` : no ownership is set, meaning that `VolumeSnapshot` objects persist after the `Backup` and/or the `Cluster` resources are removed
- `backup` : the `VolumeSnapshot` object is owned by the `Backup` resource that originated it, and when the backup object is removed, the volume snapshot is also removed
- `cluster` : the `VolumeSnapshot` object is owned by the `Cluster` resource that is backed up, and when the Postgres cluster is removed, the volume snapshot is also removed

In case a `VolumeSnapshot` is deleted, the `deletionPolicy` specified in the `VolumeSnapshotContent` is evaluated:

- if set to `Retain` , the `VolumeSnapshotContent` object is kept
- if set to `Delete` , the `VolumeSnapshotContent` object is removed as well

Warning: *VolumeSnapshotContent* objects do not keep all the information regarding the backup and the cluster they refer to (like the annotations and labels that are contained in the *VolumeSnapshot* object). Although possible, restoring from just this kind of object might not be straightforward. For this reason, our recommendation is to always backup the *VolumeSnapshot* definitions, even using a Kubernetes level data protection solution.

The value in `VolumeSnapshotContent` is determined by the `deletionPolicy` set in the corresponding `VolumeSnapshotClass` definition, which is referenced in the `.spec.backup.volumeSnapshot.className` option.

Please refer to the *Kubernetes documentation on Volume Snapshot Classes*

for details on this standard behavior.

Example

The following example shows how to configure volume snapshot base backups on an EKS cluster on AWS using the `ebs-sc` storage class and the `csi-aws-vsc` volume snapshot class.

Important: If you are interested in testing the example, please read Volume Snapshots for detailed instructions on the installation process for the storage class and the snapshot class.

The following manifest creates a `Cluster` that is ready to be used for volume snapshots and that stores the WAL archive in a S3 bucket via IAM role for the Service Account (IRSA, see *AWS S3*):

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: hendrix
spec:
  instances: 3

  storage:
    storageClass: ebs-sc
    size: 10Gi
  walStorage:
    storageClass: ebs-sc
    size: 10Gi

  backup:
    volumeSnapshot:
       className: csi-aws-vsc
    barmanObjectStore:
      destinationPath: s3://@BUCKET_NAME@/
      s3Credentials:
        inheritFromIAMRole: true
      wal:
        compression: gzip
        maxParallel: 2

  serviceAccountTemplate:
    metadata:
      annotations:
        eks.amazonaws.com/role-arn: "@ARN@"
- --
apiVersion: postgresql.cnpg.io/v1
kind: ScheduledBackup
metadata:
  name: hendrix-vs-backup
spec:
  cluster:
    name: hendrix
  method: volumeSnapshot
  schedule: 0 0 0 * * *
  backupOwnerReference: cluster
  immediate: true
```

The last resource defines daily volume snapshot backups at midnight, requesting one immediately after the cluster is created.

1.4.21 Recovery

In PostgreSQL terminology, recovery is the process of starting a PostgreSQL instance using an existing backup. The PostgreSQL recovery mechanism is very solid and rich. It also supports point-in-time recovery (PITR), which allows you to restore a given cluster up to any point in time, from the first available backup in your catalog to the last archived WAL. (The WAL archive is mandatory in this case.)

In CloudNativePG, you can' t perform recovery in place on an existing cluster. Recovery is instead a way to bootstrap a new Postgres cluster starting from an available physical backup.

Note: For details on the *bootstrap* stanza, see *Bootstrap* .

The `recovery` bootstrap mode lets you create a cluster from an existing physical base backup. You then reapply the WAL files containing the REDO log from the archive.

WAL files are pulled from the defined *recovery object store*.

Base backups can be taken either on object stores or using volume snapshots (from version 1.21).

Warning: Recovery using volume snapshots had an initial release on 1.20.1. Because of the amount of progress on the feature for 1.21.0, to use volume snapshots, we strongly advise you to upgrade to 1.21 or more advanced releases.

You can achieve recovery from a *recovery object store* in two ways:

- We recommend using a recovery object store, that is, a backup of another cluster created by Barman Cloud and defined by way of the `barmanObjectStore` option in the `externalClusters` section.
- Alternatively, you can use an existing `Backup` object in the same namespace.

Both recovery methods enable either full recovery (up to the last available WAL) or up to a *Point in time recovery (PITR)* . When performing a full recovery, you can also start the cluster in replica mode (see *Replica clusters* for reference).

Important: If using replica mode, make sure that the PostgreSQL configuration (*.spec.postgresql.parameters*) of the recovered cluster is compatible with the original one from a physical replication standpoint.

For recovery using *volume snapshots*:

- Use a consistent set of `VolumeSnapshot` objects that all belong to the same backup and are identified by the same `cnpg.io/cluster` and `cnpg.io/backupName` labels. Then, recover through the `volumeSnapshots` option in the `.spec.bootstrap.recovery` stanza, as described in Recovery from `VolumeSnapshot objects <Recovery from *VolumeSnapshot* objects>` .

Recovery from an object store

You can recover from a backup created by Barman Cloud and stored on a supported object store. After you define the external cluster, including all the required configuration in the `barmanObjectStore` section, you need to reference it in the `.spec.recovery.source` option.

This example defines a recovery object store in a blob container in Azure:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-restore
spec:
  [...]

  superuserSecret:
    name: superuser-secret

  bootstrap:
    recovery:
      source: clusterBackup

  externalClusters:
    - name: clusterBackup
      barmanObjectStore:
        destinationPath: https://STORAGEACCOUNTNAME.blob.core.windows.net/CONTAINERNAME/
        azureCredentials:
          storageAccount:
            name: recovery-object-store-secret
            key: storage_account_name
          storageKey:
            name: recovery-object-store-secret
            key: storage_account_key
        wal:
          maxParallel: 8
```

Important: By default, the *recovery* method strictly uses the *name* of the cluster in the *externalClusters* section as the name of the main folder of the backup data within the object store. This name is normally reserved for the name of the server. You can specify a different folder name using the *barmanObjectStore.serverName* property.

Note: This example takes advantage of the parallel WAL restore feature, dedicating up to 8 jobs to concurrently fetch the required WAL files from the archive. This feature can appreciably reduce the recovery time. Make sure that you plan ahead for this scenario and correctly tune the value of this parameter for your environment. It will make a difference when you need it, and you will.

Recovery from `VolumeSnapshot` objects

Warning: When creating replicas after recovering the primary instance from the volume snapshot, the operator might end up using *pg_basebackup* to synchronize them. This behavior results in a slower process, depending on the size of the database. This limitation will be lifted in the future when support for online backups and PVC cloning are introduced.

CloudNativePG can create a new cluster from a `VolumeSnapshot` of a PVC of an existing `Cluster` that' s been taken using the declarative API for volume snapshot backups. You must specify the name of the snapshot, as in the following example:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-restore
spec:
  [...]

  bootstrap:
    recovery:
      volumeSnapshots:
        storage:
          name: <snapshot name>
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io
```

In case the backed-up cluster was using a separate PVC to store the WAL files, the recovery must include that too:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-restore
spec:
  [...]

  bootstrap:
    recovery:
      volumeSnapshots:
        storage:
          name: <snapshot name>
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io

        walStorage:
          name: <snapshot name>
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io
```

Warning: If bootstrapping a replica-mode cluster from snapshots, to leverage snapshots for the standby instances and not just the primary, we recommend that you:

1. Start with a single instance replica cluster. The primary instance will be recovered using the snapshot, and available WALs from the source cluster. 2. Take a snapshot of the primary in the replica cluster. 3. Increase the number of instances in the replica cluster as desired.

Recovery from a `Backup` object

If a `Backup` resource is already available in the namespace in which you need to create the cluster, you can specify the name using `.spec.bootstrap.recovery.backup.name` , as in the following example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    recovery:
      backup:
        name: backup-example

  storage:
    size: 1Gi
```

This bootstrap method allows you to specify just a reference to the backup that needs to be restored.

The previous example implies the application database and its owning user is the default one, `app` . If the PostgreSQL cluster being restored was using different names, you can specify them as documented in *Configure the application database*.

Additional considerations

Whether you recover from a recovery object store, a volume snapshot, or an existing `Backup` resource, the following considerations apply:

- The application database name and the application database user are preserved from the backup that' s being restored. The operator doesn' t currently attempt to back up the underlying secrets, as this is part of the usual maintenance activity of the Kubernetes cluster.
- To preserve the original postgres user password, you need to properly configure `enableSuperuserAccess` and supply a `superuserSecret` .
- By default, the recovery continues up to the latest available WAL on the default target timeline (`latest`). You can optionally specify a `recoveryTarget` to perform a point-in-time recovery (see *Point in time recovery (PITR)*).

Important: Consider using the *barmanObjectStore.wal.maxParallel* option to speed up WAL fetching from the archive by concurrently downloading the transaction logs from the recovery object store.

Point in time recovery (PITR)

Instead of replaying all the WALs up to the latest one, after extracting a base backup, you can ask PostgreSQL to stop replaying WALs at any given point in time. PostgreSQL uses this technique to achieve PITR. The presence of a WAL archive is mandatory.

Important: PITR requires you to specify a recovery target by using the options described in *Recovery targets* .

The operator generates the configuration parameters required for this feature to work if you specify a recovery target.

PITR from an object store

This example uses a recovery object store in Azure that contains both the base backups and the WAL archive. The recovery target is based on a requested timestamp.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-restore-pitr
spec:
  instances: 3

  storage:
    size: 5Gi

  bootstrap:
    recovery:
      # Recovery object store containing WAL archive and base backups
      source: clusterBackup
      recoveryTarget:
        # Time base target for the recovery
        targetTime: "2023-08-11 11:14:21.00000+02"

  externalClusters:
    - name: clusterBackup
      barmanObjectStore:
        destinationPath: https://STORAGEACCOUNTNAME.blob.core.windows.net/CONTAINERNAME/
        azureCredentials:
          storageAccount:
            name: recovery-object-store-secret
            key: storage_account_name
          storageKey:
            name: recovery-object-store-secret
            key: storage_account_key
        wal:
          maxParallel: 8
```

In this example, you had to specify only the `targetTime` in the form of a timestamp. You didn't have to specify the base backup from which to start the recovery.

The `backupID` option is the one that allows you to specify the base backup from which to initiate the recovery process. By default, this value is empty.

If you assign a value to it (in the form of a Barman backup ID), the operator uses that backup as the base for the recovery.

Important: You need to make sure that such a backup exists and is accessible.

If you don't specify the backup ID, the operator detects the base backup for the recovery as follows:

- When you use `targetTime` or `targetLSN` , the operator selects the closest backup that was completed before that target.
- Otherwise, the operator selects the last available backup, in chronological order.

PITR from `VolumeSnapshot` objects

The example that follows uses:

- A Kubernetes volume snapshot for the `PGDATA` containing the base backup from which to start the recovery process. This snapshot is identified in the `recovery.volumeSnapshots` section and called `test-snapshot-1` .
- A recovery object store in MinIO containing the WAL archive. The object store is identified by the `recovery.source` option in the form of an external cluster definition.

The recovery target is based on a requested timestamp.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-snapshot
spec:
  # ...
  bootstrap:
    recovery:
      source: cluster-example-with-backup
      volumeSnapshots:
        storage:
          name: test-snapshot-1
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io
      recoveryTarget:
        targetTime: "2023-07-06T08:00:39"
  externalClusters:
    - name: cluster-example-with-backup
      barmanObjectStore:
        destinationPath: s3://backups/
        endpointURL: http://minio:9000
        s3Credentials:
          accessKeyId:
            name: minio
            key: ACCESS_KEY_ID
          secretAccessKey:
            name: minio
            key: ACCESS_SECRET_KEY
```

Note: If the backed-up cluster had *walStorage* enabled, you also must specify the volume snapshot containing the *PGWAL* directory, as mentioned in *Recovery from VolumeSnapshot objects* .

Warning: It's your responsibility to ensure that the end time of the base backup in the volume snapshot is before the recovery target timestamp.

Recovery targets

Here are the recovery target criteria you can use:

targetTime
[Time stamp up to which recovery proceeds, expressed in] RFC 3339 format. (The precise stopping point is also influenced by

the `exclusive` option.)

targetXID : Transaction ID up to which recovery proceeds. (The precise stopping point is also influenced by the `exclusive` option.) Keep in mind that while transaction IDs are assigned sequentially at transaction start, transactions can complete in a different numeric order. The transactions that are recovered are those that committed before (and optionally including) the specified one.

targetName : Named restore point (created with `pg_create_restore_point()`) to which recovery proceeds.

targetLSN : LSN of the write-ahead log location up to which recovery proceeds. (The precise stopping point is also influenced by the `exclusive` option.)

targetImmediate : Recovery ends as soon as a consistent state is reached, that is, as early as possible. When restoring from an online backup, this means the point where taking the backup ended.

Important: The operator can retrieve the closest backup when you specify either *targetTime* or *targetLSN* . However, this isn't possible for the remaining targets: *targetName* , *targetXID* , and *targetImmediate* . In such cases, it's important to specify *backupID* , unless the last available backup in the catalog is acceptable.

This example uses a `targetName` -based recovery target:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
  bootstrap:
    recovery:
      source: clusterBackup
      recoveryTarget:
        backupID: 20220616T142236
        targetName: restore_point_1
[...]
```

You can choose only a single one among the targets in each `recoveryTarget` configuration.

Additionally, you can specify `targetTLI` to force recovery to a specific timeline.

By default, the previous parameters are considered to be inclusive, stopping just after the recovery target, matching the behavior in PostgreSQL .

You can request exclusive behavior, stopping right before the recovery target, by setting the `exclusive` parameter to `true` . The following example shows this behavior, relying on a blob container in Azure for both base backups and the WAL archive:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-restore-pitr
spec:
  instances: 3

  storage:
    size: 5Gi

  bootstrap:
    recovery:
      source: clusterBackup
      recoveryTarget:
        backupID: 20220616T142236
        targetName: "maintenance-activity"
        exclusive: true

  externalClusters:
    - name: clusterBackup
      barmanObjectStore:
        destinationPath: https://STORAGEACCOUNTNAME.blob.core.windows.net/CONTAINERNAME/
        azureCredentials:
          storageAccount:
            name: recovery-object-store-secret
            key: storage_account_name
          storageKey:
            name: recovery-object-store-secret
            key: storage_account_key
        wal:
          maxParallel: 8
```

Configure the application database

For the recovered cluster, you can configure the application database name and credentials with additional configuration. To update application database credentials, you can generate your own passwords, store them as secrets, and update the database to use the secrets. Or you can also let the operator generate a secret with a randomly secure password for use. See Bootstrap an empty cluster (`initdb ) <Bootstrap an empty cluster (*initdb* )>`

for more information about secrets.

This example configures the application database `app` with owner `app` and supplied secret `app-secret` .

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  bootstrap:
    recovery:
```

(continues on next page)

(continued from previous page)

```
    database: app
    owner: app
    secret:
      name: app-secret
    [...]
```

With this configuration, the following happens after recovery is complete:

1. If database `app` doesn't exist, a new database `app` is created.
2. If user `app` doesn't exist, a new user `app` is created.
3. if user `app` isn't the owner of the database, user `app` is granted as owner of database `app` .
4. If the value of `username` matches the value of `owner` in the secret, the password of application database is changed to the value of `password` in the secret.

Important: For a replica cluster with replica mode enabled, the operator doesn't create any database or user in the PostgreSQL instance. These are recovered from the original cluster.

How recovery works under the hood

You can use the data uploaded to the object storage to *bootstrap* a new cluster from an existing backup. The operator orchestrates the recovery process using the `barman-cloud-restore` tool (for the base backup) and the `barman-cloud-wal-restore` tool (for WAL files, including parallel support, if requested).

For details and instructions on the `recovery` bootstrap method, see
Bootstrap from a backup (`recovery ) <Bootstrap from a backup (*recovery* )>` .

Important: If you're not familiar with how PostgreSQL PITR

works, we suggest that you configure the recovery cluster as the original one when it comes to `.spec.postgresql.parameters` . Once the new cluster is restored, you can then change the settings as desired.

The way it works is that the operator injects an init container in the first instance of the new cluster, and the init container starts recovering the backup from the object storage.

Important: The duration of the base backup copy in the new PVC depends on the size of the backup, as well as the speed of both the network and the storage.

When the base backup recovery process is complete, the operator starts the Postgres instance in recovery mode. In this phase, PostgreSQL is up, though not able to accept connections, and the pod is healthy according to the liveness probe. By way of the `restore_command` , PostgreSQL starts fetching WAL files from the archive. (You can speed up this phase by setting the `maxParallel` option and enabling the parallel WAL restore capability.)

This phase terminates when PostgreSQL reaches the target (either the end of the WAL or the required target in case of PITR. You can optionally specify a `recoveryTarget` to perform a PITR. If left unspecified, the recovery continues up to the latest available WAL on the default target timeline (`latest`).

Once the recovery is complete, the operator sets the required superuser password into the instance. The new primary instance starts as usual, and the remaining instances join the cluster as replicas.

The process is transparent for the user and is managed by the instance manager running in the pods.

Restoring into a cluster with a backup section

A manifest for a cluster restore might include a `backup` section. This means that,after recovery, the new cluster starts archiving WALs and taking backups if configured to do so.

For example, this section is part of a manifest for a cluster bootstrapping from the cluster `cluster-example-backup` . In the storage bucket, it creates a folder named `recoveredCluster` , where the base backups and WALs of the recovered cluster are stored.

```
backup:
  barmanObjectStore:
    destinationPath: s3://backups/
    endpointURL: http://minio:9000
    serverName: "recoveredCluster"
    s3Credentials:
      accessKeyId:
        name: minio
        key: ACCESS_KEY_ID
      secretAccessKey:
        name: minio
        key: ACCESS_SECRET_KEY
  retentionPolicy: "30d"

externalClusters:
- name: cluster-example-backup
  barmanObjectStore:
    destinationPath: s3://backups/
    endpointURL: http://minio:9000
    s3Credentials:
```

Don’t reuse the same `barmanObjectStore` configuration for different clusters. There might be cases where the existing information in the storage buckets could be overwritten by the new cluster.

> **Warning:** The operator includes a safety check to ensure a cluster doesn't overwrite a storage bucket that contained information. A cluster that would overwrite existing storage remains in the state *Setting up primary* with pods in an error state. The pod logs show: *ERROR: WAL archive check failed for server recoveredCluster: Expected empty archive* .

Important: If you set the *cnpg.io/skipEmptyWalArchiveCheck* annotation to *enabled* in the recovered cluster, you can skip the safety check. We don't recommend skipping the check because, for the general use case, the check works fine. Skip this check only if you're familiar with the PostgreSQL recovery system, as severe data loss can occur.

1.4.22 PostgreSQL Configuration

Users that are familiar with PostgreSQL are aware of the existence of the following three files to configure an instance:

- `postgresql.conf` : main run-time configuration file of PostgreSQL
- `pg_hba.conf` : clients authentication file
- `pg_ident.conf` : map external users to internal users

Due to the concepts of declarative configuration and immutability of the PostgreSQL containers, users are not allowed to directly touch those files. Configuration is possible through the `postgresql` section of the `Cluster` resource definition by defining custom `postgresql.conf` , `pg_hba.conf` , and `pg_ident.conf` settings via the `parameters` , the `pg_hba` , and the `pg_ident` keys.

These settings are the same across all instances.

> **Warning:** Please don't use the *ALTER SYSTEM* query to change the configuration of the PostgreSQL instances in an imperative way. Changing some of the options that are normally controlled by the operator might indeed lead to an unpredictable/unrecoverable state of the cluster. Moreover, *ALTER SYSTEM* changes are not replicated across the cluster. See Enabling ALTER SYSTEM below for details.

A reference for custom settings usage is included in the samples, see.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-custom
spec:
  instances: 3

  # Parameters and pg_hba configuration will be append
  # to the default ones to make the cluster work
  postgresql:
    parameters:
      max_worker_processes: "60"
    pg_hba:
      # To access through TCP/IP you will need to get username
      # and password from the secret cluster-example-custom-app
      - host all all all md5

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Require 1Gi of space per instance using default storage class
  storage:
    size: 1Gi
```

The postgresql section

The PostgreSQL instance in the pod starts with a default postgresql.conf file, to which these settings are automatically added:

```
listen_addresses = *
include custom.conf
```

The custom.conf file will contain the user-defined settings in the postgresql section, as in the following example:

```yaml
# ...
postgresql:
  parameters:
    shared_buffers: "1GB"
# ...
```

See also:

Refer to the PostgreSQL documentation for more information on the available parameters , also known as GUC (Grand Unified Configuration). Please note that CloudNativePG accepts only strings for the PostgreSQL parameters.

The content of custom.conf is automatically generated and maintained by the operator by applying the following sections in this order:

- Global default parameters
- Default parameters that depend on the PostgreSQL major version
- User-provided parameters
- Fixed parameters

The **global default parameters** are:

```
archive_mode = on
dynamic_shared_memory_type = posix
logging_collector = on
log_destination = csvlog
log_directory = /controller/log
log_filename = postgres
log_rotation_age = 0
log_rotation_size = 0
log_truncate_on_rotation = false
max_parallel_workers = 32
max_replication_slots = 32
max_worker_processes = 32
shared_memory_type = mmap # for PostgreSQL >= 12 only
wal_keep_size = 512MB # for PostgreSQL >= 13 only
wal_keep_segments = 32 # for PostgreSQL <= 12 only
wal_level = logical
wal_log_hints = on
wal_sender_timeout = 5s
wal_receiver_timeout = 5s
```

> **Warning:** It is your duty to plan for WAL segments retention in your PostgreSQL cluster and properly configure either *wal_keep_size* or *wal_keep_segments* , depending on the server version, based on the expected and observed workloads.

Alternatively, if the only streaming replication clients are the replica instances running in the High Availability cluster, you can take advantage of the replication slots feature, which adds support for replication slots at the cluster level. You can enable the feature with the `replicationSlots.highAvailability` option (for more information, please refer to the *About the replication user* .)

Without replication slots nor continuous backups in place, configuring `wal_keep_size` or `wal_keep_segments` is the only way to protect standbys from falling out of sync. If a standby did fall out of sync it would produce error messages like: `"could not receive data from WAL stream: ERROR: requested WAL segment ********** has already been removed"` . This will require you to dedicate a part of your `PGDATA` , or the volume dedicated to storing WAL files, to keep older WAL segments for streaming replication purposes.

The following parameters are **fixed** and exclusively controlled by the operator:

```
archive_command = /controller/manager wal-archive %p
full_page_writes = on
hot_standby = true
listen_addresses = *
port = 5432
restart_after_crash = false
ssl = on
ssl_ca_file = /controller/certificates/client-ca.crt
ssl_cert_file = /controller/certificates/server.crt
ssl_key_file = /controller/certificates/server.key
unix_socket_directories = /controller/run
```

Since the fixed parameters are added at the end, they can't be overridden by the user via the YAML configuration. Those parameters are required for correct WAL archiving and replication.

Replication settings

The `primary_conninfo` , `restore_command` , and `recovery_target_timeline` parameters are managed automatically by the operator according to the state of the instance in the cluster.

```
primary_conninfo = host=cluster-example-rw user=postgres dbname=postgres
recovery_target_timeline = latest
```

Log control settings

The operator requires PostgreSQL to output its log in CSV format, and the instance manager automatically parses it and outputs it in JSON format. For this reason, all log settings in PostgreSQL are fixed and cannot be changed.

For further information, please refer to the *Logging* .

Shared Preload Libraries

The `shared_preload_libraries` option in PostgreSQL exists to specify one or more shared libraries to be pre-loaded at server start, in the form of a comma-separated list. Typically, it is used in PostgreSQL to load those extensions that need to be available to most database sessions in the whole system (e.g. `pg_stat_statements`).

In CloudNativePG the `shared_preload_libraries` option is empty by default. Although you can override the content of `shared_preload_libraries` , we recommend that only expert Postgres users take advantage of this option.

Important: In case a specified library is not found, the server fails to start, preventing CloudNativePG from any self-healing attempt and requiring manual intervention. Please make sure you always test both the extensions and the settings of *shared_preload_libraries* if you plan to directly manage its content.

CloudNativePG is able to automatically manage the content of the `shared_preload_libraries` option for some of the most used PostgreSQL extensions (see the *Managed extensions* section below for details).

Specifically, as soon as the operator notices that a configuration parameter requires one of the managed libraries, it will automatically add the needed library. The operator will also remove the library as soon as no actual parameter requires it.

Important: Please always keep in mind that removing libraries from *shared_preload_libraries* requires a restart of all instances in the cluster in order to be effective.

You can provide additional `shared_preload_libraries` via `.spec.postgresql.shared_preload_libraries` as a list of strings: the operator will merge them with the ones that it automatically manages.

Managed extensions

As anticipated in the previous section, CloudNativePG automatically manages the content in `shared_preload_libraries` for some well-known and supported extensions. The current list includes:

- `auto_explain`
- `pg_stat_statements`
- `pgaudit`
- `pg_failover_slots`

Some of these libraries also require additional objects in a database before using them, normally views and/or functions managed via the `CREATE EXTENSION` command to be run in a database (the `DROP EXTENSION` command typically removes those objects).

For such libraries, CloudNativePG automatically handles the creation and removal of the extension in all databases that accept a connection in the cluster, identified by the following query:

```
SELECT datname FROM pg_database WHERE datallowconn
```

Note: The above query also includes template databases like *template1* .

Enabling `auto_explain`

The auto_explain

extension provides a means for logging execution plans of slow statements automatically, without having to manually run `EXPLAIN` (helpful for tracking down un-optimized queries).

You can enable `auto_explain` by adding to the configuration a parameter that starts with `auto_explain.` as in the following example excerpt (which automatically logs execution plans of queries that take longer than 10 seconds to complete):

```
# ...
postgresql:
  parameters:
    auto_explain.log_min_duration: "10s"
# ...
```

Note: Enabling auto_explain can lead to performance issues. Please refer to the auto explain documentation

Enabling `pg_stat_statements`

The pg_stat_statements

extension is one of the most important capabilities available in PostgreSQL for real-time monitoring of queries.

You can enable `pg_stat_statements` by adding to the configuration a parameter that starts with `pg_stat_statements.` as in the following example excerpt:

```
# ...
postgresql:
  parameters:
    pg_stat_statements.max: "10000"
    pg_stat_statements.track: all
# ...
```

As explained previously, the operator will automatically add `pg_stat_statements` to `shared_preload_libraries` and run `CREATE EXTENSION IF NOT EXISTS pg_stat_statements` on each database, enabling you to run queries against the `pg_stat_statements` view.

Enabling `pgaudit`

The `pgaudit` extension provides detailed session and/or object audit logging via the standard PostgreSQL logging facility.

CloudNativePG has transparent and native support for PGAudit on PostgreSQL clusters. For further information, please refer to the *PGAudit logs*

You can enable `pgaudit` by adding to the configuration a parameter that starts with `pgaudit.` as in the following example excerpt:

yaml # postgresql:

parameters: pgaudit.log: "all, -misc" pgaudit.log_catalog: "off" pgaudit.log_parameter: "on" pgaudit.log_relation: "on" #

Enabling pg_failover_slots

The pg_failover_slots

extension by EDB ensures that logical replication slots can survive a failover scenario. Failovers are normally implemented using physical streaming replication, like in the case of CloudNativePG.

You can enable `pg_failover_slots` by adding to the configuration a parameter that starts with `pg_failover_slots.` : as explained above, the operator will transparently manage the `pg_failover_slots` entry in the `shared_preload_libraries` option depending on this.

Please refer to the

for details on this extension.

Additionally, for each database that you intend to you use with `pg_failover_slots` you need to add an entry in the `pg_hba` section that enables each replica to connect to the primary. For example, suppose that you want to use the `app` database with `pg_failover_slots` , you need to add this entry in the `pg_hba` section:

```
postgresql:
  pg_hba:
    - hostssl app streaming_replica all cert
```

The pg_hba section

`pg_hba` is a list of PostgreSQL Host Based Authentication rules used to create the `pg_hba.conf` used by the pods.

Important: See the PostgreSQL documentation for *more information on `pg_hba.conf* <https://www.postgresql.org/docs/current/auth-pg-hba-conf.html>`_ .

Since the first matching rule is used for authentication, the `pg_hba.conf` file generated by the operator can be seen as composed of four sections:

1. Fixed rules
2. User-defined rules
3. Optional LDAP section
4. Default rules

Fixed rules:

```
local all all peer

hostssl postgres streaming_replica all cert
hostssl replication streaming_replica all cert
```

Default rules:

```
host all all all <default-authentication-method>
```

From PostgreSQL 14 the default value of the `password_encryption` database parameter is set to `scram-sha-256` . Because of that, the default authentication method is `scram-sha-256` from this PostgreSQL version.

PostgreSQL 13 and older will use `md5` as the default authentication method.

The resulting `pg_hba.conf` will look like this:

```
local all all peer

hostssl postgres streaming_replica all cert
hostssl replication streaming_replica all cert

<user defined rules>
<user defined LDAP>

host all all all scram-sha-256 # (or md5 for PostgreSQL version <= 13)
```

Inside the cluster manifest, `pg_hba` lines are added as list items in `.spec.postgresql.pg_hba` , as in the following excerpt:

```
postgresql:
  pg_hba:
    - hostssl app app 10.244.0.0/16 md5
```

In the above example we are enabling access for the `app` user to the `app` database using MD5 password authentication (you can use `scram-sha-256` if you prefer) via a secure channel (`hostssl`).

LDAP Configuration

Under the `postgres` section of the cluster spec there is an optional `ldap` section available to define an LDAP configuration to be converted into a rule added into the `pg_hba.conf` file.

This will support two modes: `simple bind` mode which requires specifying a `server` , `prefix` and `suffix` in the LDAP section and the `search+bind` mode which requires specifying `server` , `baseDN` , `binDN` , and a `bindPassword` which is a secret containing the ldap password. Additionally, in `search+bind` mode you have the option to specify a `searchFilter` or `searchAttribute` . If no `searchAttribute` is specified the default one of `uid` will be used.

Additionally, both modes allow the specification of a `scheme` for ldapscheme and a `port` . Neither scheme nor port are required, however.

This section filled out for search+bind could look as follows:

```
postgresql:
  ldap:
    server: openldap.default.svc.cluster.local
    bindSearchAuth:
      baseDN: ou=org,dc=example,dc=com
      bindDN: cn=admin,dc=example,dc=com
      bindPassword:
        name: ldapBindPassword
        key: data
      searchAttribute: uid
```

The pg_ident section

pg_ident is a list of PostgreSQL User Name Maps that CloudNativePG uses to generate and maintain the ident map file (known as pg_ident.conf) inside the data directory.

Important: See the PostgreSQL documentation for *more information on `pg_ident.conf* <https://www.postgresql.org/docs/current/auth-username-maps.html>`_ .

The pg_ident.conf file written by the operator is made up of the following two sections:

1. Fixed rules
2. User-defined rules

Currently the only fixed rule, automatically generated by the operator, is:

```
local <postgres system user> postgres
```

The instance manager detects the user running the PostgreSQL instance and automatically adds a rule to map it to the postgres user in the database.

If the postgres user is not properly configured inside the container, the instance manager will allow any local user to connect and then log a warning message like the following:

```
Unable to identify the current user. Falling back to insecure mapping.
```

The resulting pg_ident.conf will look like this:

```
local <postgres system user> postgres

<user defined lines>
```

Inside the cluster manifest, pg_ident lines are added as list items in .spec.postgresql.pg_ident . For example:

```
postgresql:
  pg_ident:
    - "mymap /^(.*)@mydomain\.com$ \1"
```

Changing configuration

You can apply configuration changes by editing the postgresql section of the Cluster resource.

After the change, the cluster instances will immediately reload the configuration to apply the changes. If the change involves a parameter requiring a restart, the operator will perform a rolling upgrade.

Enabling `ALTER SYSTEM`

CloudNativePG strongly advocates employing the Cluster manifest as the exclusive method for altering the configuration of a PostgreSQL cluster. This approach guarantees coherence across the entire high-availability cluster and aligns with best practices for Infrastructure-as-Code.

In CloudNativePG version 1.22 and onwards, the default configuration disables the use of `ALTER SYSTEM` on new Postgres clusters. This decision is rooted in the recognition of potential risks associated with this command. To enable the use of `ALTER SYSTEM` , you can explicitly set `.spec.postgresql.enableAlterSystem` to `true` .

> **Warning:** Proceed with caution when utilizing *ALTER SYSTEM* . This command operates directly on the connected instance and does not undergo replication. CloudNativePG assumes responsibility for certain fixed parameters and complete control over others, emphasizing the need for careful consideration.

When `.spec.postgresql.enableAlterSystem` is configured as `false` , any attempt to execute `ALTER SYSTEM` will result in an error. The error message might resemble the following:

```
ERROR:  could not open file "postgresql.auto.conf": Permission denied
```

Dynamic Shared Memory settings

PostgreSQL supports a few implementations for dynamic shared memory management through the dynamic_shared_memory_type

configuration option. In CloudNativePG we recommend to limit ourselves to any of the following two values:

- `posix` : which relies on POSIX shared memory allocated using `shm_open` (default setting)
- `sysv` : which is based on System V shared memory allocated via `shmget`

In PostgreSQL, this setting is particularly important for memory allocation in parallel queries. For details, please refer to this *thread from the `pgsql-general* mailing list <https://www.postgresql.org/message-id/CA%2BhUKGJOj7qzDLxeFPVvto8YEWop6FSQoTYPO9Z6Ee%3Di-nPS_Q%40mail.gmail.com>`_ .

POSIX shared memory

The default setting of `posix` should be enough in most cases, considering that the operator automatically mounts a *memory-bound ``EmptyDir`` volume* called `shm` under `/dev/shm` . You can verify the size of such volume inside the running Postgres container with:

```
mount | grep shm
```

You should get something similar to the following output:

```
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=******)
```

If you would like to set a maximum size for the `shm` volume, you can do so by setting the `.spec.ephemeralVolumesSizeLimit.shm` field in the `Cluster` resource. For example:

```
spec:
  ephemeralVolumesSizeLimit:
    shm: 1Gi
```

System V shared memory

In case your Kubernetes cluster has a high enough value for the `SHMMAX` and `SHMALL` parameters, you can also set:

```
dynamic_shared_memory_type: "sysv"
```

You can check the `SHMMAX` /`SHMALL` from inside a PostgreSQL container, by running:

```
ipcs -lm
```

For example:

```
- ----- Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398509481980
min seg size (bytes) = 1
```

As you can see, the very high number of `max total shared memory` recommends setting `dynamic_shared_memory_type` to `sysv` .

An alternate method is to run:

```
cat /proc/sys/kernel/shmall
cat /proc/sys/kernel/shmmax
```

Fixed parameters

Some PostgreSQL configuration parameters should be managed exclusively by the operator. The operator prevents the user from setting them using a webhook.

Users are not allowed to set the following configuration parameters in the `postgresql` section:

- `allow_system_table_mods`
- `archive_cleanup_command`
- `archive_command`
- `archive_mode`
- `bonjour`
- `bonjour_name`
- `cluster_name`
- `config_file`
- `data_directory`
- `data_sync_retry`
- `event_source`
- `external_pid_file`
- `full_page_writes`
- `hba_file`

- hot_standby
- ident_file
- jit_provider
- listen_addresses
- log_destination
- log_directory
- log_file_mode
- log_filename
- log_rotation_age
- log_rotation_size
- log_truncate_on_rotation
- logging_collector
- port
- primary_conninfo
- primary_slot_name
- promote_trigger_file
- recovery_end_command
- recovery_min_apply_delay
- recovery_target
- recovery_target_action
- recovery_target_inclusive
- recovery_target_lsn
- recovery_target_name
- recovery_target_time
- recovery_target_timeline
- recovery_target_xid
- restart_after_crash
- restore_command
- shared_preload_libraries
- ssl
- ssl_ca_file
- ssl_cert_file
- ssl_crl_file
- ssl_dh_params_file
- ssl_ecdh_curve
- ssl_key_file

- ssl_passphrase_command
- ssl_passphrase_command_supports_reload
- ssl_prefer_server_ciphers
- stats_temp_directory
- synchronous_standby_names
- syslog_facility
- syslog_ident
- syslog_sequence_numbers
- syslog_split_messages
- unix_socket_directories
- unix_socket_group
- unix_socket_permissions

1.4.23 Database Role Management

From its inception, CloudNativePG has managed the creation of specific roles required in PostgreSQL instances:

- some reserved users, such as the `postgres` superuser, `streaming_replica` and `cnpg_pooler_pgbouncer` (when the PgBouncer `Pooler` is used)
- The application user, set as the low-privilege owner of the application database

This process is described in the *Bootstrap* section.

With the `managed` stanza in the cluster spec, CloudNativePG now provides full lifecycle management for roles specified in `.spec.managed.roles` .

This feature enables declarative management of existing roles, as well as the creation of new roles if they are not already present in the database. Role creation will occur *after* the database bootstrapping is complete.

An example manifest for a cluster with declarative role management can be found in the file.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-roles
spec:
  instances: 3
  storage:
    size: 1Gi

  managed:
    roles:
    - name: app
      createdb: true
      login: true
    - name: dante
      ensure: present
      comment: my database-side comment
      login: true
```

(continues on next page)

(continued from previous page)

```
      superuser: false
      createdb: true
      createrole: false
      inherit: false
      replication: false
      bypassrls: false
      connectionLimit: 4
      validUntil: "2053-04-12T15:04:05Z"
      inRoles:
        - pg_monitor
        - pg_signal_backend
      passwordSecret:
        name: cluster-example-dante
- --
apiVersion: v1
data:
  username: ZGFudGU=
  password: ZGFudGU=
kind: Secret
metadata:
  name: cluster-example-dante
type: kubernetes.io/basic-auth
```

Here is an excerpt from that file:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
spec:
  managed:
    roles:
    - name: dante
      ensure: present
      comment: Dante Alighieri
      login: true
      superuser: false
      inRoles:
        - pg_monitor
        - pg_signal_backend
```

The role specification in `.spec.managed.roles` adheres to the PostgreSQL structure and naming conventions . Please refer to the *API reference* for the full list of attributes you can define for each role.

A few points are worth noting:

1. The `ensure` attribute is **not** part of PostgreSQL. It enables declarative role management to create and remove roles. The two possible values are `present` (the default) and `absent` .
2. The `inherit` attribute is true by default, following PostgreSQL conventions.
3. The `connectionLimit` attribute defaults to -1, in line with PostgreSQL conventions.
4. Role membership with `inRoles` defaults to no memberships.

Declarative role management ensures that PostgreSQL instances align with the spec. If a user modifies role attributes directly in the database, the CloudNativePG operator will revert those changes during the next reconciliation cycle.

Password management

The declarative role management feature includes reconciling of role passwords. Passwords are managed in fundamentally different ways in the Kubernetes world and in PostgreSQL, and as a result there are a few things to note.

Managed role configurations may optionally specify the name of a **Secret** where the username and password are stored (encoded in Base64 as is usual in Kubernetes). For example:

```
managed:
  roles:
  - name: dante
    ensure: present
    [··· snipped ···]
    passwordSecret:
      name: cluster-example-dante
```

This would assume the existence of a Secret called `cluster-example-dante` , containing a username and password. The username should match the role we are setting the password for. For example, :

```
apiVersion: v1
data:
  username: ZGFudGU=
  password: ZGFudGU=
kind: Secret
metadata:
  name: cluster-example-dante
  labels:
    cnpg.io/reload: "true"
type: kubernetes.io/basic-auth
```

If there is no `passwordSecret` specified for a role, the instance manager will not try to CREATE / ALTER the role with a password. This keeps with PostgreSQL conventions, where ALTER will not update passwords unless directed to with `WITH PASSWORD` .

If a role was initially created with a password, and we would like to set the password to NULL in PostgreSQL, this necessitates being explicit on the part of the user of CloudNativePG. To distinguish "no password provided in spec" from "set the password to NULL" , the field `DisablePassword` should be used.

Imagine we decided we would like to have no password on the `dante` role in the database. In such case we would specify the following:

```
managed:
  roles:
  - name: dante
    ensure: present
    [··· snipped ···]
    disablePassword: true
```

NOTE: it is considered an error to set both `passwordSecret` and `disablePassword` on a given role. This configuration will be rejected by the validation webhook.

Password expiry, `VALID UNTIL`

The `VALID UNTIL` role attribute in PostgreSQL controls password expiry. Roles created without `VALID UNTIL` specified get NULL by default in PostgreSQL, meaning that their password will never expire.

PostgreSQL uses a timestamp type for `VALID UNTIL` , which includes support for the value `'infinity'` indicating that the password never expires. Please see the PostgreSQL documentation

for reference.

With declarative role management, the `validUntil` attribute for managed roles controls password expiry. `validUntil` can only take:

- a Kubernetes timestamp, or
- be omitted (defaulting to `null`)

In the first case, the given `validUntil` timestamp will be set in the database as the `VALID UNTIL` attribute of the role.

In the second case (omitted `validUntil`) the operator will ensure password never expires, mirroring the behavior of PostgreSQL. Specifically:

- in case of new role, it will omit the `VALID UNTIL` clause in the role creation statement
- in case of existing role, it will set `VALID UNTIL` to `infinity` if `VALID UNTIL` was not set to `NULL` in the database (this is due to PostgreSQL not allowing `VALID UNTIL NULL` in the `ALTER ROLE` SQL statement)

> **Warning:** The declarative role management feature has changed behavior since its initial version (1.20.0). In 1.20.0, a role without a *passwordSecret* would lead to setting the password to NULL in PostgreSQL. In practice there is little difference from 1.20.0. New roles created without *passwordSecret* will have a *NULL* password. The relevant change is when using the managed roles to manage roles that had been previously created. In 1.20.0, doing this might inadvertently result in setting existing passwords to *NULL* .

Password hashed

You can also provide pre-encrypted passwords by specifying the password in MD5/SCRAM-SHA-256 hash format:

```
kind: Secret
type: kubernetes.io/basic-auth
metadata:
  name: cluster-example-cavalcanti
  labels:
    cnpg.io/reload: "true"
apiVersion: v1
stringData:
  username: cavalcanti
  password: SCRAM-SHA-256$<iteration count>:<salt>$<StoredKey>:<ServerKey>
```

Unrealizable role configurations

In PostgreSQL, in some cases, commands cannot be honored by the database and will be rejected. Please refer to the PostgreSQL documentation on error codes

for details.

Role operations can produce such fundamental errors. Two examples:

- We ask PostgreSQL to create the role `petrarca` as a member of the role (group) `poets` , but `poets` does not exist.
- We ask PostgreSQL to drop the role `dante` , but the role `dante` is the owner of the database `inferno` .

These fundamental errors cannot be fixed by the database, nor the CloudNativePG operator, without clarification from the human administrator. The two examples above could be fixed by creating the role `poets` or dropping the database `inferno` respectively, but they might have originated due to human error, and in such case, the "fix" proposed might be the wrong thing to do.

CloudNativePG will record when such fundamental errors occur, and will display them in the cluster Status. Which segues into⋯

Status of managed roles

The Cluster status includes a section for the managed roles' status, as shown below:

```
status:
  [⋯snipped⋯]
  managedRolesStatus:
    byStatus:
      not-managed:
      - app
      pending-reconciliation:
      - dante
      - petrarca
      reconciled:
      - ariosto
      reserved:
      - postgres
      - streaming_replica
    cannotReconcile:
      dante:
      - could not perform DELETE on role dante: owner of database inferno
      petrarca:
      - could not perform UPDATE_MEMBERSHIPS on role petrarca: role "poets" does not␣
↪exist
```

Note the special sub-section `cannotReconcile` for operations the database (and CloudNativePG) cannot honor, and which require human intervention.

This section covers roles reserved for operator use and those that are **not** under declarative management, providing a comprehensive view of the roles in the database instances.

The *Kubectl Plugin* also shows the status of managed roles in its `status` sub-command:

```
Managed roles status
Status                  Roles
- -----                   -----
pending-reconciliation  petrarca
reconciled              app,dante
reserved                postgres,streaming_replica

Irreconcilable roles
Role      Errors
- ---      ------
petrarca  could not perform UPDATE_MEMBERSHIPS on role petrarca: role "poets" does not␣
↪exist
```

Important: In terms of backward compatibility, declarative role management is designed to ignore roles that exist in the database but are not included in the spec. The lifecycle of these roles will continue to be managed within PostgreSQL, allowing CloudNativePG users to adopt this feature at their convenience.

1.4.24 Tablespaces

A tablespace is a robust and widely embraced feature in database management systems. It offers a powerful means to enhance the vertical scalability of a database by decoupling the physical and logical modeling of data. Essentially, it serves as a technique for physical database modeling, enabling the efficient distribution of I/O operations across multiple volumes on distinct storage. It thereby optimizes performance through parallel on-disk read/write operations.

In the context of the database industry, tablespaces play a strategic role, particularly when paired with table partitioning, a logical database modeling technique. They prove instrumental in managing large-scale databases and are also used for tasks such as separating tables from indexes or executing temporary operations.

Tablespaces in PostgreSQL have been playing a pivotal role since 2005 (version 8.0), while declarative partitioning was introduced in 2017 (version 10). Consequently, tablespaces are seamlessly integrated into all supported releases of PostgreSQL. Quoting from the *PostgreSQL documentation on tablespaces* :

> By using tablespaces, an administrator can control the disk layout of a > PostgreSQL installation. This is useful in at least two ways. > > - First, if the partition or volume on which the cluster was initialized runs > out of space and cannot be extended, a tablespace can be created on a > different partition and used until the system can be reconfigured. > - Second, tablespaces allow an administrator to use knowledge of the usage > pattern of database objects to optimize performance.

Declarative tablespaces

CloudNativePG provides support for PostgreSQL tablespaces through *declarative tablespaces*, operating at two distinct levels:

- Kubernetes, managing persistent volume claims, identically to how PGDATA and WAL volumes are handled
- PostgreSQL, managing the `TABLESPACE` global objects in the PostgreSQL instance

Being a part of the Kubernetes ecosystem, CloudNativePG' s declarative tablespaces are implemented by leveraging persistent volume claims (and persistent volumes). Each tablespace defined in the cluster is housed in its own persistent volume. CloudNativePG takes care of generating the PVCs. It mounts the required volumes in the instance pods in normalized locations and ensures replicas are ready to support tablespaces before activating them in the primary.

You can set up tablespaces when creating the cluster or add them later, provided the storage is available when requested. Currently, you can't remove them. However, this limitation will be addressed in a future minor or patch version of CloudNativePG.

Using declarative tablespaces

Using declarative tablespaces is straightforward. You can find a full example in

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-tablespaces
spec:
  instances: 3

  storage:
    size: 1Gi

  tablespaces:
    - name: atablespace
      storage:
        size: 1Gi
        storageClass: standard
      temporary: true
    - name: another_tablespace
      storage:
        size: 2Gi
        storageClass: standard
      temporary: true
    - name: tablespacea1
      storage:
        size: 2Gi
        storageClass: standard
```

.

To use them, use the new `tablespaces` stanza on a new or existing `Cluster` resource:

```yaml
spec:
  instances: 3

  # ...

  tablespaces:
    - name: tbs1
      storage:
        size: 1Gi
    - name: tbs2
      storage:
        size: 2Gi
    - name: tbs3
      storage:
        size: 2Gi
```

Each tablespace has its own storage section where you can configure the size and the storage class of the generated PVC. The administrator can thus plan to use different storage classes for different kinds of workloads, as explained in *Storage classes and tablespaces* .

CloudNativePG creates the persistent volume claims for each instance in the high-availability Postgres cluster. It mounts them in each pod when they have been provisioned. Then, it ensures that the `tbs1` , `tbs2` , and `tbs3` tablespaces are created on the primary PostgreSQL instance using the `CREATE TABLESPACE` command. This process is quick, and you see this reflected in Postgres:

```
app=# SELECT oid, spcname FROM pg_tablespace;
  oid  |      spcname
- ------+--------------------
  1663 | pg_default
  1664 | pg_global
 16387 | tbs1
 16388 | tbs2
 16389 | tbs3
(5 rows)
```

You can start using them right away:

```
app=# CREATE TABLE fibonacci(num INTEGER) TABLESPACE tbs1;
CREATE TABLE
```

The cluster status has a section for tablespaces:

```
status:

  <- snipped ->
  tablespacesStatus:
  - name: atablespace
    state: reconciled
  - name: another_tablespace
    state: reconciled
  - name: tablespacea1
    state: reconciled
```

Storage classes and tablespaces

You can use different storage classes for your tablespaces, just as you can for PGDATA and WAL volumes. This is a convenient way of optimizing your resources, balancing performance and costs of your storage based on data access usage and expectations.

This example helps to explain the feature:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: yardbirds
spec:
  instances: 3

  storage:
```

(continues on next page)

(continued from previous page)

```
    size: 10Gi
  walStorage:
    size: 10Gi
  tablespaces:
    - name: current
      size: 100Gi
      storageClass: fastest
    - name: this_year
      size: 500Gi
      storageClass: balanced
```

The `yardbirds` cluster example requests 4 persistent volume claims using 3 different storage classes:

- Default storage class – Used by the `PGDATA` and WAL volumes.
- `fastest` – Used by the `current` tablespace to store the most active and demanding set of data in the database.
- `balanced` – Used by the `this_year` tablespace to store older partitions of data that are rarely accessed by users and where performance expectations aren't the highest.

You can then take advantage of horizontal table partitioning and create the current month's table (for example, facts for December 2023) in the `current` tablespace:

```
CREATE TABLE facts_202312 PARTITION OF facts
    FOR VALUES FROM (2023-12-01) TO (2024-01-01)
    TABLESPACE current;
```

Important: This example assumes you're familiar with PostgreSQL declarative partitioning .

Tablespace ownership

By default, unless otherwise specified, tablespaces are owned by the `app` application user, as defined in `.spec.bootstrap.initdb.owner` . See *Bootstrap a new cluster* for details. This default behavior works in most microservice database use cases.

You can set the owner of a tablespace in the `owner` stanza, for example the `postgres` user, like in the following excerpt:

```
# ...
tablespaces:
  - name: clapton
    owner:
      name: postgres
    storage:
      size: 1Gi
```

Important: If you change the ownership of a tablespace, make sure that you're using an existing role. Otherwise, the status of the cluster reports the issue and stops reconciling tablespaces until fixed. It's your responsibility to monitor the status and the log and to promptly intervene by fixing the issue.

If you define a tablespace with an owner that doesn't exist, CloudNativePG can't create the tablespace and reflects this in the cluster status:

```yaml
spec:
  instances: 3

  # ...

  tablespaces:
    - name: tbs1
      storage:
        size: 1Gi
    - name: tbs2
      storage:
        size: 2Gi
    - name: tbs3
      owner:
        name: badhombre
      storage:
        size: 2Gi
        status:

  <- snipped ->
  tablespacesStatus:
  - name: tbs1
    status: reconciled
  - name: tbs2
    status: reconciled
  - error: while creating tablespace tbs3: ERROR: role "badhombre" does
      not exist (SQLSTATE 42704)
    name: tbs3
    status: pending
```

Backup and recovery

CloudNativePG handles backup of tablespaces (and the relative tablespace map) both on object stores and volume snapshots.

> **Warning:** By default, backups are taken from replica nodes. A backup taken immediately after creating tablespaces in a cluster can result in an incomplete view of the tablespaces from the replica and thus an incomplete backup. The lag will be resolved in a maximum of 5 minutes, with the next reconciliation.

Once a cluster with tablespaces has a base backup, you can restore a new cluster from it. When it comes to the recovery side, it's your responsibility to ensure that the `Cluster` definition of the recovered database contains the exact list of tablespaces.

Replica clusters

Replica clusters must have the same tablespace definition as their origin. The reason is that tablespace management commands like `CREATE TABLESPACE` are WAL logged and are replayed by any physical replication client (streaming or by way of WAL shipping).

It's your responsibility to ensure that replica clusters have the same list of tablespaces, with the same name. Storage class and size might vary.

For example:

```
spec:

  # ...
  bootstrap:
    recovery:
      # ... your selected recovery method

  tablespaces:
    - name: tbs1
      storage:
        size: 1Gi
    - name: tbs2
      storage:
        size: 2Gi
    - name: tbs3
      storage:
        size: 2Gi
```

Temporary tablespaces

PostgreSQL allows you to define one or more temporary tablespaces to create temporary objects (temporary tables and indexes on temporary tables) when a `CREATE` command doesn't explicitly specify a tablespace, and to create temporary files for purposes such as sorting large data sets. When no temporary tablespace is specified, PostgreSQL uses the default tablespace of a database, which is currently the main `PGDATA` volume.

When you specify more than one temporary tablespace, PostgreSQL randomly picks one the first time a temporary object needs to be created in a transaction. Then it sequentially iterates through the list.

Temporary tablespaces also work like regular tablespaces with regard to backups.

CloudNativePG provides the `.spec.tablespaces[*].temporary` option to determine whether to add a tablespace to the `temp_tablespaces` PostgreSQL parameter and thus become eligible to store temporary data that doesn't have an explicit tablespace assignment.

```
spec:
  [...]
  tablespaces:
    - name: atablespace
      storage:
        size: 1Gi
      temporary: true
```

They can be created at initialization time or added later, requiring a rolling update. The `temporary: true/false` option adds or removes the tablespace name to or from the list of tablespaces in the `temp_tablespaces` option. This

change doesn't require a restart of PostgreSQL.

Although temporary tablespaces can also work as regular tablespaces (meaning that users can also host regular data on them while using them for temporary operations), we recommend that you don't mix the two workloads.

See the *PostgreSQL documentation on `temp_tablespaces* <https://www.postgresql.org/docs/current/runtime-config-client.html#GUC-TEMP-TABLESPACES>`_

for details.

kubectl plugin support

The *kubectl status* plugin includes a section dedicated to tablespaces that offers a convenient overview, including tablespace status, owner, temporary flag, and any errors:

```
[...]

Tablespaces status
Tablespace          Owner  Status      Temporary  Error
- ---------          -----  ------      ---------  -----
atablespace         app    reconciled  true
another_tablespace  app    reconciled  true
tablespacea1        app    reconciled  false

Instances status
[...]
```

Limitations

Currently, you can't remove tablespaces from an existing CloudNativePG cluster.

1.4.25 Operator configuration

The operator for CloudNativePG is installed from a standard deployment manifest and follows the convention over configuration paradigm. While this is fine in most cases, there are some scenarios where you want to change the default behavior, such as:

- defining annotations and labels to be inherited by all resources created by the operator and that are set in the cluster resource
- defining a different default image for PostgreSQL or an additional pull secret

By default, the operator is installed in the `cnpg-system` namespace as a Kubernetes `Deployment` called `cnpg-controller-manager` .

Note: In the examples below we assume the default name and namespace for the operator deployment.

The behavior of the operator can be customized through a `ConfigMap` /`Secret` that is located in the same namespace of the operator deployment and with `cnpg-controller-manager-config` as the name.

Important: Any change to the config's *ConfigMap /Secret* will not be automatically detected by the operator, - and as such, it needs to be reloaded (see below). Moreover, changes only apply to the resources created after the configuration is reloaded.

Important: The operator first processes the ConfigMap values and then the Secret' s, in this order. As a result, if a parameter is defined in both places, the one in the Secret will be used.

Available options

The operator looks for the following environment variables to be defined in the `ConfigMap` /`Secret` :

Name | Description —- | ——— – `INHERITED_ANNOTATIONS` | list of annotation names that, when defined in a `Cluster` metadata, will be inherited by all the generated resources, including pods `INHERITED_LABELS` | list of label names that, when defined in a `Cluster` metadata, will be inherited by all the generated resources, including pods `PULL_SECRET_NAME` | name of an additional pull secret to be defined in the operator' s namespace and to be used to download images `ENABLE_AZURE_PVC_UPDATES` | Enables to delete Postgres pod if its PVC is stuck in Resizing condition. This feature is mainly for the Azure environment (default `false`) `ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES` | when set to `true` , enables in-place updates of the instance manager after an update of the operator, avoiding rolling updates of the cluster (default `false`) `MONITORING_QUERIES_CONFIGMAP` | The name of a ConfigMap in the operator' s namespace with a set of default queries (to be specified under the key `queries`) to be applied to all created Clusters `MONITORING_QUERIES_SECRET` | The name of a Secret in the operator' s namespace with a set of default queries (to be specified under the key `queries`) to be applied to all created Clusters `CERTIFICATE_DURATION` | Determines the lifetime of the generated certificates in days. Default is 90. `EXPIRING_CHECK_THRESHOLD` | Determines the threshold, in days, for identifying a certificate as expiring. Default is 7. `CREATE_ANY_SERVICE` | when set to `true` , will create `-any` service for the cluster. Default is `false`

Values in `INHERITED_ANNOTATIONS` and `INHERITED_LABELS` support path-like wildcards. For example, the value `example.com/*` will match both the value `example.com/one` and `example.com/two` .

When you specify an additional pull secret name using the `PULL_SECRET_NAME` parameter, the operator will use that secret to create a pull secret for every created PostgreSQL cluster. That secret will be named `<cluster-name>-pull` .

The namespace where the operator looks for the `PULL_SECRET_NAME` secret is where you installed the operator. If the operator is not able to find that secret, it will ignore the configuration parameter.

Warning: Previous versions of the operator copied the *PULL_SECRET_NAME* secret inside the namespaces where you deploy the PostgreSQL clusters. From version "1.11.0" the behavior changed to match the previous description. The pull secrets created by the previous versions of the operator are unused.

Defining an operator config map

The example below customizes the behavior of the operator, by defining the label/annotation names to be inherited by the resources created by any `Cluster` object that is deployed at a later time, and by enabling in-place updates for the instance manager.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cnpg-controller-manager-config
  namespace: cnpg-system
data:
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload, app
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: true
```

Defining an operator secret

The example below customizes the behavior of the operator, by defining the label/annotation names to be inherited by the resources created by any `Cluster` object that is deployed at a later time, and by enabling in-place updates for the instance manager.

```yaml
apiVersion: v1
kind: Secret
metadata:
  name: cnpg-controller-manager-config
  namespace: cnpg-system
type: Opaque
stringData:
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload, app
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: true
```

Restarting the operator to reload configs

For the change to be effective, you need to recreate the operator pods to reload the config map. If you have installed the operator on Kubernetes using the manifest you can do that by issuing:

```bash
kubectl rollout restart deployment \
    -n cnpg-system \
    cnpg-controller-manager
```

In general, given a specific namespace, you can delete the operator pods with the following command:

```bash
kubectl delete pods -n [NAMESPACE_NAME_HERE] \
  -l app.kubernetes.io/name=cloudnative-pg
```

> **Warning:** Customizations will be applied only to *Cluster* resources created after the reload of the operator deployment.

Following the above example, if the `Cluster` definition contains a `categories` annotation and any of the `environment` , `workload` , or `app` labels, these will be inherited by all the resources generated by the deployment.

pprof HTTP Server

The operator can expose a PPROF HTTP server with the following endpoints on `localhost:6060` :

- `/debug/pprof/` . Responds to a request for "/debug/pprof/" with an HTML page listing the available profiles
- `/debug/pprof/cmdline` . Responds with the running program's command line, with arguments separated by NULL bytes.
- `/debug/pprof/profile` . Responds with the pprof-formatted cpu profile. Profiling lasts for duration specified in seconds GET parameter, or for 30 seconds if not specified.
- `/debug/pprof/symbol` . Looks up the program counters listed in the request, responding with a table mapping program counters to function names.
- `/debug/pprof/trace` . Responds with the execution trace in binary form. Tracing lasts for duration specified in seconds GET parameter, or for 1 second if not specified.

To enable the operator you need to edit the operator deployment add the flag `--pprof-server=true` .

You can do this by executing these commands:

```
kubectl edit deployment -n cnpg-system cnpg-controller-manager
```

Then on the edit page scroll down the container args and add `--pprof-server=true` , example:

```
containers:
- args:
  - controller
  - --enable-leader-election
  - --config-map-name=cnpg-controller-manager-config
  - --secret-name=cnpg-controller-manager-config
  - --log-level=info
  - --pprof-server=true # relevant line
  command:
  - /manager
```

Save the changes, the deployment now will execute a rollout and the new pod will have the PPROF server enabled.

Once the pod is running you can exec inside the container by doing:

```
kubectl exec -ti -n cnpg-system <pod name> -- bash
```

Once inside execute:

```
curl localhost:6060/debug/pprof/
```

1.4.26 Instance pod configuration

Projected volumes

CloudNativePG supports mounting custom files inside the Postgres pods through `.spec.projectedVolumeTemplate` . This ability is useful for several Postgres features and extensions that require additional data files. In CloudNativePG, the `.spec.projectedVolumeTemplate` field is a projected volume

template in Kubernetes that allows you to mount arbitrary data under the `/projected` folder in Postgres pods.

This simple example shows how to mount an existing TLS secret (named `sample-secret`) as files into Postgres pods. The values for the secret keys `tls.crt` and `tls.key` in `sample-secret` are mounted as files into the paths `/projected/certificate/tls.crt` and `/projected/certificate/tls.key` in the Postgres pod.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-projected-volumes
spec:
  instances: 3
  projectedVolumeTemplate:
    sources:
      - secret:
          name: sample-secret
          items:
            - key: tls.crt
              path: certificate/tls.crt
            - key: tls.key
              path: certificate/tls.key
  storage:
    size: 1Gi
```

You can find a complete example that uses a projected volume template to mount the secret and ConfigMap in the deployment manifest.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-projected-volume
spec:
  instances: 3
  projectedVolumeTemplate:
    sources:
      - secret:
          name: sample-secret
          items:
            - key: tls.crt
              path: certificate/tls.crt
            - key: tls.key
              path: certificate/tls.key
      - configMap:
          name: sample-configmap
          items:
            - key: key1
```

(continues on next page)

(continued from previous page)

```
              path: config/key1
            - key: key2
              path: config/key2
  storage:
    size: 1Gi

- --
apiVersion: v1
data:
  tls.crt:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNQekNDQWVXZ0F3SUJBZ0lRSHVDY2lKcDVkZis0dHZBcDBrTk9rekFLQmdncW
  tls.key:␣
↪LS0tLS1CRUdJTiBFQyBQUklWQVRFIEtFWS0tLS0tCk1IY0NBUUVFSUtucWxVQjFqU2sxWW5VZzAyb0tGbXlRdDJLUEZwaFc0K1lmQU
kind: Secret
metadata:
  name: sample-secret
type: kubernetes.io/tls
- --
apiVersion: v1
data:
  key1: value1
  key2: value2
  key3: value3
kind: ConfigMap
metadata:
  name: sample-configmap
```

Ephemeral volumes

CloudNativePG relies on ephemeral volumes

for part of the internal activities. Ephemeral volumes exist for the sole duration of a pod' s life, without persisting across pod restarts.

1.4.27 Volume Claim Template for Temporary Storage

The operator uses by default an `emptyDir` volume, which can be customized by using the `.spec.ephemeralVolumesSizeLimit field` . This can be overridden by specifying a volume claim template in the `.spec.ephemeralVolumeSource` field.

In the following example, a `1Gi` ephemeral volume is set.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-ephemeral-volume-source
spec:
  instances: 3
  ephemeralVolumeSource:
    volumeClaimTemplate:
      spec:
```

(continues on next page)

(continued from previous page)

```
        accessModes: ["ReadWriteOnce"]
        # example storageClassName, replace with one existing in your Kubernetes cluster
        storageClassName: "scratch-storage-class"
        resources:
          requests:
            storage: 1Gi
```

Both `.spec.emphemeralVolumeSource` and `.spec.ephemeralVolumesSizeLimit.temporaryData` cannot be specified simultaneously.

Volume for shared memory

This volume is used as shared memory space for Postgres and as an ephemeral type but stored in memory. You can configure an upper bound on the size using the `.spec.ephemeralVolumesSizeLimit.shm` field in the cluster spec. Use this field only in case of PostgreSQL running with `posix shared memory dynamic allocation <PostgreSQL Configuration>` .

Environment variables

You can customize some system behavior using environment variables. One example is the `LDAPCONF` variable, which can point to a custom LDAP configuration file. Another example is the `TZ` environment variable, which represents the timezone used by the PostgreSQL container.

CloudNativePG allows you to set custom environment variables using the `env` and the `envFrom` stanza of the cluster specification.

This example defines a PostgreSQL cluster using the `Australia/Sydney` timezone as the default cluster-level timezone:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3

  env:
  - name: TZ
    value: Australia/Sydney

  storage:
    size: 1Gi
```

The `envFrom` stanza can refer to ConfigMaps or secrets to use their content as environment variables:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
```

(continues on next page)

(continued from previous page)

```
envFrom:
- configMapRef:
    name: config-map-name
- secretRef:
    name: secret-name

storage:
  size: 1Gi
```

The operator doesn't allow setting the following environment variables:

- `POD_NAME`
- `NAMESPACE`
- Any environment variable whose name starts with `PG` .

Any change in the `env` or in the `envFrom` section triggers a rolling update of the PostgreSQL pods.

If the `env` or the `envFrom` section refers to a secret or a ConfigMap, the operator doesn't detect any changes in them and doesn't trigger a rollout. The kubelet uses the same behavior with pods, and you must trigger the pod rollout manually.

1.4.28 Storage

Storage is the most critical component in a database workload. Storage must always be available, scale, perform well, and guarantee consistency and durability. The same expectations and requirements that apply to traditional environments, such as virtual machines and bare metal, are also valid in container contexts managed by Kubernetes.

Important: When it comes to dynamically provisioned storage, Kubernetes has its own specifics. These include *storage classes*, *persistent volumes*, and *Persistent Volume Claims (PVCs)*. You need to own these concepts, on top of all the valuable knowledge you've built over the years in terms of storage for database workloads on VMs and physical servers.

There are two primary methods of access to storage:

- **Network** – Either directly or indirectly. (Think of an NFS volume locally mounted on a host running Kubernetes.)
- **Local** – Directly attached to the node where a pod is running. This also includes directly attached disks on bare metal installations of Kubernetes.

Network storage, which is the most common usage pattern in Kubernetes, presents the same issues of throughput and latency that you can experience in a traditional environment. These issues can be accentuated in a shared environment, where I/O contention with several applications increases the variability of performance results.

Local storage enables shared-nothing architectures, which is more suitable for high transactional and very large database (VLDB) workloads, as it guarantees higher and more predictable performance.

Warning: Before you deploy a PostgreSQL cluster with CloudNativePG, ensure that the storage you're using is recommended for database workloads. We recommend clearly setting performance expectations by first benchmarking the storage using tools such as fio

and then the database using pgbench .

Note: CloudNativePG doesn't use *StatefulSet* for managing data persistence. Rather, it manages PVCs directly. If you want to know more, see *Custom Pod Controller* .

Backup and recovery

Since CloudNativePG supports volume snapshots for both backup and recovery, we recommend that you also consider this aspect when you choose your storage solution, especially if you manage very large databases.

Important: See the Kubernetes documentation for a list of all the supported container storage interface (CSI) drivers

that provide snapshot capabilities.

Benchmarking CloudNativePG

Before deploying the database in production, we recommend that you benchmark CloudNativePG in a controlled Kubernetes environment. Follow the guidelines in *Benchmarking* .

Briefly, we recommend operating at two levels:

- Measuring the performance of the underlying storage using fio, with relevant metrics for database workloads such as throughput for sequential reads, sequential writes, random reads, and random writes
- Measuring the performance of the database using pgbench, the default benchmarking tool distributed with PostgreSQL

Important: You must measure both the storage and database performance before putting the database into production. These results are extremely valuable not just in the planning phase (for example, capacity planning). They are also valuable in the production lifecycle, particularly in emergency situations when you don't have time to run this kind of test. Databases change and evolve over time, and so does the distribution of data, potentially affecting performance. Knowing the theoretical maximum throughput of sequential reads or writes is extremely useful in those situations. This is true especially in shared-nothing contexts, where results don't vary due to the influence of external workloads.

Know your system: benchmark it.

Encryption at rest

Encryption at rest is possible with CloudNativePG. The operator delegates that to the underlying storage class. See the storage class for information about this important security feature.

Persistent Volume Claim (PVC)

The operator creates a PVC for each PostgreSQL instance, with the goal of storing the `PGDATA` . It then mounts it into each pod.

Additionally, it supports creating clusters with:

- A separate PVC on which to store PostgreSQL WAL, as explained in *Volume for WAL*
- Additional separate volumes reserved for PostgreSQL tablespaces, as explained in TablespaceState {postgresql-cnpg-io-v1-TablespaceState}

In CloudNativePG, the volumes attached to a single PostgreSQL instance are defined as a *PVC group*.

Configuration via a storage class

Important: CloudNativePG was designed to work interchangeably with all storage classes. As usual, we recommend properly benchmarking the storage class in a controlled environment before deploying to production.

The easiest way to configure the storage for a PostgreSQL class is to request storage of a certain size, like in the following example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-class
spec:
  instances: 3
  storage:
    size: 1Gi
```

Using the previous configuration, the generated PVCs are satisfied by the default storage class. If the target Kubernetes cluster has no default storage class, or even if you need your PVCs to be satisfied by a known storage class, you can set it into the custom resource:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-class
spec:
  instances: 3
  storage:
    storageClass: standard
    size: 1Gi
```

Configuration via a PVC template

To further customize the generated PVCs, you can provide a PVC template inside the custom resource, like in the following example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-pvc-template
spec:
  instances: 3

  storage:
    pvcTemplate:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
      storageClassName: standard
      volumeMode: Filesystem
```

Volume for WAL

By default, PostgreSQL stores all its data in the so-called `PGDATA` (a directory). One of the core directories inside `PGDATA` is `pg_wal` , which contains the log of transactional changes that occurred in the database, in the form of segment files. (`pg_wal` is historically known as `pg_xlog` in PostgreSQL.)

Note: Normally, each segment is 16MB in size, but you can configure the size using the *walSegmentSize* option. This option is applied at cluster initialization time, as described in Bootstrap an empty cluster (`initdb ) <Bootstrap an empty cluster (*initdb* )>` .

In most cases, having `pg_wal` on the same volume where `PGDATA` resides is fine. However, having WALs stored in a separate volume has a few benefits:

- **I/O performance** – By storing WAL files on different storage from `PGDATA` , PostgreSQL can exploit parallel I/O for WAL operations (normally sequential writes) and for data files (tables and indexes for example), thus improving vertical scalability.
- **More reliability** – By reserving dedicated disk space to WAL files, you can be sure that exhausting space on the `PGDATA` volume never interferes with WAL writing. This behavior ensures that your PostgreSQL primary is correctly shut down.
- **Finer control** – You can define the amount of space dedicated to both `PGDATA` and `pg_wal` , fine tune WAL configuration and checkpoints, and even use a different storage class for cost optimization.
- **Better I/O monitoring** – You can constantly monitor the load and disk usage on both `PGDATA` and `pg_wal` . You can also set alerts that notify you in case, for example, `PGDATA` requires resizing.

See also:

See Reliability and the Write-Ahead Log

in the PostgreSQL documentation for more information.

You can add a separate volume for WAL using the `.spec.walStorage` option. It follows the same rules described for the `storage` field and provisions a dedicated PVC. For example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: separate-pgwal-volume
spec:
  instances: 3
  storage:
    size: 1Gi
  walStorage:
    size: 1Gi
```

Important: Removing *walStorage* isn't supported. Once added, a separate volume for WALs can't be removed from an existing Postgres cluster.

Volumes for tablespaces

CloudNativePG supports declarative tablespaces. You can add one or more volumes, each dedicated to a single PostgreSQL tablespace. See TablespaceState {postgresql-cnpg-io-v1-TablespaceState} for details.

Volume expansion

Kubernetes exposes an API allowing *expanding PVCs*

that' s enabled by default. However, it needs to be supported by the underlying `StorageClass` .

To check if a certain `StorageClass` supports volume expansion, you can read the `allowVolumeExpansion` field for your storage class:

```
$ kubectl get storageclass -o jsonpath={$.allowVolumeExpansion} premium-storage
true
```

Using the volume expansion Kubernetes feature

Given the storage class supports volume expansion, you can change the size requirement of the `Cluster` , and the operator applies the change to every PVC.

If the `StorageClass` supports *online volume resizing* , the change is immediately applied to the pods. If the underlying storage class doesn' t support that, you must delete the pod to trigger the resize.

The best way to proceed is to delete one pod at a time, starting from replicas and waiting for each pod to be back up.

Expanding PVC volumes on AKS

Currently, Azure can resize the PVC's volume without restarting the pod only on specific regions . CloudNativePG has overcome this limitation through the `ENABLE_AZURE_PVC_UPDATES` environment variable in the *Operator configuration* . When set to `true` , CloudNativePG triggers a rolling update of the Postgres cluster.

Alternatively, you can use the following workaround to manually resize the volume in AKS.

Workaround for volume expansion on AKS

You can manually resize a PVC on AKS. As an example, suppose you have a cluster with three replicas:

```
$ kubectl get pods
NAME                 READY   STATUS    RESTARTS   AGE
cluster-example-1    1/1     Running   0          2m37s
cluster-example-2    1/1     Running   0          2m22s
cluster-example-3    1/1     Running   0          2m10s
```

An Azure disk can be expanded only while in "unattached" state, as described in the Kubernetes documentation . This means that, to resize a disk used by a PostgreSQL cluster, you need to perform a manual rollout, first cordoning the node that hosts the pod using the PVC bound to the disk. This prevents the operator from re-creating the pod and immediately reattaching it to its PVC before the background disk resizing is complete.

First, edit the cluster definition, applying the new size. In this example, the new size is `2Gi` .

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3

  storage:
    storageClass: default
    size: 2Gi
```

Assuming the `cluster-example-1` pod is the cluster's primary, you can proceed with the replicas first. For example, start with cordoning the Kubernetes node that hosts the `cluster-example-3` pod:

```
kubectl cordon <node of cluster-example-3>
```

Then delete the `cluster-example-3` pod:

```
$ kubectl delete pod/cluster-example-3
```

Run the following command:

```
kubectl get pvc -w -o=jsonpath={.status.conditions[].message} cluster-example-3
```

Wait until you see the following output:

```
Waiting for user to (re-)start a Pod to finish file system resize of volume on node.
```

Then, you can uncordon the node:

```
kubectl uncordon <node of cluster-example-3>
```

Wait for the pod to be re-created correctly and get in a "Running and Ready" state:

```
kubectl get pods -w cluster-example-3
cluster-example-3   0/1     Init:0/1   0          12m
cluster-example-3   1/1     Running   0          12m
```

Verify the PVC expansion by running the following command, which returns `2Gi` as configured:

```
kubectl get pvc cluster-example-3 -o=jsonpath={.status.capacity.storage}
```

You can repeat these steps for the remaining pods.

Important: Leave the resizing of the disk associated with the primary instance as the last disk, after promoting through a switchover a new resized pod, using *kubectl cnpg promote* . For example, use *kubectl cnpg promote cluster-example 3* to promote *cluster-example-3* to primary.

Re-creating storage

If the storage class doesn't support volume expansion, you can still regenerate your cluster on different PVCs. Allocate new PVCs with increased storage and then move the database there. This operation is feasible only when the cluster contains more than one node.

While you do that, you need to prevent the operator from changing the existing PVC by disabling the `resizeInUseVolumes` flag, like in the following example:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-pvc-template
spec:
  instances: 3

  storage:
    storageClass: standard
    size: 1Gi
    resizeInUseVolumes: False
```

To move the entire cluster to a different storage area, you need to re-create all the PVCs and all the pods. Suppose you have a cluster with three replicas, like in the following example:

```
$ kubectl get pods
NAME                 READY   STATUS    RESTARTS   AGE
cluster-example-1    1/1     Running   0          2m37s
cluster-example-2    1/1     Running   0          2m22s
cluster-example-3    1/1     Running   0          2m10s
```

To re-create the cluster using different PVCs, you can edit the cluster definition to disable `resizeInUseVolumes` . Then re-create every instance in a different PVC.

For example, re-create the storage for `cluster-example-3` :

```
$ kubectl delete pvc/cluster-example-3 pod/cluster-example-3
```

Important: If you created a dedicated WAL volume, both PVCs must be deleted during this process. The same procedure applies if you want to regenerate the WAL volume PVC. You can do this by also disabling *resizeInUseVolumes* for the *.spec.walStorage* section.

For example, if a PVC dedicated to WAL storage is present:

```
$ kubectl delete pvc/cluster-example-3 pvc/cluster-example-3-wal pod/cluster-example-3
```

Having done that, the operator orchestrates creating another replica with a resized PVC:

```
$ kubectl get pods
NAME                          READY   STATUS      RESTARTS   AGE
cluster-example-1             1/1     Running     0          5m58s
cluster-example-2             1/1     Running     0          5m43s
cluster-example-4-join-v2     0/1     Completed   0          17s
cluster-example-4             1/1     Running     0          10s
```

Static provisioning of persistent volumes

CloudNativePG was designed to work with dynamic volume provisioning. This capability allows storage volumes to be created on demand when requested by users by way of storage classes and PVC templates. See *Re-creating storage* .

However, in some cases, Kubernetes administrators prefer to manually create storage volumes and then create the related `PersistentVolume` objects for their representation inside the Kubernetes cluster. This is also known as *pre-provisioning* of volumes.

Important: We recommend that you avoid pre-provisioning volumes, as it has an effect on the high availability and self-healing capabilities of the operator. It breaks the fully declarative model on which CloudNativePG was built. To use a pre-provisioned volume in CloudNativePG:

1. Manually create the volume outside Kubernetes.
2. Create the `PersistentVolume` object to match this volume using the correct parameters as required by the actual CSI driver (that is, `volumeHandle` , `fsType` , `storageClassName` , and so on).
3. Create the Postgres `Cluster` using, for each storage section, a coherent *pvcTemplate*

section that can help Kubernetes match the `PersistentVolume` and enable CloudNativePG to create the needed `PersistentVolumeClaim` .

Warning: With static provisioning, it's your responsibility to ensure that Postgres pods can be correctly scheduled by Kubernetes where a pre-provisioned volume exists. (The scheduling configuration is based on the affinity rules of your cluster.) Make sure you check for any pods stuck in *Pending* after you deploy the cluster. If the condition persists, investigate why it's happening.

Block storage considerations (Ceph/ Longhorn)

Most block storage solutions in Kubernetes recommend having multiple replicas of a volume to improve resiliency. This works well for workloads that don't have resiliency built into the application. However, CloudNativePG has this resiliency built directly into the Postgres `Cluster` through the number of instances and the persistent volumes that are attached to them.

In these cases, it makes sense to define the storage class used by the Postgres clusters as one replica. By having additional replicas defined in the storage solution (like Longhorn and Ceph), you might incur what's known as write amplification, unnecessarily increasing disk I/O and space used.

1.4.29 Labels and annotations

Resources in Kubernetes are organized in a flat structure, with no hierarchical information or relationship between them. However, such resources and objects can be linked together and put in relationship through *labels* and *annotations*.

Note: For more information, see the Kubernetes documentation on annotations and labels .

In brief:

- An annotation is used to assign additional non-identifying information to resources with the goal of facilitating integration with external tools.
- A label is used to group objects and query them through the Kubernetes native selector capability.

You can select one or more labels or annotations to use in your CloudNativePG deployments. Then you need to configure the operator so that when you define these labels or annotations in a cluster's metadata, they're inherited by all resources created by it (including pods).

Note: Label and annotation inheritance is the technique adopted by CloudNativePG instead of alternative approaches such as pod templates.

Predefined labels

These predefined labels are managed by CloudNativePG.

`cnpg.io/backupDate` : The date of the backup in ISO 8601 format (`YYYYMMDD`)

`cnpg.io/backupName` : Backup identifier, available only on `Backup` and `VolumeSnapshot` resources

`cnpg.io/backupMonth` : The year/month when a backup was taken

`cnpg.io/backupTimeline` : The timeline of the instance when a backup was taken

`cnpg.io/backupYear` : The year a backup was taken

`cnpg.io/cluster` : Name of the cluster

`cnpg.io/immediateBackup` : Applied to a `Backup` resource if the backup is the first one created from a `ScheduledBackup` object having `immediate` set to `true`

`cnpg.io/instanceName` : Name of the PostgreSQL instance (replaces the old and deprecated `postgresql` label)

`cnpg.io/jobRole` : Role of the job (that is, `import` , `initdb` , `join` , ⋯)

`cnpg.io/onlineBackup` : Whether the backup is online (hot) or taken when Postgres is down (cold)

`cnpg.io/podRole` : Distinguishes pods dedicated to pooler deployment from those used for database instances

`cnpg.io/poolerName` : Name of the PgBouncer pooler

`cnpg.io/pvcRole` : Purpose of the PVC, such as `PG_DATA` or `PG_WAL`

`cnpg.io/reload` : Available on `ConfigMap` and `Secret` resources. When set to `true` , a change in the resource is automatically reloaded by the operator.

`role` - **deprecated** : Whether the instance running in a pod is a `primary` or a `replica` . This label is deprecated, you should use `cnpg.io/instanceRole` instead.

`cnpg.io/scheduled-backup` : When available, name of the `ScheduledBackup` resource that created a given `Backup` object

`cnpg.io/instanceRole` : Whether the instance running in a pod is a `primary` or a `replica` .

Predefined annotations

These predefined annotations are managed by CloudNativePG.

`container.apparmor.security.beta.kubernetes.io/*` : Name of the AppArmor profile to apply to the named container. See *Restricting Pod access using AppArmor*

for details.

`cnpg.io/backupEndTime` : The time a backup ended.

`cnpg.io/backupEndWAL` : The WAL at the conclusion of a backup.

`cnpg.io/backupStartTime` : The time a backup started.

`cnpg.io/backupStartWAL` : The WAL at the start of a backup.

`cnpg.io/coredumpFilter` : Filter to control the coredump of Postgres processes, expressed with a bitmask. By default it' s set to `0x31` to exclude shared memory segments from the dump. See *PostgreSQL core dumps*

for more information.

`cnpg.io/clusterManifest` : Manifest of the `Cluster` owning this resource (such as a PVC). This label replaces the old, deprecated `cnpg.io/hibernateClusterManifest` label.

`cnpg.io/fencedInstances` : List of the instances that need to be fenced, expressed in JSON format. The whole cluster is fenced if the list contains the `*` element.

`cnpg.io/forceLegacyBackup` : Applied to a `Cluster` resource for testing purposes only, to simulate the behavior of `barman-cloud-backup` prior to version 3.4 (Jan 2023) when the `--name` option wasn' t available.

`cnpg.io/hash` : The hash value of the resource.

`cnpg.io/hibernation` : Applied to a `Cluster` resource to control the *declarative hibernation feature* . Allowed values are `on` and `off` .

`cnpg.io/managedSecrets` : Pull secrets managed by the operator and automatically set in the `ServiceAccount` resources for each Postgres cluster.

`cnpg.io/nodeSerial` : On a pod resource, identifies the serial number of the instance within the Postgres cluster.

`cnpg.io/operatorVersion` : Version of the operator.

`cnpg.io/pgControldata` : Output of the `pg_controldata` command. This annotation replaces the old, deprecated `cnpg.io/hibernatePgControlData` annotation.

`cnpg.io/podEnvHash` : Deprecated, as the `cnpg.io/podSpec` annotation now also contains the pod environment.

`cnpg.io/podSpec` : Snapshot of the `spec` of the pod generated by the operator. This annotation replaces the old, deprecated `cnpg.io/podEnvHash` annotation.

`cnpg.io/poolerSpecHash` : Hash of the pooler resource.

`cnpg.io/pvcStatus` : Current status of the PVC: `initializing` , `ready` , or `detached` .

`cnpg.io/reconciliationLoop` : When set to `disabled` on a `Cluster` , the operator prevents the reconciliation loop from running.

`cnpg.io/reloadedAt` : Contains the latest cluster `reload` time. `reload` is triggered by the user through a plugin.

`cnpg.io/skipEmptyWalArchiveCheck` : When set to `true` on a `Cluster` resource, the operator disables the check that ensures that the WAL archive is empty before writing data. Use at your own risk.

`cnpg.io/skipEmptyWalArchiveCheck` : When set to `true` on a `Cluster` resource, the operator disables WAL archiving. This will set `archive_mode` to `off` and require a restart of all PostgreSQL instances. Use at your own risk.

`cnpg.io/snapshotStartTime` : The time a snapshot started.

`cnpg.io/snapshotEndTime` : The time a snapshot was marked as ready to use.

`kubectl.kubernetes.io/restartedAt` : When available, the time of last requested restart of a Postgres cluster.

Prerequisites

By default, no label or annotation defined in the cluster' s metadata is inherited by the associated resources. To enable label/annotation inheritance, follow the instructions provided in *Operator configuration* .

The following continues from that example and limits it to the following:

- Annotations: `categories`
- Labels: `app` , `environment` , and `workload`

Note: Feel free to select the names that most suit your context for both annotations and labels. You can also use wildcards in naming and adopt strategies like using *mycompany/** for all labels or setting annotations starting with *mycompany/* to be inherited.

Defining cluster' s metadata

When defining the cluster, before any resource is deployed, you can set the metadata as follows:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
  annotations:
    categories: database
  labels:
    environment: production
    workload: database
    app: sso
spec:
     # ... <snip>
```

Once the cluster is deployed, you can verify, for example, that the labels were correctly set in the pods:

```
kubectl get pods --show-labels
```

Current limitations

Currently, CloudNativePG doesn't automatically propagate labels or annotations deletions. Therefore, when an annotation or label is removed from a cluster that was previously propagated to the underlying pods, the operator doesn't remove it on the associated resources.

1.4.30 Monitoring

Important: Installing Prometheus and Grafana is beyond the scope of this project. We assume they are correctly installed in your system. However, for experimentation we provide instructions in *Part 4 of the Quickstart* .

Monitoring Instances

For each PostgreSQL instance, the operator provides an exporter of metrics for Prometheus via HTTP, on port 9187, named `metrics` . The operator comes with a *Predefined set of metrics* , as well as a highly configurable and customizable system to define additional queries via one or more `ConfigMap` or `Secret` resources (see the *User defined metrics* below for details).

Important: Starting from version 1.11, CloudNativePG already installs by default a set of predefined metrics in a *ConfigMap* called *default-monitoring* .

Note: You can inspect the exported metrics by following the instructions in the *How to inspect the exported metrics*

section below.

All monitoring queries that are performed on PostgreSQL are:

- atomic (one transaction per query)
- executed with the `pg_monitor` role
- executed with `application_name` set to `cnpg_metrics_exporter`
- executed as user `postgres`

Please refer to the "Predefined Roles" section in PostgreSQL
documentation

for details on the `pg_monitor` role.

Queries, by default, are run against the *main database*, as defined by the specified `bootstrap` method of the `Cluster` resource, according to the following logic:

- using `initdb` : queries will be run by default against the specified database in `initdb.database` , or `app` if not specified
- using `recovery` : queries will be run by default against the specified database in `recovery.database` , or `postgres` if not specified

- using `pg_basebackup` : queries will be run by default against the specified database in `pg_basebackup.database` , or `postgres` if not specified

The default database can always be overridden for a given user-defined metric, by specifying a list of one or more databases in the `target_databases` option.

See also:

If you are interested in evaluating the integration of CloudNativePG with Prometheus and Grafana, you can find a quick setup guide in *Part 4 of the quickstart*

Prometheus Operator example

A specific PostgreSQL cluster can be monitored using the Prometheus Operator's resource PodMonitor . A PodMonitor correctly pointing to a Cluster can be automatically created by the operator by setting `.spec.monitoring.enablePodMonitor` to `true` in the Cluster resource itself (default: false).

Important: Any change to the *PodMonitor* created automatically will be overridden by the Operator at the next reconciliation cycle, in case you need to customize it, you can do so as described below.

To deploy a `PodMonitor` for a specific Cluster manually, you can just define it as follows, changing it as needed:

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cluster-example
spec:
  selector:
    matchLabels:
      "cnpg.io/cluster": cluster-example
  podMetricsEndpoints:
  - port: metrics
```

Important: Make sure you modify the example above with a unique name as well as the correct cluster's namespace and labels (we are using *cluster-example*).

Important: Label *postgresql* , used in previous versions of this document, is deprecated and will be removed in the future. Please use the label *cnpg.io/cluster* instead to select the instances.

Predefined set of metrics

Every PostgreSQL instance exporter automatically exposes a set of predefined metrics, which can be classified in two major categories:

- PostgreSQL related metrics, starting with `cnpg_collector_*` , including:
  - number of WAL files and total size on disk
  - number of `.ready` and `.done` files in the archive status folder

  - requested minimum and maximum number of synchronous replicas, as well as the expected and actually observed values
  - number of distinct nodes accommodating the instances
  - timestamps indicating last failed and last available backup, as well as the first point of recoverability for the cluster
  - flag indicating if replica cluster mode is enabled or disabled
  - flag indicating if a manual switchover is required
  - flag indicating if fencing is enabled or disabled
- Go runtime related metrics, starting with `go_*`

Below is a sample of the metrics returned by the `localhost:9187/metrics` endpoint of an instance. As you can see, the Prometheus format is self-documenting:

```
#  HELP cnpg_collector_collection_duration_seconds Collection time duration in seconds

#  TYPE cnpg_collector_collection_duration_seconds gauge

cnpg_collector_collection_duration_seconds{collector="Collect.up"} 0.0031393

#  HELP cnpg_collector_collections_total Total number of times PostgreSQL was accessed␣
↪for metrics.

#  TYPE cnpg_collector_collections_total counter

cnpg_collector_collections_total 2

#  HELP cnpg_collector_fencing_on 1 if the instance is fenced, 0 otherwise

#  TYPE cnpg_collector_fencing_on gauge

cnpg_collector_fencing_on 0

#  HELP cnpg_collector_nodes_used NodesUsed represents the count of distinct nodes␣
↪accommodating the instances. A value of -1 suggests that the metric is not available.␣
↪A value of 1 suggests that all instances are hosted on a single node, implying the␣
↪absence of High Availability (HA). Ideally this value should match the number of␣
↪instances in the cluster.

#  TYPE cnpg_collector_nodes_used gauge

cnpg_collector_nodes_used 3

#  HELP cnpg_collector_last_collection_error 1 if the last collection ended with error,␣
↪0 otherwise.

#  TYPE cnpg_collector_last_collection_error gauge

cnpg_collector_last_collection_error 0

#  HELP cnpg_collector_manual_switchover_required 1 if a manual switchover is required,␣
```

(continues on next page)

(continued from previous page)

```
↪0 otherwise

#  TYPE cnpg_collector_manual_switchover_required gauge

cnpg_collector_manual_switchover_required 0

#  HELP cnpg_collector_pg_wal Total size in bytes of WAL segments in the /var/lib/
↪postgresql/data/pgdata/pg_wal directory  computed as (wal_segment_size * count)

#  TYPE cnpg_collector_pg_wal gauge

cnpg_collector_pg_wal{value="count"} 9
cnpg_collector_pg_wal{value="slots_max"} NaN
cnpg_collector_pg_wal{value="keep"} 32
cnpg_collector_pg_wal{value="max"} 64
cnpg_collector_pg_wal{value="min"} 5
cnpg_collector_pg_wal{value="size"} 1.50994944e+08
cnpg_collector_pg_wal{value="volume_max"} 128
cnpg_collector_pg_wal{value="volume_size"} 2.147483648e+09

#  HELP cnpg_collector_pg_wal_archive_status Number of WAL segments in the /var/lib/
↪postgresql/data/pgdata/pg_wal/archive_status directory (ready, done)

#  TYPE cnpg_collector_pg_wal_archive_status gauge

cnpg_collector_pg_wal_archive_status{value="done"} 6
cnpg_collector_pg_wal_archive_status{value="ready"} 0

#  HELP cnpg_collector_replica_mode 1 if the cluster is in replica mode, 0 otherwise

#  TYPE cnpg_collector_replica_mode gauge

cnpg_collector_replica_mode 0

#  HELP cnpg_collector_sync_replicas Number of requested synchronous replicas␣
↪(synchronous_standby_names)

#  TYPE cnpg_collector_sync_replicas gauge

cnpg_collector_sync_replicas{value="expected"} 0
cnpg_collector_sync_replicas{value="max"} 0
cnpg_collector_sync_replicas{value="min"} 0
cnpg_collector_sync_replicas{value="observed"} 0

#  HELP cnpg_collector_up 1 if PostgreSQL is up, 0 otherwise.

#  TYPE cnpg_collector_up gauge

cnpg_collector_up{cluster="cluster-example"} 1

#  HELP cnpg_collector_postgres_version Postgres version
```

(continues on next page)

(continued from previous page)

```
#  TYPE cnpg_collector_postgres_version gauge

cnpg_collector_postgres_version{cluster="cluster-example",full="16.2"} 16.2

#  HELP cnpg_collector_last_failed_backup_timestamp The last failed backup as a unix␣
↪timestamp

#  TYPE cnpg_collector_last_failed_backup_timestamp gauge

cnpg_collector_last_failed_backup_timestamp 0

#  HELP cnpg_collector_last_available_backup_timestamp The last available backup as a␣
↪unix timestamp

#  TYPE cnpg_collector_last_available_backup_timestamp gauge

cnpg_collector_last_available_backup_timestamp 1.63238406e+09

#  HELP cnpg_collector_first_recoverability_point The first point of recoverability for␣
↪the cluster as a unix timestamp

#  TYPE cnpg_collector_first_recoverability_point gauge

cnpg_collector_first_recoverability_point 1.63238406e+09

#  HELP cnpg_collector_lo_pages Estimated number of pages in the pg_largeobject table

#  TYPE cnpg_collector_lo_pages gauge

cnpg_collector_lo_pages{datname="app"} 0
cnpg_collector_lo_pages{datname="postgres"} 78

#  HELP cnpg_collector_wal_buffers_full Number of times WAL data was written to disk␣
↪because WAL buffers became full. Only available on PG 14+

#  TYPE cnpg_collector_wal_buffers_full gauge

cnpg_collector_wal_buffers_full{stats_reset="2023-06-19T10:51:27.473259Z"} 6472

#  HELP cnpg_collector_wal_bytes Total amount of WAL generated in bytes. Only available␣
↪on PG 14+

#  TYPE cnpg_collector_wal_bytes gauge

cnpg_collector_wal_bytes{stats_reset="2023-06-19T10:51:27.473259Z"} 1.0035147e+07

#  HELP cnpg_collector_wal_fpi Total number of WAL full page images generated. Only␣
↪available on PG 14+

#  TYPE cnpg_collector_wal_fpi gauge

cnpg_collector_wal_fpi{stats_reset="2023-06-19T10:51:27.473259Z"} 1474
```

(continues on next page)

(continued from previous page)

```
# HELP cnpg_collector_wal_records Total number of WAL records generated. Only available␣
↪on PG 14+

# TYPE cnpg_collector_wal_records gauge

cnpg_collector_wal_records{stats_reset="2023-06-19T10:51:27.473259Z"} 26178

# HELP cnpg_collector_wal_sync Number of times WAL files were synced to disk via issue_
↪xlog_fsync request (if fsync is on and wal_sync_method is either fdatasync, fsync or␣
↪fsync_writethrough, otherwise zero). Only available on PG 14+

# TYPE cnpg_collector_wal_sync gauge

cnpg_collector_wal_sync{stats_reset="2023-06-19T10:51:27.473259Z"} 37

# HELP cnpg_collector_wal_sync_time Total amount of time spent syncing WAL files to␣
↪disk via issue_xlog_fsync request, in milliseconds (if track_wal_io_timing is enabled,␣
↪fsync is on, and wal_sync_method is either fdatasync, fsync or fsync_writethrough,␣
↪otherwise zero). Only available on PG 14+

# TYPE cnpg_collector_wal_sync_time gauge

cnpg_collector_wal_sync_time{stats_reset="2023-06-19T10:51:27.473259Z"} 0

# HELP cnpg_collector_wal_write Number of times WAL buffers were written out to disk␣
↪via XLogWrite request. Only available on PG 14+

# TYPE cnpg_collector_wal_write gauge

cnpg_collector_wal_write{stats_reset="2023-06-19T10:51:27.473259Z"} 7243

# HELP cnpg_collector_wal_write_time Total amount of time spent writing WAL buffers to␣
↪disk via XLogWrite request, in milliseconds (if track_wal_io_timing is enabled,␣
↪otherwise zero). This includes the sync time when wal_sync_method is either open_
↪datasync or open_sync. Only available on PG 14+

# TYPE cnpg_collector_wal_write_time gauge

cnpg_collector_wal_write_time{stats_reset="2023-06-19T10:51:27.473259Z"} 0

# HELP cnpg_last_error 1 if the last collection ended with error, 0 otherwise.

# TYPE cnpg_last_error gauge

cnpg_last_error 0

# HELP go_gc_duration_seconds A summary of the pause duration of garbage collection␣
↪cycles.

# TYPE go_gc_duration_seconds summary
```

(continues on next page)

(continued from previous page)

```
go_gc_duration_seconds{quantile="0"} 5.01e-05
go_gc_duration_seconds{quantile="0.25"} 7.27e-05
go_gc_duration_seconds{quantile="0.5"} 0.0001748
go_gc_duration_seconds{quantile="0.75"} 0.0002959
go_gc_duration_seconds{quantile="1"} 0.0012776
go_gc_duration_seconds_sum 0.0035741
go_gc_duration_seconds_count 13

#  HELP go_goroutines Number of goroutines that currently exist.

#  TYPE go_goroutines gauge

go_goroutines 25

#  HELP go_info Information about the Go environment.

#  TYPE go_info gauge

go_info{version="go1.20.5"} 1

#  HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.

#  TYPE go_memstats_alloc_bytes gauge

go_memstats_alloc_bytes 4.493744e+06

#  HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.

#  TYPE go_memstats_alloc_bytes_total counter

go_memstats_alloc_bytes_total 2.1698216e+07

#  HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket␣
↪hash table.

#  TYPE go_memstats_buck_hash_sys_bytes gauge

go_memstats_buck_hash_sys_bytes 1.456234e+06

#  HELP go_memstats_frees_total Total number of frees.

#  TYPE go_memstats_frees_total counter

go_memstats_frees_total 172118

#  HELP go_memstats_gc_cpu_fraction The fraction of this programs available CPU time␣
↪used by the GC since the program started.

#  TYPE go_memstats_gc_cpu_fraction gauge

go_memstats_gc_cpu_fraction 1.0749468700447189e-05
```

(continues on next page)

(continued from previous page)

```
#  HELP go_memstats_gc_sys_bytes Number of bytes used for garbage collection system␣
↪metadata.

#  TYPE go_memstats_gc_sys_bytes gauge

go_memstats_gc_sys_bytes 5.530048e+06

#  HELP go_memstats_heap_alloc_bytes Number of heap bytes allocated and still in use.

#  TYPE go_memstats_heap_alloc_bytes gauge

go_memstats_heap_alloc_bytes 4.493744e+06

#  HELP go_memstats_heap_idle_bytes Number of heap bytes waiting to be used.

#  TYPE go_memstats_heap_idle_bytes gauge

go_memstats_heap_idle_bytes 5.8236928e+07

#  HELP go_memstats_heap_inuse_bytes Number of heap bytes that are in use.

#  TYPE go_memstats_heap_inuse_bytes gauge

go_memstats_heap_inuse_bytes 7.528448e+06

#  HELP go_memstats_heap_objects Number of allocated objects.

#  TYPE go_memstats_heap_objects gauge

go_memstats_heap_objects 26306

#  HELP go_memstats_heap_released_bytes Number of heap bytes released to OS.

#  TYPE go_memstats_heap_released_bytes gauge

go_memstats_heap_released_bytes 5.7401344e+07

#  HELP go_memstats_heap_sys_bytes Number of heap bytes obtained from system.

#  TYPE go_memstats_heap_sys_bytes gauge

go_memstats_heap_sys_bytes 6.5765376e+07

#  HELP go_memstats_last_gc_time_seconds Number of seconds since 1970 of last garbage␣
↪collection.

#  TYPE go_memstats_last_gc_time_seconds gauge

go_memstats_last_gc_time_seconds 1.6311727586032727e+09

#  HELP go_memstats_lookups_total Total number of pointer lookups.
```

(continues on next page)

(continued from previous page)

```
#  TYPE go_memstats_lookups_total counter

go_memstats_lookups_total 0

#  HELP go_memstats_mallocs_total Total number of mallocs.

#  TYPE go_memstats_mallocs_total counter

go_memstats_mallocs_total 198424

#  HELP go_memstats_mcache_inuse_bytes Number of bytes in use by mcache structures.

#  TYPE go_memstats_mcache_inuse_bytes gauge

go_memstats_mcache_inuse_bytes 14400

#  HELP go_memstats_mcache_sys_bytes Number of bytes used for mcache structures obtained␣
↪from system.

#  TYPE go_memstats_mcache_sys_bytes gauge

go_memstats_mcache_sys_bytes 16384

#  HELP go_memstats_mspan_inuse_bytes Number of bytes in use by mspan structures.

#  TYPE go_memstats_mspan_inuse_bytes gauge

go_memstats_mspan_inuse_bytes 191896

#  HELP go_memstats_mspan_sys_bytes Number of bytes used for mspan structures obtained␣
↪from system.

#  TYPE go_memstats_mspan_sys_bytes gauge

go_memstats_mspan_sys_bytes 212992

#  HELP go_memstats_next_gc_bytes Number of heap bytes when next garbage collection will␣
↪take place.

#  TYPE go_memstats_next_gc_bytes gauge

go_memstats_next_gc_bytes 8.689632e+06

#  HELP go_memstats_other_sys_bytes Number of bytes used for other system allocations.

#  TYPE go_memstats_other_sys_bytes gauge

go_memstats_other_sys_bytes 2.566622e+06

#  HELP go_memstats_stack_inuse_bytes Number of bytes in use by the stack allocator.

#  TYPE go_memstats_stack_inuse_bytes gauge
```

(continues on next page)

(continued from previous page)

```
go_memstats_stack_inuse_bytes 1.343488e+06

#  HELP go_memstats_stack_sys_bytes Number of bytes obtained from system for stack␣
↪allocator.

#  TYPE go_memstats_stack_sys_bytes gauge

go_memstats_stack_sys_bytes 1.343488e+06

#  HELP go_memstats_sys_bytes Number of bytes obtained from system.

#  TYPE go_memstats_sys_bytes gauge

go_memstats_sys_bytes 7.6891144e+07

#  HELP go_threads Number of OS threads created.

#  TYPE go_threads gauge

go_threads 18
```

Note: *cnpg_collector_postgres_version* is a GaugeVec metric containing the *Major.Minor* version of PostgreSQL. The full semantic version *Major.Minor.Patch* can be found inside one of its label field named *full* .

Note: *cnpg_collector_first_recoverability_point* and *cnpg_collector_last_available_backup_timestamp* will be zero until your first backup to the object store. This is separate from the WAL archival.

User defined metrics

This feature is currently in *beta* state and the format is inspired by the queries.yaml file (release 0.12)

of the PostgreSQL Prometheus Exporter.

Custom metrics can be defined by users by referring to the created `Configmap` /`Secret` in a `Cluster` definition under the `.spec.monitoring.customQueriesConfigMap` or `customQueriesSecret` section as in the following example:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
  namespace: test
spec:
  instances: 3

  storage:
    size: 1Gi
```

(continues on next page)

(continued from previous page)

```
  monitoring:
    customQueriesConfigMap:
      - name: example-monitoring
        key: custom-queries
```

The `customQueriesConfigMap` /`customQueriesSecret` sections contain a list of `ConfigMap` /`Secret` references specifying the key in which the custom queries are defined. Take care that the referred resources have to be created **in the same namespace as the Cluster** resource.

Note: If you want ConfigMaps and Secrets to be **automatically** reloaded by instances, you can add a label with key *cnpg.io/reload* to it, otherwise you will have to reload the instances using the *kubectl cnpg reload* subcommand.

Important: When a user defined metric overwrites an already existing metric the instance manager prints a json warning log, containing the message:*Query with the same name already found. Overwriting the existing one.* and a key *queryName* containing the overwritten query name.

Example of a user defined metric

Here you can see an example of a `ConfigMap` containing a single custom query, referenced by the `Cluster` example above:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-monitoring
  namespace: test
  labels:
    cnpg.io/reload: ""
data:
  custom-queries: |
    pg_replication:
      query: "SELECT CASE WHEN NOT pg_is_in_recovery()
              THEN 0
              ELSE GREATEST (0,
                EXTRACT(EPOCH FROM (now() - pg_last_xact_replay_timestamp())))
              END AS lag,
              pg_is_in_recovery() AS in_recovery,
              EXISTS (TABLE pg_stat_wal_receiver) AS is_wal_receiver_up,
              (SELECT count(*) FROM pg_stat_replication) AS streaming_replicas"

      metrics:
        - lag:
            usage: "GAUGE"
            description: "Replication lag behind primary in seconds"
        - in_recovery:
            usage: "GAUGE"
            description: "Whether the instance is in recovery"
        - is_wal_receiver_up:
```

(continues on next page)

(continued from previous page)

```
            usage: "GAUGE"
            description: "Whether the instance wal_receiver is up"
        - streaming_replicas:
            usage: "GAUGE"
            description: "Number of streaming replicas connected to the instance"
```

A list of basic monitoring queries can be found in the *default-monitoring.yaml*

that is already installed in your CloudNativePG deployment (see *Default set of metrics*).

Example of a user defined metric running on multiple databases

If the `target_databases` option lists more than one database the metric is collected from each of them.

Database auto-discovery can be enabled for a specific query by specifying a *shell-like pattern* (i.e., containing `*` , `?` or `[]`) in the list of `target_databases` . If provided, the operator will expand the list of target databases by adding all the databases returned by the execution of `SELECT datname FROM pg_database WHERE datallowconn AND NOT datistemplate` and matching the pattern according to path.Match() rules.

Note: The * character has a special meaning in yaml, so you need to quote (`"*"`) the *target_databases* value when it includes such a pattern.

It is recommended that you always include the name of the database in the returned labels, for example using the `current_database()` function as in the following example:

```
some_query: |
  query: |
    SELECT
     current_database() as datname,
     count(*) as rows
    FROM some_table
  metrics:
    - datname:
        usage: "LABEL"
        description: "Name of current database"
    - rows:
        usage: "GAUGE"
        description: "number of rows"
  target_databases:
    - albert
    - bb
    - freddie
```

This will produce in the following metric being exposed:

```
cnpg_some_query_rows{datname="albert"} 2
cnpg_some_query_rows{datname="bb"} 5
cnpg_some_query_rows{datname="freddie"} 10
```

Here is an example of a query with auto-discovery enabled which also runs on the `template1` database (otherwise not returned by the aforementioned query):

```
some_query: |
  query: |
    SELECT
     current_database() as datname,
     count(*) as rows
    FROM some_table
  metrics:
    - datname:
        usage: "LABEL"
        description: "Name of current database"
    - rows:
        usage: "GAUGE"
        description: "number of rows"
  target_databases:
    - "*"
    - "template1"
```

The above example will produce the following metrics (provided the databases exist):

```
cnpg_some_query_rows{datname="albert"} 2
cnpg_some_query_rows{datname="bb"} 5
cnpg_some_query_rows{datname="freddie"} 10
cnpg_some_query_rows{datname="template1"} 7
cnpg_some_query_rows{datname="postgres"} 42
```

Structure of a user defined metric

Every custom query has the following basic structure:

```
<MetricName>:
      query: "<SQLQuery>"
      metrics:
        - <ColumnName>:
            usage: "<MetricType>"
            description: "<MetricDescription>"
```

Here is a short description of all the available fields:

- `<MetricName>` : the name of the Prometheus metric
  - `query` : the SQL query to run on the target database to generate the metrics
  - `primary` : whether to run the query only on the primary instance
  - `master` : same as `primary` (for compatibility with the Prometheus PostgreSQL exporter' s syntax - deprecated)
  - `runonserver` : a semantic version range to limit the versions of PostgreSQL the query should run on (e.g. `">=11.0.0"` or `">=12.0.0 <=15.0.0"`)
  - `target_databases` : a list of databases to run the `query` against, or a shell-like pattern

to enable auto discovery. Overwrites the default database if provided. - `metrics` : section containing a list of all exported columns, defined as follows: - `<ColumnName>` : the name of the column returned by the query - `usage` : one

of the values described below - `description` : the metric' s description - `metrics_mapping` : the optional column mapping when `usage` is set to `MAPPEDMETRIC`

The possible values for `usage` are:

Column Usage Label	Description
DISCARD	this column should be ignored
LABEL	use this column as a label
COUNTER	use this column as a counter
GAUGE	use this column as a gauge
MAPPEDMETRIC	use this column with the supplied mapping of text values
DURATION	use this column as a text duration (in milliseconds)
HISTOGRAM	use this column as a histogram

Please visit the Metric Types

from the Prometheus documentation for more information.

Output of a user defined metric

Custom defined metrics are returned by the Prometheus exporter endpoint (`:9187/metrics`) with the following format:

```
cnpg_<MetricName>_<ColumnName>{<LabelColumnName>=<LabelColumnValue> ... } <ColumnValue>
```

Note: *LabelColumnName* are metrics with *usage* set to *LABEL* and their *Value*

Considering the `pg_replication` example above, the exporter' s endpoint would return the following output when invoked:

```
#  HELP cnpg_pg_replication_in_recovery Whether the instance is in recovery

#  TYPE cnpg_pg_replication_in_recovery gauge

cnpg_pg_replication_in_recovery 0
#  HELP cnpg_pg_replication_lag Replication lag behind primary in seconds

#  TYPE cnpg_pg_replication_lag gauge

cnpg_pg_replication_lag 0
#  HELP cnpg_pg_replication_streaming_replicas Number of streaming replicas connected to␣
↪the instance

#  TYPE cnpg_pg_replication_streaming_replicas gauge

cnpg_pg_replication_streaming_replicas 2
#  HELP cnpg_pg_replication_is_wal_receiver_up Whether the instance wal_receiver is up

#  TYPE cnpg_pg_replication_is_wal_receiver_up gauge

cnpg_pg_replication_is_wal_receiver_up 0
```

Default set of metrics

The operator can be configured to automatically inject in a Cluster a set of monitoring queries defined in a ConfigMap or a Secret, inside the operator' s namespace. You have to set the `MONITORING_QUERIES_CONFIGMAP` or `MONITORING_QUERIES_SECRET` key in the *Operator configuration* , respectively to the name of the ConfigMap or the Secret; the operator will then use the content of the `queries` key.

Any change to the `queries` content will be immediately reflected on all the deployed Clusters using it.

The operator installation manifests come with a predefined ConfigMap, called `cnpg-default-monitoring` , to be used by all Clusters. `MONITORING_QUERIES_CONFIGMAP` is by default set to `cnpg-default-monitoring` in the operator configuration.

If you want to disable the default set of metrics, you can:

- disable it at operator level: set the `MONITORING_QUERIES_CONFIGMAP` /`MONITORING_QUERIES_SECRET` key to `""` (empty string), in the operator ConfigMap. Changes to operator ConfigMap require an operator restart.
- disable it for a specific Cluster: set `.spec.monitoring.disableDefaultQueries` to `true` in the Cluster.

Important: The ConfigMap or Secret specified via *MONITORING_QUERIES_CONFIGMAP* /*MONITORING_QUERIES_SECRET* will always be copied to the Cluster's namespace with a fixed name: *cnpg-default-monitoring* . So that, if you intend to have default metrics, you should not create a ConfigMap with this name in the cluster's namespace.

Differences with the Prometheus Postgres exporter

CloudNativePG is inspired by the PostgreSQL Prometheus Exporter, but presents some differences. In particular, the `cache_seconds` field is not implemented in CloudNativePG' s exporter.

Monitoring the operator

The operator internally exposes Prometheus metrics via HTTP on port 8080, named `metrics` .

Note: You can inspect the exported metrics by following the instructions in the *How to inspect the exported metrics*

section below.

Currently, the operator exposes default `kubebuilder` metrics, see
kubebuilder documentation for more details.

Prometheus Operator example

The operator deployment can be monitored using the Prometheus Operator by defining the following PodMonitor

resource:

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cnpg-controller-manager
```

(continues on next page)

(continued from previous page)

```
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: cloudnative-pg
  podMetricsEndpoints:
    - port: metrics
```

How to inspect the exported metrics

In this section we provide some basic instructions on how to inspect the metrics exported by a specific PostgreSQL instance manager (primary or replica) or the operator, using a temporary pod running `curl` in the same namespace.

Note: In the example below we assume we are working in the default namespace, alongside with the PostgreSQL cluster. Please feel free to adapt this example to your use case, by applying basic Kubernetes knowledge.

Create the `curl.yaml` file with this content:

```
apiVersion: v1
kind: Pod
metadata:
  name: curl
spec:
  containers:
  - name: curl
    image: curlimages/curl:8.2.1
    command: [sleep, 3600]
```

Then create the pod:

```
kubectl apply -f curl.yaml
```

In case you want to inspect the metrics exported by an instance, you need to connect to port 9187 of the target pod. This is the generic command to be run (make sure you use the correct IP for the pod):

```
kubectl exec -ti curl -- curl -s <pod_ip>:9187/metrics
```

For example, if your PostgreSQL cluster is called `cluster-example` and you want to retrieve the exported metrics of the first pod in the cluster, you can run the following command to programmatically get the IP of that pod:

```
POD_IP=$(kubectl get pod cluster-example-1 --template {{.status.podIP}})
```

And then run:

```
kubectl exec -ti curl -- curl -s ${POD_IP}:9187/metrics
```

In case you want to access the metrics of the operator, you need to point to the pod where the operator is running, and use TCP port 8080 as target.

At the end of the inspection, please make sure you delete the `curl` pod:

```
kubectl delete -f curl.yaml
```

Auxiliary resources

Important: These resources are provided for illustration and experimentation, and do not represent any kind of recommendation for your production system

In the *doc/src/samples/monitoring/*

directory you will find a series of sample files for observability. Please refer to *Part 4 of the quickstart*

section for context:

- `kube-stack-config.yaml` : a configuration file for the kube-stack helm chart installation. It ensures that Prometheus listens for all PodMonitor resources.
- `prometheusrule.yaml` : a `PrometheusRule` with alerts for CloudNativePG. NOTE: this does not include inter-operation with notification services. Please refer to the Prometheus documentation .
- `podmonitor.yaml` : a `PodMonitor` for the CloudNativePG Operator deployment.

In addition, we provide the "raw" sources for the Prometheus alert rules in the `alerts.yaml` file.

The Grafana dashboard has a dedicated repository now.

Note that, for the configuration of `kube-prometheus-stack` , other fields and settings are available over what we provide in `kube-stack-config.yaml` .

You can execute `helm show values prometheus-community/kube-prometheus-stack` to view them. For further information, please refer to the kube-prometheus-stack

page.

1.4.31 Logging

The operator is designed to log in JSON format directly to standard output, including PostgreSQL logs.

Each log entry has the following fields:

- `level` – Log level (`info` , `notice` , ⋯).
- `ts` – The timestamp (epoch with microseconds).
- `logger` – The type of the record (for example, `postgres` or `pg_controldata`).
- `msg` – The actual message or the keyword `record` in case the message is parsed in JSON format.
- `record` – The actual record with structure that varies depending on the `logger` type.
- `logging_podName` – The pod where the log was created.

Warning: Long-term storage and management of logs is outside the operator's purview, and needs to be provided at the level of the Kubernetes installation. See the *Kubernetes Logging Architecture*

documentation.

Note: If your log ingestion system requires it, you can rename the *level* and *ts* field names using the *log-field-level* and *log-field-timestamp* flags of the operator controller. Edit the *Deployment* definition of the *cloudnative-pg* operator.

Operator log

You can specify a log level in the cluster spec with the option `logLevel` . You can set it to `error` , `warning` , `info` (default), `debug` , or `trace` .

Currently, you can set the log level only when an instance starts. You can't change it at runtime. If you change the value in the cluster spec after the cluster was started, it takes effect only in the new pods and not the old ones.

PostgreSQL log

Each entry in the PostgreSQL log is a JSON object having the `logger` key set to `postgres` and the structure described in the following example:

```
{
  "level": "info",
  "ts": 1619781249.7188137,
  "logger": "postgres",
  "msg": "record",
  "record": {
    "log_time": "2021-04-30 11:14:09.718 UTC",
    "user_name": "",
    "database_name": "",
    "process_id": "25",
    "connection_from": "",
    "session_id": "608be681.19",
    "session_line_num": "1",
    "command_tag": "",
    "session_start_time": "2021-04-30 11:14:09 UTC",
    "virtual_transaction_id": "",
    "transaction_id": "0",
    "error_severity": "LOG",
    "sql_state_code": "00000",
    "message": "database system was interrupted; last known up at 2021-04-30 11:14:07 UTC
↪",
    "detail": "",
    "hint": "",
    "internal_query": "",
    "internal_query_pos": "",
    "context": "",
    "query": "",
    "query_pos": "",
    "location": "",
    "application_name": "",
    "backend_type": "startup"
  },
  "logging_pod": "cluster-example-1",
}
```

Internally, the operator relies on the PostgreSQL CSV log format. See the PostgreSQL documentation for more information about the CSV log format.

PGAudit logs

CloudNativePG has transparent and native support for PGAudit on PostgreSQL clusters.

To enable this support, add the required `pgaudit` parameters to the `postgresql` section in the configuration of the cluster.

Important: You need to add the PGAudit library to *shared_preload_libraries* . CloudNativePG adds the library based on the presence of *pgaudit.** parameters in the postgresql configuration. The operator detects and manages the addition and removal of the library from *shared_preload_libraries* .

The operator also takes care of creating and removing the extension from all the available databases in the cluster.

Important: CloudNativePG runs the *CREATE EXTENSION* and *DROP EXTENSION* commands in all databases in the cluster that accept connections.

This example shows a PostgreSQL 13 `Cluster` deployment that results in `pgaudit` being enabled with the requested configuration:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:13

  postgresql:
    parameters:
      "pgaudit.log": "all, -misc"
      "pgaudit.log_catalog": "off"
      "pgaudit.log_parameter": "on"
      "pgaudit.log_relation": "on"

  storage:
    size: 1Gi
```

The audit CSV logs entries returned by PGAudit are then parsed and routed to stdout in JSON format, similarly to all the remaining logs:

- `.logger` is set to `pgaudit` .
- `.msg` is set to `record` .
- `.record` contains the whole parsed record as a JSON object. This is similar to `logging_collector` logs, except for `.record.audit` , which contains the PGAudit CSV message formatted as a JSON object.

This example shows sample log entries:

```
{
  "level": "info",
  "ts": 1627394507.8814096,
  "logger": "pgaudit",
  "msg": "record",
```

(continues on next page)

(continued from previous page)

```
  "record": {
    "log_time": "2021-07-27 14:01:47.881 UTC",
    "user_name": "postgres",
    "database_name": "postgres",
    "process_id": "203",
    "connection_from": "[local]",
    "session_id": "610011cb.cb",
    "session_line_num": "1",
    "command_tag": "SELECT",
    "session_start_time": "2021-07-27 14:01:47 UTC",
    "virtual_transaction_id": "3/336",
    "transaction_id": "0",
    "error_severity": "LOG",
    "sql_state_code": "00000",
    "backend_type": "client backend",
    "audit": {
      "audit_type": "SESSION",
      "statement_id": "1",
      "substatement_id": "1",
      "class": "READ",
      "command": "SELECT FOR KEY SHARE",
      "statement": "SELECT pg_current_wal_lsn()",
      "parameter": "<none>"
    }
  },
  "logging_pod": "cluster-example-1",
}
```

See the PGAudit documentation for more details about each field in a record.

Other logs

All logs that are produced by the operator and its instances are in JSON format, with `logger` set according to the process that produced them. Therefore, all the possible `logger` values are the following:

- `barman-cloud-wal-archive` : from `barman-cloud-wal-archive` directly
- `barman-cloud-wal-restore` : from `barman-cloud-wal-restore` directly
- `initdb` : from running `initdb`
- `pg_basebackup` : from running `pg_basebackup`
- `pg_controldata` : from running `pg_controldata`
- `pg_ctl` : from running any `pg_ctl` subcommand
- `pg_rewind` : from running `pg_rewind`
- `pgaudit` : from PGAudit extension
- `postgres` : from the `postgres` instance (having `msg` different than `record`)
- `wal-archive` : from the `wal-archive` subcommand of the instance manager
- `wal-restore` : from the `wal-restore` subcommand of the instance manager

Except for `postgres` , which has the aforementioned structures, all other possible values have `msg` set to the escaped message that' s logged.

1.4.32 Certificates

CloudNativePG was designed to natively support TLS certificates. To set up a cluster, the operator requires:

- A server certification authority (CA) certificate
- A server TLS certificate signed by the server CA
- A client CA certificate
- A streaming replication client certificate generated by the client CA

Note: You can find all the secrets used by the cluster and their expiration dates in the cluster's status.

CloudNativePG is very flexible when it comes to TLS certificates. It primarily operates in two modes:

1. **Operator managed** – Certificates are internally managed by the operator in a fully automated way and signed using a CA created by CloudNativePG.
2. ***User provided*** – Certificates are generated outside the operator and imported in the cluster definition as secrets. CloudNativePG integrates itself with cert-manager

(See *Cert-manager example* .)

You can also choose a hybrid approach, where only part of the certificates is generated outside CNPG.

Operator-managed mode

By default, the operator generates a single CA and uses it for both client and server certificates, which are then managed and renewed automatically.

Server certificates

Server CA secret

The operator generates a self-signed CA and stores it in a generic secret containing the following keys:

- `ca.crt` – CA certificate used to validate the server certificate, used as `sslrootcert` in clients' connection strings.
- `ca.key` – The key used to sign the server SSL certificate automatically.

Server TLS secret

The operator uses the generated self-signed CA to sign a server TLS certificate. It' s stored in a secret of type `kubernetes.io/tls` and configured to be used as `ssl_cert_file` and `ssl_key_file` by the instances. This approach enables clients to verify their identity and connect securely.

Server alternative DNS names

In addition to the default ones, you can specify DNS server alternative names as part of the generated server TLS secret.

Client certificates

Client CA secret

By default, the same self-signed CA as the server CA is used. The public part is passed as `ssl_ca_file` to all the instances so it can verify client certificates it signed. The private key is stored in the same secret and used to sign client certificates generated by the `kubectl cnpg` plugin.

Client `streaming_replica` ` certificate

The operator uses the generated self-signed CA to sign a client certificate for the user `streaming_replica` , storing it in a secret of type `kubernetes.io/tls` . To allow secure connection to the primary instance, this certificate is passed as `sslcert` and `sslkey` in the replicas' connection strings.

User-provided certificates mode

Server certificates

If required, you can also provide the two server certificates, generating them using a separate component such as cert-manager . To use a custom server TLS certificate for a cluster, you must specify the following parameters:

- `serverTLSSecret` – The name of a secret of type `kubernetes.io/tls` containing the server TLS certificate. It must contain both the standard `tls.crt` and `tls.key` keys.
- `serverCASecret` – The name of a secret containing the `ca.crt` key.

Note: The operator still creates and manages the two secrets related to client certificates.

Note: If you want ConfigMaps and secrets to be reloaded by instances, you can add a label with the key *cnpg.io/reload* to it. Otherwise you must reload the instances using the *kubectl cnpg reload* subcommand.

Example

Given the following files:

- `server-ca.crt` – The certificate of the CA that signed the server TLS certificate.
- `server.crt` – The certificate of the server TLS certificate.
- `server.key` – The private key of the server TLS certificate.

Create a secret containing the CA certificate:

```
kubectl create secret generic my-postgresql-server-ca \
  --from-file=ca.crt=./server-ca.crt
```

Create a secret with the TLS certificate:

```
kubectl create secret tls my-postgresql-server \
  --cert=./server.crt --key=./server.key
```

Create a PostgreSQL cluster referencing those secrets:

```
kubectl apply -f - <<EOF
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    serverCASecret: my-postgresql-server-ca
    serverTLSSecret: my-postgresql-server
  storage:
    storageClass: standard
    size: 1Gi
EOF
```

The new cluster uses the provided server certificates for TLS connections.

Cert-manager example

This simple example shows how to use cert-manager

to set up a self-signed CA and generate the needed TLS server certificate:

```
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: v1
kind: Secret
```

(continues on next page)

(continued from previous page)

```
metadata:
  name: my-postgres-server-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-server-cert
spec:
  secretName: my-postgres-server-cert
  usages:
    - server auth
  dnsNames:
    - cluster-example-lb.internal.mydomain.net
    - cluster-example-rw
    - cluster-example-rw.default
    - cluster-example-rw.default.svc
    - cluster-example-r
    - cluster-example-r.default
    - cluster-example-r.default.svc
    - cluster-example-ro
    - cluster-example-ro.default
    - cluster-example-ro.default.svc
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

Cert-manager creates a secret named `my-postgres-server-cert` . It contains all the needed files and can be referenced from a cluster as follows:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    serverTLSSecret: my-postgres-server-cert
    serverCASecret: my-postgres-server-cert
  storage:
    size: 1Gi
```

You can find a complete example using cert-manager to manage both server and client CA and certificates in the deployment manifest.

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
```

(continues on next page)

(continued from previous page)

```
  selfSigned: {}
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: server-ca
spec:
  isCA: true
  commonName: my-selfsigned-server-ca
  secretName: server-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: server-ca-issuer
spec:
  ca:
    secretName: server-ca-key-pair
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-server-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-server-cert
spec:
  secretName: my-postgres-server-cert
  usages:
    - server auth
  dnsNames:
    - cluster-example-lb.internal.mydomain.net
    - cluster-example-rw
    - cluster-example-rw.default
    - cluster-example-rw.default.svc
    - cluster-example-r
    - cluster-example-r.default
    - cluster-example-r.default.svc
    - cluster-example-ro
    - cluster-example-ro.default
    - cluster-example-ro.default.svc
```

(continues on next page)

(continued from previous page)

```
  issuerRef:
    name: server-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: client-ca
spec:
  isCA: true
  commonName: my-selfsigned-client-ca
  secretName: client-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: client-ca-issuer
spec:
  ca:
    secretName: client-ca-key-pair
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-client-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-client-cert
spec:
  secretName: my-postgres-client-cert
  usages:
    - client auth
  commonName: streaming_replica
  issuerRef:
    name: client-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
```

(continues on next page)

(continued from previous page)

```
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    serverTLSSecret: my-postgres-server-cert
    serverCASecret: my-postgres-server-cert
    clientCASecret: my-postgres-client-cert
    replicationTLSSecret: my-postgres-client-cert
  storage:
    size: 1Gi
```

Client certificate

If required, you can also provide the two client certificates, generating them using a separate component such as cert-manager or HashiCorp vault . To use a custom CA to verify client certificates for a cluster, you must specify the following parameters:

- `replicationTLSSecret` – The name of a secret of type `kubernetes.io/tls` containing the client certificate for user `streaming_replica` . It must contain both the standard `tls.crt` and `tls.key` keys.
- `clientCASecret` – The name of a secret containing the `ca.crt` key of the CA to use to verify client certificate.

Note: The operator still creates and manages the two secrets related to server certificates.

Note: As the cluster isn't in control of the client CA secret key, you can no longer generate client certificates using *kubectl cnpg certificate* .

Note: If you want ConfigMaps and secrets to be automatically reloaded by instances, you can add a label with the key *cnpg.io/reload* to it. Otherwise, you must reload the instances using the *kubectl cnpg reload* subcommand.

Cert-manager example

This simple example shows how to use cert-manager

to set up a self-signed CA and generate the needed TLS server certificate:

```
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: v1
```

(continues on next page)

(continued from previous page)

```
kind: Secret
metadata:
  name: my-postgres-client-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-client-cert
spec:
  secretName: my-postgres-client-cert
  usages:
    - client auth
  commonName: streaming_replica
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

Cert-manager creates a secret named `my-postgres-client-cert` that contains all the needed files. You can reference it from a cluster as follows:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    clientCASecret: my-postgres-client-cert
    replicationTLSSecret: my-postgres-client-cert
  storage:
    size: 1Gi
```

You can find a complete example using cert-manager to manage both server and client CA and certificates in the deployment manifest.

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: server-ca
spec:
  isCA: true
  commonName: my-selfsigned-server-ca
```

(continues on next page)

(continued from previous page)

```
  secretName: server-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: server-ca-issuer
spec:
  ca:
    secretName: server-ca-key-pair
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-server-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-server-cert
spec:
  secretName: my-postgres-server-cert
  usages:
    - server auth
  dnsNames:
    - cluster-example-lb.internal.mydomain.net
    - cluster-example-rw
    - cluster-example-rw.default
    - cluster-example-rw.default.svc
    - cluster-example-r
    - cluster-example-r.default
    - cluster-example-r.default.svc
    - cluster-example-ro
    - cluster-example-ro.default
    - cluster-example-ro.default.svc
  issuerRef:
    name: server-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: client-ca
```

(continues on next page)

(continued from previous page)

```
spec:
  isCA: true
  commonName: my-selfsigned-client-ca
  secretName: client-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: client-ca-issuer
spec:
  ca:
    secretName: client-ca-key-pair
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-client-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-client-cert
spec:
  secretName: my-postgres-client-cert
  usages:
    - client auth
  commonName: streaming_replica
  issuerRef:
    name: client-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    serverTLSSecret: my-postgres-server-cert
    serverCASecret: my-postgres-server-cert
    clientCASecret: my-postgres-client-cert
    replicationTLSSecret: my-postgres-client-cert
```

(continues on next page)

(continued from previous page)

```
  storage:
    size: 1Gi
```

1.4.33 Client TLS/SSL connections

See also:

See *Certificates*

for more details on how CloudNativePG supports TLS certificates.

The CloudNativePG operator was designed to work with TLS/SSL for both encryption in transit and authentication on the server and client sides. Clusters created using the CNPG operator come with a certification authority (CA) to create and sign TLS client certificates. Using the cnpg plugin for kubectl, you can issue a new TLS client certificate for authenticating a user instead of using passwords.

These instructions for authenticating using TLS/SSL certificates assume you installed a cluster using the manifest.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  postgresql:
    pg_hba:
      - hostssl app all all cert

  storage:
    size: 1Gi
```

According to the convention-over-configuration paradigm, that file creates an `app` database that's owned by a user called app. (You can change this convention by way of the `initdb` configuration in the `bootstrap` section.)

Issuing a new certificate

See also:

See the *Certificates in the CloudNativePG plugin*

content for details on how to use the plugin for kubectl.

You can create a certificate for the app user in the `cluster-example` PostgreSQL cluster as follows:

```
kubectl cnpg certificate cluster-app \
  --cnpg-cluster cluster-example \
  --cnpg-user app
```

You can now verify the certificate:

```
kubectl get secret cluster-app \
  -o jsonpath="{.data[tls\.crt]}" \
  | base64 -d | openssl x509 -text -noout \
  | head -n 11
```

Output:

```
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      5d:e1:72:8a:39:9f:ce:51:19:9d:21:ff:1e:4b:24:5d
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: OU = default, CN = cluster-example
    Validity
      Not Before: Mar 22 10:22:14 2021 GMT
      Not After : Mar 22 10:22:14 2022 GMT
    Subject: CN = app
```

As you can see, TLS client certificates by default are created with 90 days of validity, and with a simple CN that corresponds to the username in PostgreSQL. You can specify the validity and threshold values using the `EXPIRE_CHECK_THRESHOLD` and `CERTIFICATE_DURATION` parameters. This is necessary to leverage the `cert` authentication method for `hostssl` entries in `pg_hba.conf` .

Testing the connection via a TLS certificate

Next, test this client certificate by configuring a demo client application that connects to your CloudNativePG cluster.

The following manifest, called `cert-test.yaml` , creates a demo pod with a test application in the same namespace where your database cluster is running:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cert-test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webtest
  template:
    metadata:
      labels:
        app: webtest
    spec:
      containers:
        - image: ghcr.io/cloudnative-pg/webtest:1.6.0
          name: cert-test
          volumeMounts:
            - name: secret-volume-root-ca
              mountPath: /etc/secrets/ca
            - name: secret-volume-app
              mountPath: /etc/secrets/app
          ports:
            - containerPort: 8080
          env:
            - name: DATABASE_URL
              value: >
```

(continues on next page)

(continued from previous page)

```
                sslkey=/etc/secrets/app/tls.key
                sslcert=/etc/secrets/app/tls.crt
                sslrootcert=/etc/secrets/ca/ca.crt
                host=cluster-example-rw.default.svc
                dbname=app
                user=app
                sslmode=verify-full
            - name: SQL_QUERY
              value: SELECT 1
          readinessProbe:
            httpGet:
              port: 8080
              path: /tx
      volumes:
        - name: secret-volume-root-ca
          secret:
            secretName: cluster-example-ca
            defaultMode: 0600
        - name: secret-volume-app
          secret:
            secretName: cluster-app
            defaultMode: 0600
```

This pod mounts secrets managed by the CloudNativePG operator, including:

- `sslcert` – The TLS client public certificate.
- `sslkey` – The TLS client certificate private key.
- `sslrootcert` – The TLS CA certificate that signed the certificate on the server to use to verify the identity of the instances.

They're used to create the default resources that psql (and other libpq-based applications, like pgbench) requires to establish a TLS-encrypted connection to the Postgres database.

By default, psql searches for certificates in the `~/.postgresql` directory of the current user, but you can use the `sslkey` , `sslcert` , and `sslrootcert` options to point libpq to the actual location of the cryptographic material. The content of these files is gathered from the secrets that were previously created by using the cnpg plugin for kubectl.

Deploy the application:

```
kubectl create -f cert-test.yaml
```

Then use the created pod as the PostgreSQL client to validate the SSL connection and authentication using the TLS certificates you just created.

A readiness probe was configured to ensure that the application is ready when the database server can be reached.

You can verify that the connection works. To do so, execute an interactive `bash` inside the pod's container to run psql using the necessary options. The PostgreSQL server is exposed through the read-write Kubernetes service. Point the psql command to connect to this service:

```
kubectl exec -it cert-test -- bash -c "psql
sslkey=/etc/secrets/app/tls.key sslcert=/etc/secrets/app/tls.crt
sslrootcert=/etc/secrets/ca/ca.crt host=cluster-example-rw.default.svc dbname=app
user=app sslmode=verify-full -c select version();"
```

Output:

```
                                        version
- ----------------------------------------------------------------------------------
- -----------------
PostgreSQL 16.2 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat
8. 3.1-5), 64-bit
(1 row)
```

About TLS protocol versions

By default, the operator sets both ssl_min_protocol_version

and ssl_max_protocol_version

to `TLSv1.3` .

Important: In PostgreSQL 11, these two GUCs don't exist. Hence, in these specific versions these values aren't set, and the default values are used.

This assumes that the PostgreSQL operand images include an OpenSSL library that supports the `TLSv1.3` version. If not, or if your client applications need a lower version number, you need to manually configure it in the PostgreSQL configuration as any other Postgres GUC.

1.4.34 Connecting from an application

Applications are supposed to work with the services created by CloudNativePG in the same Kubernetes cluster:

- `[cluster name]-rw`
- `[cluster name]-ro`
- `[cluster name]-r`

Those services are entirely managed by the Kubernetes cluster and implement a form of Virtual IP as described in the Service .

Hint: It is highly recommended using those services in your applications, and avoiding connecting directly to a specific PostgreSQL instance, as the latter can change during the cluster lifetime.

You can use these services in your applications through:

- DNS resolution
- environment variables

For the credentials to connect to PostgreSQL, you can use the secrets generated by the operator.

See also:

Please refer to the *Connection pooling* for information about how to take advantage of PgBouncer as a connection pooler, and create an access layer between your applications and the PostgreSQL clusters.

DNS resolution

You can use the Kubernetes DNS service to point to a given server. The Kubernetes DNS service is required by the operator. You can do that by using the name of the service if the application is deployed in the same namespace as the PostgreSQL cluster. In case the PostgreSQL cluster resides in a different namespace, you can use the full qualifier: `service-name.namespace-name` .

DNS is the preferred and recommended discovery method.

Environment variables

If you deploy your application in the same namespace that contains the PostgreSQL cluster, you can also use environment variables to connect to the database.

For example, suppose that your PostgreSQL cluster is called `pg-database` , you can use the following environment variables in your applications:

- `PG_DATABASE_R_SERVICE_HOST` : the IP address of the service pointing to all the PostgreSQL instances for read-only workloads
- `PG_DATABASE_RO_SERVICE_HOST` : the IP address of the service pointing to all hot-standby replicas of the cluster
- `PG_DATABASE_RW_SERVICE_HOST` : the IP address of the service pointing to the *primary* instance of the cluster

Secrets

The PostgreSQL operator will generate up to two `basic-auth` type secrets for every PostgreSQL cluster it deploys:

- `[cluster name]-app` (unless you have provided an existing secret through `.spec.bootstrap.initdb.secret.name`)
- `[cluster name]-superuser` (if `.spec.enableSuperuserAccess` is set to `true` and you have not specified a different secret using `.spec.superuserSecret`)

Each secret contain the following:

- username
- password
- hostname to the RW service
- port number
- database name
- a working .pgpass file
- uri
- jdbc-uri

The `-app` credentials are the ones that should be used by applications connecting to the PostgreSQL cluster, and correspond to the user *owning* the database.

The `-superuser` ones are supposed to be used only for administrative purposes, and correspond to the `postgres` user. Since version 1.21, superuser access over the network is disabled by default.

1.4.35 Connection pooling

CloudNativePG provides native support for connection pooling with
PgBouncer , one of the most popular open source connection poolers

for PostgreSQL, through the `Pooler` custom resource definition (CRD).

In brief, a pooler in CloudNativePG is a deployment of PgBouncer pods that sits between your applications and a PostgreSQL service, for example, the `rw` service. It creates a separate, scalable, configurable, and highly available database access layer.

Architecture

The following diagram highlights how introducing a database access layer based on PgBouncer changes the architecture of CloudNativePG. Instead of directly connecting to the PostgreSQL primary service, applications can connect to the equivalent service for PgBouncer. This ability enables reuse of existing connections for faster performance and better resource management on the PostgreSQL side.

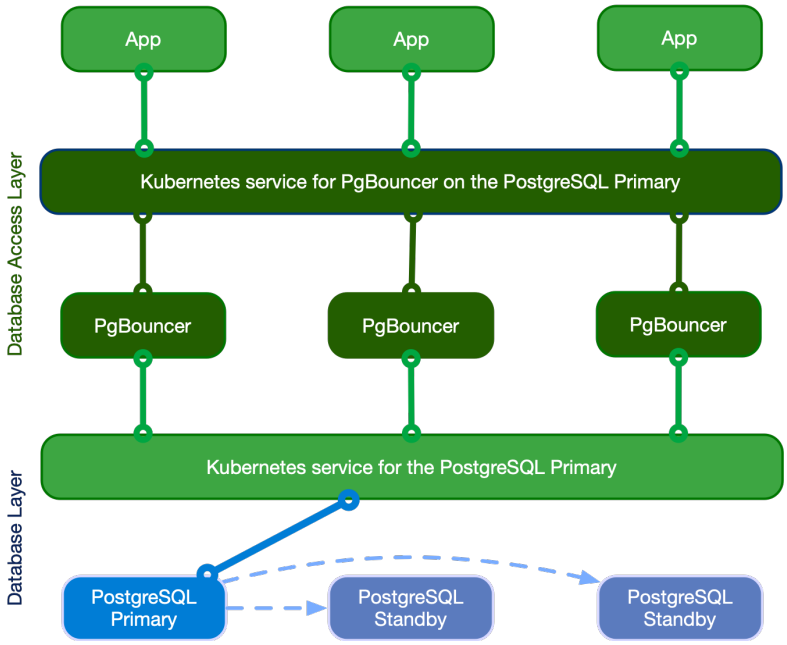


Fig. 15: Applications writing to the single primary via PgBouncer

Quick start

This example helps to show how CloudNativePG implements a PgBouncer pooler:

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example
```

(continues on next page)

(continued from previous page)

```
  instances: 3
  type: rw
  pgbouncer:
    poolMode: session
    parameters:
      max_client_conn: "1000"
      default_pool_size: "10"
```

Important: The pooler name can't be the same as any cluster name in the same namespace.

This example creates a `Pooler` resource called `pooler-example-rw` that's strictly associated with the Postgres `Cluster` resource called `cluster-example` . It points to the primary, identified by the read/write service (`rw` , therefore `cluster-example-rw`).

The `Pooler` resource must live in the same namespace as the Postgres cluster. It consists of a Kubernetes deployment of 3 pods running the latest stable image of PgBouncer , configured with the session

and accepting up to 1000 connections each. The default pool size is 10 user/database pairs toward PostgreSQL.

Important: The *Pooler* resource sets only the * fallback database in PgBouncer. This setting means that that all parameters in the connection strings passed from the client are relayed to the PostgreSQL server. For details, see ["Section [databases]" in the PgBouncer documentation](https://www.pgbouncer.org/config.html#section-databases).

CloudNativePG also creates a secret with the same name as the pooler containing the configuration files used with PgBouncer.

See also:

For details, see PgBouncerSpec {postgresql-cnpg-io-v1-PgBouncerSpec}

in the API reference.

Pooler resource lifecycle

`Pooler` resources aren't cluster-managed resources. You create poolers manually when they're needed. You can also deploy multiple poolers per PostgreSQL cluster.

What's important is that the life cycles of the `Cluster` and the `Pooler` resources are currently independent. Deleting the cluster doesn't imply the deletion of the pooler, and vice versa.

Important: Once you know how a pooler works, you have full freedom in terms of possible architectures. You can have clusters without poolers, clusters with a single pooler, or clusters with several poolers, that is, one per application.

Security

Any PgBouncer pooler is transparently integrated with CloudNativePG support for in-transit encryption by way of TLS connections, both on the client (application) and server (PostgreSQL) side of the pool.

Specifically, PgBouncer reuses the certificates of the PostgreSQL server. It also uses TLS client certificate authentication to connect to the PostgreSQL server to run the `auth_query` for clients' password authentication (see *Using authentication*).

Containers run as the pgbouncer system user, and access to the `pgbouncer` database is allowed only by way of local connections, through peer authentication.

Certificates

By default, a PgBouncer pooler uses the same certificates that are used by the cluster. However, if you provide those certificates, the pooler accepts secrets with the following formats:

1. Basic Auth
2. TLS
3. Opaque

In the Opaque case, it looks for the following specific keys that need to be used:

- tls.crt
- tls.key

So you can treat this secret as a TLS secret, and start from there.

Authentication

Password-based authentication is the only supported method for clients of PgBouncer in CloudNativePG.

Internally, the implementation relies on PgBouncer's `auth_user` and `auth_query` options. Specifically, the operator:

- Creates a standard user called `cnpg_pooler_pgbouncer` in the PostgreSQL server
- Creates the lookup function in the `postgres` database and grants execution privileges to the cnpg_pooler_pgbouncer user (PoLA)
- Issues a TLS certificate for this user
- Sets `cnpg_pooler_pgbouncer` as the `auth_user`
- Configures PgBouncer to use the TLS certificate to authenticate `cnpg_pooler_pgbouncer` against the PostgreSQL server
- Removes all the above when it detects that a cluster doesn't have any pooler associated to it

Important: If you specify your own secrets, the operator doesn't automatically integrate the pooler.

To manually integrate the pooler, if you specified your own secrets, you must run the following queries from inside your cluster.

First, you must create the role:

```
CREATE ROLE cnpg_pooler_pgbouncer WITH LOGIN;
```

Then, for each application database, grant the permission for `cnpg_pooler_pgbouncer` to connect to it:

```
GRANT CONNECT ON DATABASE { database name here } TO cnpg_pooler_pgbouncer;
```

Finally, as a *superuser* connect in each application database, and then create the authentication function inside each of the application databases:

```
CREATE OR REPLACE FUNCTION public.user_search(uname TEXT)
  RETURNS TABLE (username name, passwd text)
  LANGUAGE sql SECURITY DEFINER AS
  SELECT username, passwd FROM pg_catalog.pg_shadow WHERE username=$1;;

REVOKE ALL ON FUNCTION public.user_search(text)
  FROM public;

GRANT EXECUTE ON FUNCTION public.user_search(text)
  TO cnpg_pooler_pgbouncer;
```

Important: Given that *user_search* is a *SECURITY DEFINER* function, you need to create it through a role with *SUPERUSER* privileges, such as the *postgres* user.

Pod templates

You can take advantage of pod templates specification in the `template` section of a `Pooler` resource. For details, see PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec} in the API reference.

Using templates, you can configure pods as you like, including fine control over affinity and anti-affinity rules for pods and nodes. By default, containers use images from `ghcr.io/cloudnative-pg/pgbouncer` .

This example shows `Pooler` specifying `PodAntiAffinity` `:

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example
  instances: 3
  type: rw

  template:
    metadata:
      labels:
        app: pooler
    spec:
      containers: []
      affinity:
        podAntiAffinity:
```

(continues on next page)

(continued from previous page)

```
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - pooler
            topologyKey: "kubernetes.io/hostname"
```

Note: Explicitly set *.spec.template.spec.containers* to *[]* when not modified, as it's a required field for a *PodSpec* . If *.spec.template.spec.containers* isn't set, the Kubernetes api-server returns the following error when trying to apply the manifest:*error validating "pooler.yaml": error validating data: ValidationError(Pooler.spec.template.spec): missing required field "containers"*

This example sets resources and changes the used image:

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example
  instances: 3
  type: rw

  template:
    metadata:
      labels:
        app: pooler
    spec:
      containers:
        - name: pgbouncer
          image: my-pgbouncer:latest
          resources:
            requests:
              cpu: "0.1"
              memory: 100Mi
            limits:
              cpu: "0.5"
              memory: 500Mi
```

Service Template

Sometimes, your pooler will require some different labels, annotations, or even change the type of the service, you can achieve that by using the `serviceTemplate` field:

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example
  instances: 3
  type: rw
  serviceTemplate:
    metadata:
      labels:
        app: pooler
    spec:
      type: LoadBalancer
  pgbouncer:
    poolMode: session
    parameters:
      max_client_conn: "1000"
      default_pool_size: "10"
```

High availability (HA)

Because of Kubernetes' deployments, you can configure your pooler to run on a single instance or over multiple pods. The exposed service makes sure that your clients are randomly distributed over the available pods running PgBouncer, which then manages and reuses connections toward the underlying server (if using the `rw` service) or servers (if using the `ro` service with multiple replicas).

> **Warning:** If your infrastructure spans multiple availability zones with high latency across them, be aware of network hops. Consider, for example, the case of your application running in zone 2, connecting to PgBouncer running in zone 3, and pointing to the PostgreSQL primary in zone 1.

PgBouncer configuration options

The operator manages most of the configuration options for PgBouncer , allowing you to modify only a subset of them.

> **Warning:** You are responsible for correctly setting the value of each option, as the operator doesn't validate them.

These are the PgBouncer options you can customize, with links to the PgBouncer documentation for each parameter. Unless stated otherwise, the default values are the ones directly set by PgBouncer.

- application_name_add_host
- autodb_idle_timeout

- client_idle_timeout
- client_login_timeout
- default_pool_size
- disable_pqexec
- idle_transaction_timeout
- ignore_startup_parameters : to be appended to `extra_float_digits,options` - required by CNP
- log_connections
- log_disconnections
- log_pooler_errors
- log_stats : by default disabled (`0`), given that statistics are already collected by the Prometheus export as described in the MonitoringConfiguration {postgresql-cnpg-io-v1-MonitoringConfiguration} section below
- max_client_conn
- max_db_connections
- max_prepared_statements
- max_user_connections
- min_pool_size
- query_timeout
- query_wait_timeout
- reserve_pool_size
- reserve_pool_timeout
- server_check_delay
- server_check_query
- server_connect_timeout
- server_fast_close
- server_idle_timeout
- server_lifetime
- server_login_retry
- server_reset_query
- server_reset_query_always
- server_round_robin
- stats_period
- tcp_keepalive
- tcp_keepcnt
- tcp_keepidle
- tcp_keepintvl
- tcp_user_timeout

- verbose

Customizations of the PgBouncer configuration are written declaratively in the `.spec.pgbouncer.parameters` map.

The operator reacts to the changes in the pooler specification, and every PgBouncer instance reloads the updated configuration without disrupting the service.

> **Warning:** Every PgBouncer pod has the same configuration, aligned with the parameters in the specification. A mistake in these parameters might disrupt the operability of the whole pooler. The operator doesn't validate the value of any option.

Monitoring

The PgBouncer implementation of the `Pooler` comes with a default Prometheus exporter. It makes available several metrics having the `cnpg_pgbouncer_` prefix by running:

- `SHOW LISTS` (prefix: `cnpg_pgbouncer_lists`)
- `SHOW POOLS` (prefix: `cnpg_pgbouncer_pools`)
- `SHOW STATS` (prefix: `cnpg_pgbouncer_stats`)

Like the CloudNativePG instance, the exporter runs on port `9127` of each pod running PgBouncer and also provides metrics related to the Go runtime (with the prefix `go_*`).

> **Note:** You can inspect the exported metrics on a pod running PgBouncer. For instructions, see *How to inspect the exported metrics* . Make sure that you use the correct IP and the *9127* port.

This example shows the output for `cnpg_pgbouncer` metrics:

```
# HELP cnpg_pgbouncer_collection_duration_seconds Collection time duration in seconds

# TYPE cnpg_pgbouncer_collection_duration_seconds gauge

cnpg_pgbouncer_collection_duration_seconds{collector="Collect.up"} 0.002443168

# HELP cnpg_pgbouncer_collections_total Total number of times PostgreSQL was accessed␣
↪for metrics.

# TYPE cnpg_pgbouncer_collections_total counter

cnpg_pgbouncer_collections_total 1

# HELP cnpg_pgbouncer_last_collection_error 1 if the last collection ended with error,␣
↪0 otherwise.

# TYPE cnpg_pgbouncer_last_collection_error gauge

cnpg_pgbouncer_last_collection_error 0

# HELP cnpg_pgbouncer_lists_databases Count of databases.

# TYPE cnpg_pgbouncer_lists_databases gauge
```

(continues on next page)

(continued from previous page)

```
cnpg_pgbouncer_lists_databases 1

#  HELP cnpg_pgbouncer_lists_dns_names Count of DNS names in the cache.

#  TYPE cnpg_pgbouncer_lists_dns_names gauge

cnpg_pgbouncer_lists_dns_names 0

#  HELP cnpg_pgbouncer_lists_dns_pending Not used.

#  TYPE cnpg_pgbouncer_lists_dns_pending gauge

cnpg_pgbouncer_lists_dns_pending 0

#  HELP cnpg_pgbouncer_lists_dns_queries Count of in-flight DNS queries.

#  TYPE cnpg_pgbouncer_lists_dns_queries gauge

cnpg_pgbouncer_lists_dns_queries 0

#  HELP cnpg_pgbouncer_lists_dns_zones Count of DNS zones in the cache.

#  TYPE cnpg_pgbouncer_lists_dns_zones gauge

cnpg_pgbouncer_lists_dns_zones 0

#  HELP cnpg_pgbouncer_lists_free_clients Count of free clients.

#  TYPE cnpg_pgbouncer_lists_free_clients gauge

cnpg_pgbouncer_lists_free_clients 49

#  HELP cnpg_pgbouncer_lists_free_servers Count of free servers.

#  TYPE cnpg_pgbouncer_lists_free_servers gauge

cnpg_pgbouncer_lists_free_servers 0

#  HELP cnpg_pgbouncer_lists_login_clients Count of clients in login state.

#  TYPE cnpg_pgbouncer_lists_login_clients gauge

cnpg_pgbouncer_lists_login_clients 0

#  HELP cnpg_pgbouncer_lists_pools Count of pools.

#  TYPE cnpg_pgbouncer_lists_pools gauge

cnpg_pgbouncer_lists_pools 1

#  HELP cnpg_pgbouncer_lists_used_clients Count of used clients.
```

(continues on next page)

(continued from previous page)

```
#  TYPE cnpg_pgbouncer_lists_used_clients gauge

cnpg_pgbouncer_lists_used_clients 1

#  HELP cnpg_pgbouncer_lists_used_servers Count of used servers.

#  TYPE cnpg_pgbouncer_lists_used_servers gauge

cnpg_pgbouncer_lists_used_servers 0

#  HELP cnpg_pgbouncer_lists_users Count of users.

#  TYPE cnpg_pgbouncer_lists_users gauge

cnpg_pgbouncer_lists_users 2

#  HELP cnpg_pgbouncer_pools_cl_active Client connections that are linked to server␣
↪connection and can process queries.

#  TYPE cnpg_pgbouncer_pools_cl_active gauge

cnpg_pgbouncer_pools_cl_active{database="pgbouncer",user="pgbouncer"} 1

#  HELP cnpg_pgbouncer_pools_cl_cancel_req Client connections that have not forwarded␣
↪query cancellations to the server yet.

#  TYPE cnpg_pgbouncer_pools_cl_cancel_req gauge

cnpg_pgbouncer_pools_cl_cancel_req{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_cl_waiting Client connections that have sent queries but␣
↪have not yet got a server connection.

#  TYPE cnpg_pgbouncer_pools_cl_waiting gauge

cnpg_pgbouncer_pools_cl_waiting{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_maxwait How long the first (oldest) client in the queue has␣
↪waited, in seconds. If this starts increasing, then the current pool of servers does␣
↪not handle requests quickly enough. The reason may be either an overloaded server or␣
↪just too small of a pool_size setting.

#  TYPE cnpg_pgbouncer_pools_maxwait gauge

cnpg_pgbouncer_pools_maxwait{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_maxwait_us Microsecond part of the maximum waiting time.

#  TYPE cnpg_pgbouncer_pools_maxwait_us gauge

cnpg_pgbouncer_pools_maxwait_us{database="pgbouncer",user="pgbouncer"} 0
```

(continues on next page)

(continued from previous page)

```
# HELP cnpg_pgbouncer_pools_pool_mode The pooling mode in use. 1 for session, 2 for␣
↪transaction, 3 for statement, -1 if unknown

# TYPE cnpg_pgbouncer_pools_pool_mode gauge

cnpg_pgbouncer_pools_pool_mode{database="pgbouncer",user="pgbouncer"} 3

# HELP cnpg_pgbouncer_pools_sv_active Server connections that are linked to a client.

# TYPE cnpg_pgbouncer_pools_sv_active gauge

cnpg_pgbouncer_pools_sv_active{database="pgbouncer",user="pgbouncer"} 0

# HELP cnpg_pgbouncer_pools_sv_idle Server connections that are unused and immediately␣
↪usable for client queries.

# TYPE cnpg_pgbouncer_pools_sv_idle gauge

cnpg_pgbouncer_pools_sv_idle{database="pgbouncer",user="pgbouncer"} 0

# HELP cnpg_pgbouncer_pools_sv_login Server connections currently in the process of␣
↪logging in.

# TYPE cnpg_pgbouncer_pools_sv_login gauge

cnpg_pgbouncer_pools_sv_login{database="pgbouncer",user="pgbouncer"} 0

# HELP cnpg_pgbouncer_pools_sv_tested Server connections that are currently running␣
↪either server_reset_query or server_check_query.

# TYPE cnpg_pgbouncer_pools_sv_tested gauge

cnpg_pgbouncer_pools_sv_tested{database="pgbouncer",user="pgbouncer"} 0

# HELP cnpg_pgbouncer_pools_sv_used Server connections that have been idle for more␣
↪than server_check_delay, so they need server_check_query to run on them before they␣
↪can be used again.

# TYPE cnpg_pgbouncer_pools_sv_used gauge

cnpg_pgbouncer_pools_sv_used{database="pgbouncer",user="pgbouncer"} 0

# HELP cnpg_pgbouncer_stats_avg_query_count Average queries per second in last stat␣
↪period.

# TYPE cnpg_pgbouncer_stats_avg_query_count gauge

cnpg_pgbouncer_stats_avg_query_count{database="pgbouncer"} 1

# HELP cnpg_pgbouncer_stats_avg_query_time Average query duration, in microseconds.
```

(continues on next page)

(continued from previous page)

```
#  TYPE cnpg_pgbouncer_stats_avg_query_time gauge

cnpg_pgbouncer_stats_avg_query_time{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_avg_recv Average received (from clients) bytes per second.

#  TYPE cnpg_pgbouncer_stats_avg_recv gauge

cnpg_pgbouncer_stats_avg_recv{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_avg_sent Average sent (to clients) bytes per second.

#  TYPE cnpg_pgbouncer_stats_avg_sent gauge

cnpg_pgbouncer_stats_avg_sent{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_avg_wait_time Time spent by clients waiting for a server,␣
↪in microseconds (average per second).

#  TYPE cnpg_pgbouncer_stats_avg_wait_time gauge

cnpg_pgbouncer_stats_avg_wait_time{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_avg_xact_count Average transactions per second in last stat␣
↪period.

#  TYPE cnpg_pgbouncer_stats_avg_xact_count gauge

cnpg_pgbouncer_stats_avg_xact_count{database="pgbouncer"} 1

#  HELP cnpg_pgbouncer_stats_avg_xact_time Average transaction duration, in microseconds.

#  TYPE cnpg_pgbouncer_stats_avg_xact_time gauge

cnpg_pgbouncer_stats_avg_xact_time{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_total_query_count Total number of SQL queries pooled by␣
↪pgbouncer.

#  TYPE cnpg_pgbouncer_stats_total_query_count gauge

cnpg_pgbouncer_stats_total_query_count{database="pgbouncer"} 3

#  HELP cnpg_pgbouncer_stats_total_query_time Total number of microseconds spent by␣
↪pgbouncer when actively connected to PostgreSQL, executing queries.

#  TYPE cnpg_pgbouncer_stats_total_query_time gauge

cnpg_pgbouncer_stats_total_query_time{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_total_received Total volume in bytes of network traffic␣
↪received by pgbouncer.
```

(continues on next page)

(continued from previous page)

```
#  TYPE cnpg_pgbouncer_stats_total_received gauge

cnpg_pgbouncer_stats_total_received{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_total_sent Total volume in bytes of network traffic sent by␣
↪pgbouncer.

#  TYPE cnpg_pgbouncer_stats_total_sent gauge

cnpg_pgbouncer_stats_total_sent{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_total_wait_time Time spent by clients waiting for a server,␣
↪in microseconds.

#  TYPE cnpg_pgbouncer_stats_total_wait_time gauge

cnpg_pgbouncer_stats_total_wait_time{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_total_xact_count Total number of SQL transactions pooled by␣
↪pgbouncer.

#  TYPE cnpg_pgbouncer_stats_total_xact_count gauge

cnpg_pgbouncer_stats_total_xact_count{database="pgbouncer"} 3

#  HELP cnpg_pgbouncer_stats_total_xact_time Total number of microseconds spent by␣
↪pgbouncer when connected to PostgreSQL in a transaction, either idle in transaction or␣
↪executing queries.

#  TYPE cnpg_pgbouncer_stats_total_xact_time gauge

cnpg_pgbouncer_stats_total_xact_time{database="pgbouncer"} 0
```

As for clusters, a specific pooler can be monitored using the
Prometheus operator's resource PodMonitor . A `PodMonitor` correctly

pointing to a pooler can be created by the operator by setting `.spec.monitoring.enablePodMonitor` to `true` in the `Pooler` resource. The default is `false` .

Important: Any change to *PodMonitor* created automatically is overridden by the operator at the next reconciliation cycle. If you need to customize it, you can do so as shown in the following example.

To deploy a `PodMonitor` for a specific pooler manually, you can define it as follows and change it as needed:

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: <POOLER_NAME>
spec:
  selector:
```

(continues on next page)

(continued from previous page)

```
    matchLabels:
      cnpg.io/poolerName: <POOLER_NAME>
  podMetricsEndpoints:
  - port: metrics
```

Logging

Logs are directly sent to standard output, in JSON format, like in the following example:

```
{
  "level": "info",
  "ts": SECONDS.MICROSECONDS,
  "msg": "record",
  "pipe": "stderr",
  "record": {
    "timestamp": "YYYY-MM-DD HH:MM:SS.MS UTC",
    "pid": "<PID>",
    "level": "LOG",
    "msg": "kernel file descriptor limit: 1048576 (hard: 1048576); max_client_conn: 100,␣
↪max expected fd use: 112"
  }
}
```

Pausing connections

The `Pooler` specification allows you to take advantage of PgBouncer' s `PAUSE` and `RESUME` commands, using only declarative configuration. You can ado this using the `paused` option, which by default is set to `false` . When set to `true` , the operator internally invokes the `PAUSE` command in PgBouncer, which:

1. Closes all active connections toward the PostgreSQL server, after waiting for the queries to complete
2. Pauses any new connection coming from the client

When the `paused` option is reset to `false` , the operator invokes the `RESUME` command in PgBouncer, reopening the taps toward the PostgreSQL service defined in the `Pooler` resource.

See also:

For more information, see PAUSE .

Important: In future versions, the switchover operation will be fully integrated with the PgBouncer pooler and take advantage of the *PAUSE /RESUME* features to reduce the perceived downtime by client applications. Currently, you can achieve the same results by setting the *paused* attribute to *true* , issuing the switchover command through the Backup {postgresql-cnpg-io-v1-Backup} , and then restoring the *paused* attribute to *false* .

Limitations

Single PostgreSQL cluster

The current implementation of the pooler is designed to work as part of a specific CloudNativePG cluster (a service). It isn't currently possible to create a pooler that spans multiple clusters.

Controlled configurability

CloudNativePG transparently manages several configuration options that are used for the PgBouncer layer to communicate with PostgreSQL. Such options aren't configurable from outside and include TLS certificates, authentication settings, the `databases` section, and the `users` section. Also, considering the specific use case for the single PostgreSQL cluster, the adopted criteria is to explicitly list the options that can be configured by users.

Note: The adopted solution likely addresses the majority of use cases. It leaves room for the future implementation of a separate operator for PgBouncer to complete the gamma with more advanced and customized scenarios.

1.4.36 Replica clusters

A replica cluster is an independent CloudNativePG `Cluster` resource that has the main characteristic to be in replica from another Postgres instance, ideally also managed by CloudNativePG. Normally, a replica cluster is in another Kubernetes cluster in another region. Replica clusters can be cascading too, and they can solely rely on object stores for replication of the data from the source, as described further down.

The diagram below - taken from the *Architecture* containing more information about this capability - shows just an example of architecture that you can implement with replica clusters.

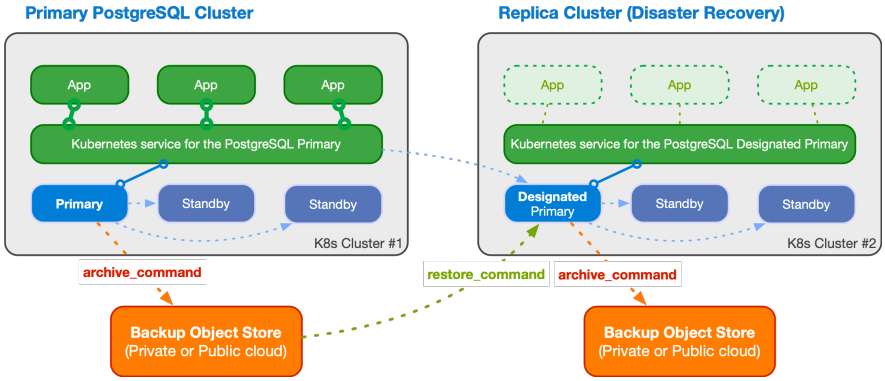


Fig. 16: An example of multi-cluster deployment with a primary and a replica cluster

Basic concepts

CloudNativePG relies on the foundations of the PostgreSQL replication framework even when a PostgreSQL cluster is created from an existing one (source) and kept synchronized through the *Replica clusters* feature. The source can be a primary cluster or another replica cluster (cascading replica cluster).

The first step is to bootstrap the replica cluster, choosing among one of the available methods:

- streaming replication, via `pg_basebackup`
- recovery from a volume snapshot
- recovery from a Barman Cloud backup in an object store

Please refer to the *Bootstrap*

for information on how to clone a PostgreSQL server using either `pg_basebackup` (streaming) or `recovery` (volume snapshot or object store).

Once the replica cluster' s base backup is available, you need to define how changes are replicated from the origin, through PostgreSQL continuous recovery. There are two options:

- use streaming replication between the replica cluster and the source (this will certainly require some administrative and security related work to be done to make sure that the network connection between the two clusters are correctly setup)
- use the WAL archive (on an object store) to fetch the WAL files that are regularly shipped from the source to the object store and pulled by `barman-cloud-wal-restore` in the replica cluster
- any of the two

All you have to do is actually define an external cluster.

If the external cluster contains a `barmanObjectStore` section:

- you' ll be able to use the WAL archive, and CloudNativePG will automatically set the `restore_command` in the designated primary instance
- you' ll be able to bootstrap the replica cluster from an object store using the `recovery` section, in case you cannot take advantage of volume snapshots

If the external cluster contains a `connectionParameters` section:

- you' ll be able to bootstrap the replica cluster via streaming replication using the `pg_basebackup` section
- CloudNativePG will automatically set the `primary_conninfo` option in the designated primary instance, so that a WAL receiver process is started to connect to the source cluster and receive data

The created replica cluster can perform backups in a reserved object store from the designated primary, enabling symmetric architectures in a distributed fashion.

You have full flexibility and freedom to decide your favorite distributed architecture for a PostgreSQL database by choosing:

- a private cloud spanning over multiple Kubernetes clusters in different data centers
- a public cloud spanning over multiple Kubernetes clusters in different regions
- a mix of the previous two (hybrid)
- a public cloud spanning over multiple Kubernetes clusters in different regions and on different Cloud Service Providers

Setting up a replica cluster

To set up a replica cluster from a source cluster, we need to create a cluster YAML file and define the following parts accordingly:

- define the `externalClusters` section in the replica cluster
- define the bootstrap part for the replica cluster. We can either bootstrap via streaming using the `pg_basebackup` section, or bootstrap from a volume snapshot or an object store using the `recovery` section
- define the continuous recovery part (`.spec.replica`) in the replica cluster. All we need to do is to enable the replica mode through option `.spec.replica.enabled` and set the `externalClusters` name in option `.spec.replica.source`

Example using pg_basebackup

This **first example** defines a replica cluster using streaming replication in both bootstrap and continuous recovery. The replica cluster connects to the source cluster using TLS authentication.

You can check the *sample YAML*

in the `samples/` subdirectory.

Note the `bootstrap` and `replica` sections pointing to the source cluster.

```yaml
bootstrap:
  pg_basebackup:
    source: cluster-example

replica:
  enabled: true
  source: cluster-example
```

In the `externalClusters` section, remember to use the right namespace for the host in the `connectionParameters` sub-section. The `-replication` and `-ca` secrets should have been copied over if necessary, in case the replica cluster is in a separate namespace.

```yaml
externalClusters:
- name: <MAIN-CLUSTER>
  connectionParameters:
    host: <MAIN-CLUSTER>-rw.<NAMESPACE>.svc
    user: streaming_replica
    sslmode: verify-full
    dbname: postgres
  sslKey:
    name: <MAIN-CLUSTER>-replication
    key: tls.key
  sslCert:
    name: <MAIN-CLUSTER>-replication
    key: tls.crt
  sslRootCert:
    name: <MAIN-CLUSTER>-ca
    key: ca.crt
```

Example using a Backup from an object store

The **second example** defines a replica cluster that bootstraps from an object store using the `recovery` section and continuous recovery using both streaming replication and the given object store. For streaming replication, the replica cluster connects to the source cluster using basic authentication.

You can check the *sample YAML*

for it in the `samples/` subdirectory.

Note the `bootstrap` and `replica` sections pointing to the source cluster.

```
bootstrap:
  recovery:
    source: cluster-example

replica:
  enabled: true
  source: cluster-example
```

In the `externalClusters` section, take care to use the right namespace in the `endpointURL` and the `connectionParameters.host` . And do ensure that the necessary secrets have been copied if necessary, and that a backup of the source cluster has been created already.

```
externalClusters:
- name: <MAIN-CLUSTER>
  barmanObjectStore:
    destinationPath: s3://backups/
    endpointURL: http://minio:9000
    s3Credentials:
      ...
  connectionParameters:
    host: <MAIN-CLUSTER>-rw.default.svc
    user: postgres
    dbname: postgres
  password:
    name: <MAIN-CLUSTER>-superuser
    key: password
```

Note: To use streaming replication between the source cluster and the replica cluster, we need to make sure there is network connectivity between the two clusters, and that all the necessary secrets which hold passwords or certificates are properly created in advance.

Example using a Volume Snapshot

If you use volume snapshots and your storage class provides snapshots cross-cluster availability, you can leverage that to bootstrap a replica cluster through a volume snapshot of the source cluster.

The **third example** defines a replica cluster that bootstraps from a volume snapshot using the `recovery` section. It uses streaming replication (via basic authentication) and the object store to fetch the WAL files.

You can check the *sample YAML*

for it in the `samples/` subdirectory.

Demoting a Primary to a Replica Cluster

CloudNativePG provides the functionality to demote a primary cluster to a replica cluster. This action is typically planned when transitioning the primary role from one data center to another. The process involves demoting the current primary cluster (e.g., cluster-eu-south) to a replica cluster and subsequently promoting the designated replica cluster (e.g., `cluster-eu-central`) to primary when fully synchronized. Provided you have defined an external cluster in the current primary `Cluster` resource that points to the replica cluster that' s been selected to become the new primary, all you need to do is to enable replica mode and define the source as follows:

```
replica:
  enabled: true
  source: cluster-eu-central
```

Promoting the designated primary in the replica cluster

To promote a replica cluster (e.g. `cluster-eu-central`) to a primary cluster and make the designated primary a real primary, all you need to do is to disable the replica mode in the replica cluster through the option `.spec.replica.enabled` :

```
replica:
  enabled: false
  source: cluster-eu-south
```

If you have first demoted the `cluster-eu-south` and waited for `cluster-eu-central` to be in sync, once `cluster-eu-central` starts as primary, the `cluster-eu-south` cluster will seamlessly start as a replica cluster, without the need of re-cloning.

If you disable replica mode without prior demotion, the replica cluster and the source cluster will become two separate clusters.

When replica mode is disabled, the **designated primary** in the replica cluster will be promoted to be that cluster' s **primary**.

You can verify the role change using the `cnpg` plugin, checking the status of the cluster which was previously the replica:

```
kubectl cnpg -n <cluster-name-space> status cluster-eu-central
```

Note: Disabling replication is an **irreversible** operation. Once replication is disabled and the designated primary is promoted to primary, the replica cluster and the source cluster become two independent clusters definitively. Ensure to follow the demotion procedure correctly to avoid unintended consequences.

Delayed replicas

In addition to standard replica clusters, our system supports the creation of **delayed replicas** through the utilization of PostgreSQL' s *recovery_min_apply_delay*

option.

Delayed replicas intentionally lag behind the primary database by a specified amount of time. This delay is configurable using the `recovery_min_apply_delay` option in PostgreSQL. The primary objective of introducing delayed replicas is to mitigate the impact of unintentional executions of SQL statements on the primary database. This is particularly useful in scenarios where an incorrect or missing `WHERE` clause is used in operations such as `UPDATE` or `DELETE` .

To introduce a delay in a replica cluster, adjust the `recovery_min_apply_delay` option. This parameter determines the time by which replicas lag behind the primary. For example:

```
# ...
postgresql:
  parameters:
    # Enforce a delay of 8 hours
    recovery_min_apply_delay = 8h
# ...
```

Monitor and adjust the delay as needed based on your recovery time objectives and the potential impact of unintended primary database operations.

The main use cases of delayed replicas can be summarized into:

1. mitigating human errors: reduce the risk of data corruption or loss resulting from unintentional SQL operations on the primary database
2. recovery time optimization: facilitate quicker recovery from unintended changes by having a delayed replica that allows you to identify and rectify issues before changes are applied to other replicas.
3. enhanced data protection: safeguard critical data by introducing a time buffer that provides an opportunity to intervene and prevent the propagation of undesirable changes.

By integrating delayed replicas into your replication strategy, you can enhance the resilience and data protection capabilities of your PostgreSQL environment. Adjust the delay duration based on your specific needs and the criticality of your data.

Important: Always measure your goals. Depending on your environment, it might be more efficient to rely on volume snapshot-based recovery for faster outcomes. Evaluate and choose the approach that best aligns with your unique requirements and infrastructure.

1.4.37 Kubernetes Upgrade and Maintenance

Maintaining an up-to-date Kubernetes cluster is crucial for ensuring optimal performance and security, particularly for self-managed clusters, especially those running on bare metal infrastructure. Regular updates help address technical debt and mitigate business risks, despite the controlled downtimes associated with temporarily removing a node from the cluster for maintenance purposes. For further insights on embracing risk in operations, refer to the Embracing Risk

chapter from the Site Reliability Engineering book.

Importance of Regular Updates

Updating Kubernetes involves planning and executing maintenance tasks, such as applying security updates to underlying Linux servers, replacing malfunctioning hardware components, or upgrading the cluster to the latest Kubernetes version. These activities are essential for maintaining a robust and secure infrastructure.

Maintenance Operations in a Cluster

Typically, maintenance operations are carried out on one node at a time, following a structured process :

1. eviction of workloads (`drain`): workloads are gracefully moved away from the node to be updated, ensuring a smooth transition.
2. performing the operation: the actual maintenance operation, such as a system update or hardware replacement, is executed.
3. rejoining the node to the cluster (`uncordon`): the updated node is reintegrated into the cluster, ready to resume its responsibilities.

This process requires either stopping workloads for the entire upgrade duration or migrating them to other nodes in the cluster.

Temporary PostgreSQL Cluster Degradation

While the standard approach ensures service reliability and leverages Kubernetes' self-healing capabilities, there are scenarios where operating with a temporarily degraded cluster may be acceptable. This is particularly relevant for PostgreSQL clusters relying on **node-local storage**, where the storage is local to the Kubernetes worker node running the PostgreSQL database. Node-local storage, or simply *local storage*, is employed to enhance performance.

Note: If your database files reside on shared storage accessible over the network, the default self-healing behavior of the operator can efficiently handle scenarios where volumes are reused by pods on different nodes after a drain operation. In such cases, you can skip the remaining sections of this document.

Pod Disruption Budgets

By default, CloudNativePG safeguards Postgres cluster operations. If a node is to be drained and contains a cluster's primary instance, a switchover happens ahead of the drain. Once the instance in the node is downgraded to replica, the draining can resume. For single-instance clusters, a switchover is not possible, so CloudNativePG will prevent draining the node where the instance is housed. Additionally, in multi-instance clusters, CloudNativePG guarantees that only one replica at a time is gracefully shut down during a drain operation.

Each PostgreSQL `Cluster` is equipped with two associated `PodDisruptionBudget` resources - you can easily confirm it with the `kubectl get pdb` command.

Our recommendation is to leave pod disruption budgets enabled for every production Postgres cluster. This can be effortlessly managed by toggling the `.spec.enablePDB` option, as detailed in the *API reference* .

PostgreSQL Clusters used for Development or Testing

For PostgreSQL clusters used for development purposes, often consisting of a single instance, it is essential to disable pod disruption budgets. Failure to do so will prevent the node hosting that cluster from being drained.

The following example illustrates how to disable pod disruption budgets for a 1-instance development cluster:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: dev
spec:
  instances: 1
  enablePDB: false

  storage:
    size: 1Gi
```

This configuration ensures smoother maintenance procedures without restrictions on draining the node during development activities.

Node Maintenance Window

Important: While CloudNativePG will continue supporting the node maintenance window, it is currently recommended to transition to direct control of pod disruption budgets, as explained in the previous section. This section is retained mainly for backward compatibility.

Prior to release 1.23, CloudNativePG had just one declarative mechanism to manage Kubernetes upgrades when dealing with local storage: you had to temporarily put the cluster in **maintenance mode** through the `nodeMaintenanceWindow` option to avoid standard self-healing procedures to kick in, while, for example, enlarging the partition on the physical node or updating the node itself.

Warning: Limit the duration of the maintenance window to the shortest amount of time possible. In this phase, some of the expected behaviors of Kubernetes are either disabled or running with some limitations, including self-healing, rolling updates, and Pod disruption budget.

The `nodeMaintenanceWindow` option of the cluster has two further settings:

`inProgress` : Boolean value that states if the maintenance window for the nodes is currently in progress or not. By default, it is set to `off` . During the maintenance window, the `reusePVC` option below is evaluated by the operator.

`reusePVC` : Boolean value that defines if an existing PVC is reused or not during the maintenance operation. By default, it is set to `on` . When **enabled**, Kubernetes waits for the node to come up again and then reuses the existing PVC; the `PodDisruptionBudget` policy is temporarily removed. When **disabled**, Kubernetes forces the recreation of the Pod on a different node with a new PVC by relying on PostgreSQL' s physical streaming replication, then destroys the old PVC together with the Pod. This scenario is generally not recommended unless the database' s size is small, and re-cloning the new PostgreSQL instance takes shorter than waiting. This behavior does **not** apply to clusters with only one instance and reusePVC disabled: see section below.

Note: When performing the *kubectl drain* command, you will need to add the *--delete-local-data* option. Don't be

afraid: it refers to another volume internally used by the operator - not the PostgreSQL data directory.

Important: *PodDisruptionBudget* management can be disabled by setting the *.spec.enablePDB* field to *false* . In that case, the operator won't create *PodDisruptionBudgets* and will delete them if they were previously created.

Single instance clusters with `reusePVC` set to `false`

Important: We recommend to always create clusters with more than one instance in order to guarantee high availability.

Deleting the only PostgreSQL instance in a single instance cluster with `reusePVC` set to `false` would imply all data being lost, therefore we prevent users from draining nodes such instances might be running on, even in maintenance mode.

However, in case maintenance is required for such a node you have two options:

1. Enable `reusePVC` , accepting the downtime
2. Replicate the instance on a different node and switch over the primary

As long as a database service downtime is acceptable for your environment, draining the node is as simple as setting the `nodeMaintenanceWindow` to `inProgress: true` and `reusePVC: true` . This will allow the instance to be deleted and recreated as soon as the original PVC is available (e.g. with node local storage, as soon as the node is back up).

Otherwise you will have to scale up the cluster, creating a new instance on a different node and promoting the new instance to primary in order to shut down the original one on the node undergoing maintenance. The only downtime in this case will be the duration of the switchover.

A possible approach could be:

1. Cordon the node on which the current instance is running.
2. Scale up the cluster to 2 instances, could take some time depending on the database size.
3. As soon as the new instance is running, the operator will automatically perform a switchover given that the current primary is running on a cordoned node.
4. Scale back down the cluster to a single instance, this will delete the old instance
5. The old primary's node can now be drained successfully, while leaving the new primary running on a new node.

1.4.38 Exposing Postgres Services

This section explains how to expose a PostgreSQL service externally, allowing access to your PostgreSQL database **from outside your Kubernetes cluster** using NGINX Ingress Controller.

If you followed the *Quickstart* , you should have by now a database that can be accessed inside the cluster via the `cluster-example-rw` (primary) and `cluster-example-r` (read-only) services in the `default` namespace. Both services use port `5432` .

Let's assume that you want to make the primary instance accessible from external accesses on port `5432` . A typical use case, when moving to a Kubernetes infrastructure, is indeed the one represented by **legacy applications** that cannot be

easily or sustainably "containerized". A sensible workaround is to allow those applications that most likely reside in a virtual machine or a physical server, to access a PostgreSQL database inside a Kubernetes cluster in the same network.

> **Warning:** Allowing access to a database from the public network could expose your database to potential attacks from malicious users. Ensure you secure your database before granting external access or that your Kubernetes cluster is only reachable from a private network.

For this example, you will use NGINX Ingress Controller , since it is maintained directly by the Kubernetes project and can be set up on every Kubernetes cluster. Many other controllers are available (see the *Kubernetes documentation*

for a comprehensive list).

We assume that:

- the NGINX Ingress controller has been deployed and works correctly
- it is possible to create a service of type `LoadBalancer` in your cluster

---

Important: Ingresses are only required to expose HTTP and HTTPS traffic. While the NGINX Ingress controller can, not all Ingress objects can expose arbitrary ports or protocols.

The first step is to create a `tcp-services` `ConfigMap` whose data field contains info on the externally exposed port and the namespace, service and port to point to internally.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: tcp-services
  namespace: ingress-nginx
data:
  5432: default/cluster-example-rw:5432
```

Then, if you've installed NGINX Ingress Controller as suggested in their documentation, you should have an `ingress-nginx` service. You'll have to add the 5432 port to the `ingress-nginx` service to expose it. The ingress will redirect incoming connections on port 5432 to your database.

```yaml
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
spec:
  type: LoadBalancer
  ports:
    - name: http
      port: 80
      targetPort: 80
      protocol: TCP
    - name: https
      port: 443
```

(continues on next page)

(continued from previous page)

```
      targetPort: 443
      protocol: TCP
    - name: postgres
      port: 5432
      targetPort: 5432
      protocol: TCP
  selector:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
```

You can use and apply it using `kubectl` .

```
- --
apiVersion: v1
kind: ConfigMap
metadata:
  name: tcp-services
  namespace: ingress-nginx
data:
  5432: default/cluster-example-rw:5432

- --
apiVersion: v1
kind: Service
metadata:
  name: ingress-nginx
  namespace: ingress-nginx
  labels:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
spec:
  type: LoadBalancer
  ports:
    - name: http
      port: 80
      targetPort: 80
      protocol: TCP
    - name: https
      port: 443
      targetPort: 443
      protocol: TCP
    - name: postgres
      port: 5432
      targetPort: 5432
      protocol: TCP
  selector:
    app.kubernetes.io/name: ingress-nginx
    app.kubernetes.io/part-of: ingress-nginx
```

Warning: If you apply this file directly, you will overwrite any previous change in your *ConfigMap* and *Service* of the Ingress

Now you will be able to reach the PostgreSQL Cluster from outside your Kubernetes cluster.

Important: Make sure you configure *pg_hba* to allow connections from the Ingress.

Testing on Minikube

On Minikube you can setup the ingress controller running:

```
minikube addons enable ingress
```

You can then patch the deployment to allow access on port 5432. Create a file called `patch.yaml` with the following content:

```
spec:
  template:
    spec:
      containers:
      - name: controller
        ports:
         - containerPort: 5432
           hostPort: 5432
```

and apply it to the `ingress-nginx-controller` deployment:

```
kubectl patch deployment ingress-nginx-controller --patch "$(cat patch.yaml)" -n ingress-
↪nginx
```

You can access the primary from your machine running:

```
psql -h $(minikube ip) -p 5432 -U postgres
```

1.4.39 Kubectl Plugin

CloudNativePG provides a plugin for `kubectl` to manage a cluster in Kubernetes.

Install

You can install the `cnpg` plugin using a variety of methods.

Note: For air-gapped systems, installation via package managers, using previously downloaded files, may be a good option.

Via the installation script

```
curl -sSfL \
  https://github.com/cloudnative-pg/cloudnative-pg/raw/main/hack/install-cnpg-plugin.sh␣
↪| \
  sudo sh -s -- -b /usr/local/bin
```

Using the Debian or RedHat packages

In the releases section of the GitHub repository , you can navigate to any release of interest (pick the same or newer release than your CloudNativePG operator), and in it you will find an **Assets** section. In that section are pre-built packages for a variety of systems. As a result, you can follow standard practices and instructions to install them in your systems.

Debian packages

For example, let' s install the 1.22.2 release of the plugin, for an Intel based 64 bit server. First, we download the right `.deb` file.

```
$ wget https://github.com/cloudnative-pg/cloudnative-pg/releases/download/v1.22.1/
↪kubectl-cnpg_1.22.2_linux_x86_64.deb
```

Then, install from the local file using `dpkg` :

```
$ dpkg -i kubectl-cnpg_1.22.2_linux_x86_64.deb
(Reading database ... 702524 files and directories currently installed.)
Preparing to unpack kubectl-cnpg_1.22.2_linux_x86_64.deb ...
Unpacking cnpg (1.22.2) over (1.22.2) ...
Setting up cnpg (1.22.2) ..
```

RPM packages

As in the example for `.deb` packages, let' s install the 1.22.2 release for an Intel 64 bit machine. Note the `--output` flag to provide a file name.

```
curl -L https://github.com/cloudnative-pg/cloudnative-pg/releases/download/v1.22.2/
↪kubectl-cnpg_1.22.2_linux_x86_64.rpm \
  --output kube-plugin.rpm
```

Then install with `yum` , and you' re ready to use:

```
$ yum --disablerepo=* localinstall kube-plugin.rpm
yum --disablerepo=* localinstall kube-plugin.rpm
Failed to set locale, defaulting to C.UTF-8
Dependencies resolved.
=================================================================================================
 Package            Architecture         Version                   Repository
      Size
=================================================================================================
```

(continues on next page)

(continued from previous page)

```
Installing:
 cnpg               x86_64               1.22.2-1                  @commandline
      17 M

Transaction Summary
=============================================================================================
Install  1 Package

Total size: 14 M
Installed size: 43 M
Is this ok [y/N]: y
```

Using Krew

If you already have Krew installed, you can simply run:

```
kubectl krew install cnpg
```

When a new version of the plugin is released, you can update the existing installation with:

```
kubectl krew update
kubectl krew upgrade cnpg
```

Supported Architectures

CloudNativePG Plugin is currently built for the following operating system and architectures:

- Linux
- amd64
- arm 5/6/7
- arm64
- s390x
- ppc64le
- macOS
- amd64
- arm64
- Windows
- 386
- amd64
- arm 5/6/7
- arm64

Configuring auto-completion

To configure auto-completion for the plugin, a helper shell script needs to be installed into your current PATH. Assuming the latter contains `/usr/local/bin` , this can be done with the following commands:

```
cat > kubectl_complete-cnpg <<EOF
# !/usr/bin/env sh

#  Call the __complete command passing it all arguments

kubectl cnpg __complete "\$@"
EOF

chmod +x kubectl_complete-cnpg

#  Important: the following command may require superuser permission

sudo mv kubectl_complete-cnpg /usr/local/bin
```

Important: The name of the script needs to be exactly the one provided since is used by the kubectl auto-complete process

Use

Once the plugin was installed and deployed, you can start using it like this:

```
kubectl cnpg <command> <args...>
```

Generation of installation manifests

The `cnpg` plugin can be used to generate the YAML manifest for the installation of the operator. This option would typically be used if you want to override some default configurations such as number of replicas, installation namespace, namespaces to watch, and so on.

For details and available options, run:

```
kubectl cnpg install generate --help
```

The main options are:

- `-n` : the namespace in which to install the operator (by default: `cnpg-system`)
- `--replicas` : number of replicas in the deployment
- `--version` : minor version of the operator to be installed, such as `1.17` . If a minor version is specified, the plugin will install the latest patch version of that minor version. If no version is supplied the plugin will install the latest `MAJOR.MINOR.PATCH` version of the operator.
- `--watch-namespace` : comma separated string containing the namespaces to watch (by default all namespaces)

An example of the `generate` command, which will generate a YAML manifest that will install the operator, is as follows:

```
kubectl cnpg install generate \
  -n king \
  --version 1.17 \
  --replicas 3 \
  --watch-namespace "albert, bb, freddie" \
  > operator.yaml
```

The flags in the above command have the following meaning:

- `-n king` install the CNPG operator into the `king` namespace
- `--version 1.17` install the latest patch version for minor version 1.17
- `--replicas 3` install the operator with 3 replicas
- `--watch-namespace "albert, bb, freddie"` have the operator watch for changes in the `albert` , `bb` and `freddie` namespaces only

Status

The `status` command provides an overview of the current status of your cluster, including:

- **general information**: name of the cluster, PostgreSQL' s system ID, number of instances, current timeline and position in the WAL
- **backup**: point of recoverability, and WAL archiving status as returned by the `pg_stat_archiver` view from the primary - or designated primary in the case of a replica cluster
- **streaming replication**: information taken directly from the `pg_stat_replication` view on the primary instance
- **instances**: information about each Postgres instance, taken directly by each instance manager; in the case of a standby, the `Current LSN` field corresponds to the latest write-ahead log location that has been replayed during recovery (replay LSN).

Important: The status information above is taken at different times and at different locations, resulting in slightly inconsistent returned values. For example, the *Current Write LSN* location in the main header, might be different from the *Current LSN* field in the instances status as it is taken at two different time intervals.

```
kubectl cnpg status sandbox
```

```
Cluster in healthy state
Name:               sandbox
Namespace:          default
System ID:          7039966298120953877
PostgreSQL Image:   ghcr.io/cloudnative-pg/postgresql:16.2
Primary instance:   sandbox-2
Instances:          3
Ready instances:    3
Current Write LSN:  3AF/EAFA6168 (Timeline: 8 - WAL File: 00000008000003AF00000075)

Continuous Backup status
First Point of Recoverability:  Not Available
Working WAL archiving:          OK
```

(continues on next page)

(continued from previous page)

```
Last Archived WAL:              00000008000003AE00000079   @   2021-12-14T10:16:29.
↪340047Z
Last Failed WAL: -

Certificates Status
Certificate Name             Expiration Date                Days Left Until Expiration
- ---------------             ---------------                --------------------------
cluster-example-ca           2022-05-05 15:02:42 +0000 UTC  87.23
cluster-example-replication  2022-05-05 15:02:42 +0000 UTC  87.23
cluster-example-server       2022-05-05 15:02:42 +0000 UTC  87.23

Streaming Replication status
Name       Sent LSN      Write LSN     Flush LSN     Replay LSN    Write Lag       ␣
↪Flush Lag        Replay Lag       State      Sync State  Sync Priority
- ---       --------      ---------     ---------     ----------    ---------        ----
↪-----        ----------       -----      ----------  -------------
sandbox-1  3AF/EB0524F0  3AF/EB011760  3AF/EAFEDE50  3AF/EAFEDE50  00:00:00.004461 ␣
↪00:00:00.007901  00:00:00.007901  streaming  quorum      1
sandbox-3  3AF/EB0524F0  3AF/EB030B00  3AF/EB030B00  3AF/EB011760  00:00:00.000977 ␣
↪00:00:00.004194  00:00:00.008252  streaming  quorum      1

Instances status
Name       Database Size  Current LSN   Replication role  Status  QoS         Manager␣
↪Version
- ---       -------------  -----------   ----------------  ------  ---         ----------
↪-----
sandbox-1  302 GB         3AF/E9FFFFE0  Standby (sync)    OK      Guaranteed  1.11.0
sandbox-2  302 GB         3AF/EAFA6168  Primary           OK      Guaranteed  1.11.0
sandbox-3  302 GB         3AF/EBAD5D18  Standby (sync)    OK      Guaranteed  1.11.0
```

You can also get a more verbose version of the status by adding `--verbose` or just `-v`

```
kubectl cnpg status sandbox --verbose
```

```
Cluster in healthy state
Name:               sandbox
Namespace:          default
System ID:          7039966298120953877
PostgreSQL Image:   ghcr.io/cloudnative-pg/postgresql:16.2
Primary instance:   sandbox-2
Instances:          3
Ready instances:    3
Current Write LSN:  3B1/61DE3158 (Timeline: 8 - WAL File: 00000008000003B100000030)

PostgreSQL Configuration
archive_command = /controller/manager wal-archive --log-destination /controller/log/
↪postgres.json %p
archive_mode = on
archive_timeout = 5min
checkpoint_completion_target = 0.9
checkpoint_timeout = 900s
cluster_name = sandbox
```

(continues on next page)

(continued from previous page)

```
dynamic_shared_memory_type = sysv
full_page_writes = on
hot_standby = true
jit = on
listen_addresses = *
log_autovacuum_min_duration = 1s
log_checkpoints = on
log_destination = csvlog
log_directory = /controller/log
log_filename = postgres
log_lock_waits = on
log_min_duration_statement = 1000
log_rotation_age = 0
log_rotation_size = 0
log_statement = ddl
log_temp_files = 1024
log_truncate_on_rotation = false
logging_collector = on
maintenance_work_mem = 2GB
max_connections = 1000
max_parallel_workers = 32
max_replication_slots = 32
max_wal_size = 15GB
max_worker_processes = 32
pg_stat_statements.max = 10000
pg_stat_statements.track = all
port = 5432
shared_buffers = 16GB
shared_memory_type = sysv
shared_preload_libraries = pg_stat_statements
ssl = on
ssl_ca_file = /controller/certificates/client-ca.crt
ssl_cert_file = /controller/certificates/server.crt
ssl_key_file = /controller/certificates/server.key
synchronous_standby_names = ANY 1 ("sandbox-1","sandbox-3")
unix_socket_directories = /controller/run
wal_keep_size = 512MB
wal_level = logical
wal_log_hints = on
cnpg.config_sha256 = 3cfa683e23fe513afaee7c97b50ce0628e0cc634bca8b096517538a9a4428efc

PostgreSQL HBA Rules

#  Grant local access

local all all peer map=local

#  Require client certificate authentication for the streaming_replica user

hostssl postgres streaming_replica all cert
hostssl replication streaming_replica all cert
hostssl all cnpg_pooler_pgbouncer all cert
```

(continues on next page)

(continued from previous page)

```
#  Otherwise use the default authentication method

host all all all scram-sha-256

Continuous Backup status
First Point of Recoverability:  Not Available
Working WAL archiving:          OK
Last Archived WAL:              00000008000003B00000001D   @   2021-12-14T10:20:42.
↪272815Z
Last Failed WAL: -

Streaming Replication status
Name       Sent LSN      Write LSN     Flush LSN     Replay LSN    Write Lag       ␣
↪Flush Lag        Replay Lag       State      Sync State  Sync Priority
- ---       --------      ---------     ---------     ----------    ---------        ----
↪-----        ----------       -----      ----------  -------------
sandbox-1  3B1/61E26448  3B1/61DF82F0  3B1/61DF82F0  3B1/61DF82F0  00:00:00.000333 ␣
↪00:00:00.000333  00:00:00.005484  streaming  quorum      1
sandbox-3  3B1/61E26448  3B1/61E26448  3B1/61DF82F0  3B1/61DF82F0  00:00:00.000756 ␣
↪00:00:00.000756  00:00:00.000756  streaming  quorum      1

Instances status
Name       Database Size  Current LSN   Replication role  Status  QoS         Manager␣
↪Version
- ---       -------------  -----------   ----------------  ------  ---         ----------
↪-----
sandbox-1                 3B1/610204B8  Standby (sync)    OK      Guaranteed  1.11.0
sandbox-2                 3B1/61DE3158  Primary           OK      Guaranteed  1.11.0
sandbox-3                 3B1/62618470  Standby (sync)    OK      Guaranteed  1.11.0
```

The command also supports output in `yaml` and `json` format.

Promote

The meaning of this command is to `promote` a pod in the cluster to primary, so you can start with maintenance work or test a switch-over situation in your cluster

```
kubectl cnpg promote cluster-example cluster-example-2
```

Or you can use the instance node number to promote

```
kubectl cnpg promote cluster-example 2
```

Certificates

Clusters created using the CloudNativePG operator work with a CA to sign a TLS authentication certificate.

To get a certificate, you need to provide a name for the secret to store the credentials, the cluster name, and a user for this certificate

```
kubectl cnpg certificate cluster-cert --cnpg-cluster cluster-example --cnpg-user appuser
```

After the secret it's created, you can get it using `kubectl`

```
kubectl get secret cluster-cert
```

And the content of the same in plain text using the following commands:

```
kubectl get secret cluster-cert -o json | jq -r .data | map(@base64d) | .[]
```

Restart

The `kubectl cnpg restart` command can be used in two cases:

- requesting the operator to orchestrate a rollout restart for a certain cluster. This is useful to apply configuration changes to cluster dependent objects, such as ConfigMaps containing custom monitoring queries.
- request a single instance restart, either in-place if the instance is the cluster's primary or deleting and recreating the pod if it is a replica.

```
#  this command will restart a whole cluster in a rollout fashion

kubectl cnpg restart [clusterName]

#  this command will restart a single instance, according to the policy above

kubectl cnpg restart [clusterName] [pod]
```

If the in-place restart is requested but the change cannot be applied without a switchover, the switchover will take precedence over the in-place restart. A common case for this will be a minor upgrade of PostgreSQL image.

Note: If you want ConfigMaps and Secrets to be **automatically** reloaded by instances, you can add a label with key *cnpg.io/reload* to it.

Reload

The `kubectl cnpg reload` command requests the operator to trigger a reconciliation loop for a certain cluster. This is useful to apply configuration changes to cluster dependent objects, such as ConfigMaps containing custom monitoring queries.

The following command will reload all configurations for a given cluster:

```
kubectl cnpg reload [cluster_name]
```

Maintenance

The `kubectl cnpg maintenance` command helps to modify one or more clusters across namespaces and set the maintenance window values, it will change the following fields:

- .spec.nodeMaintenanceWindow.inProgress
- .spec.nodeMaintenanceWindow.reusePVC

Accepts as argument `set` and `unset` using this to set the `inProgress` to `true` in case `set` and to `false` in case of `unset` .

By default, `reusePVC` is always set to `false` unless the `--reusePVC` flag is passed.

The plugin will ask for a confirmation with a list of the cluster to modify and their new values, if this is accepted this action will be applied to all the cluster in the list.

If you want to set in maintenance all the PostgreSQL in your Kubernetes cluster, just need to write the following command:

```
kubectl cnpg maintenance set --all-namespaces
```

And you' ll have the list of all the cluster to update

```
The following are the new values for the clusters
Namespace  Cluster Name     Maintenance  reusePVC
- --------  ------------     -----------  --------
default    cluster-example  true         false
default    pg-backup        true         false
test       cluster-example  true         false
Do you want to proceed? [y/n]: y
```

Report

The `kubectl cnpg report` command bundles various pieces of information into a ZIP file. It aims to provide the needed context to debug problems with clusters in production.

It has two sub-commands: `operator` and `cluster` .

report Operator

The `operator` sub-command requests the operator to provide information regarding the operator deployment, configuration and events.

Important: All confidential information in Secrets and ConfigMaps is REDACTED. The Data map will show the **keys** but the values will be empty. The flag *-S* / *--stopRedaction* will defeat the redaction and show the values. Use only at your own risk, this will share private data.

Note: By default, operator logs are not collected, but you can enable operator log collection with the *--logs* flag

- **deployment information**: the operator Deployment and operator Pod
- **configuration**: the Secrets and ConfigMaps in the operator namespace

- **events**: the Events in the operator namespace
- **webhook configuration**: the mutating and validating webhook configurations
- **webhook service**: the webhook service
- **logs**: logs for the operator Pod (optional, off by default) in JSON-lines format

The command will generate a ZIP file containing various manifest in YAML format (by default, but settable to JSON with the `-o` flag). Use the `-f` flag to name a result file explicitly. If the `-f` flag is not used, a default time-stamped filename is created for the zip file.

Note: The report plugin obeys *kubectl* conventions, and will look for objects constrained by namespace. The CNPG Operator will generally not be installed in the same namespace as the clusters. E.g. the default installation namespace is cnpg-system

```
kubectl cnpg report operator -n <namespace>
```

results in

```
Successfully written report to "report_operator_<TIMESTAMP>.zip" (format: "yaml")
```

With the `-f` flag set:

```
kubectl cnpg report operator -n <namespace> -f reportRedacted.zip
```

Unzipping the file will produce a time-stamped top-level folder to keep the directory tidy:

```
unzip reportRedacted.zip
```

will result in:

```
Archive:  reportRedacted.zip
   creating: report_operator_<TIMESTAMP>/
   creating: report_operator_<TIMESTAMP>/manifests/
  inflating: report_operator_<TIMESTAMP>/manifests/deployment.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/operator-pod.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/events.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/validating-webhook-configuration.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/mutating-webhook-configuration.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/webhook-service.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/cnpg-ca-secret.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/cnpg-webhook-cert.yaml
```

If you activated the `--logs` option, you'd see an extra subdirectory:

```
Archive:  report_operator_<TIMESTAMP>.zip
  <snipped …>
  creating: report_operator_<TIMESTAMP>/operator-logs/
  inflating: report_operator_<TIMESTAMP>/operator-logs/cnpg-controller-manager-
↪66fb98dbc5-pxkmh-logs.jsonl
```

Note: The plugin will try to get the PREVIOUS operator's logs, which is helpful when investigating restarted operators. In all cases, it will also try to get the CURRENT operator logs. If current and previous logs are available, it will show

them both.

```
====== Begin of Previous Log =====
2023-03-28T12:56:41.251711811Z {"level":"info","ts":"2023-03-28T12:56:41Z","logger":
↪"setup","msg":"Starting CloudNativePG Operator","version":"1.19.1","build":{"Version":
↪"1.19.0+dev107","Commit":"cc9bab17","Date":"2023-03-28"}}
2023-03-28T12:56:41.251851909Z {"level":"info","ts":"2023-03-28T12:56:41Z","logger":
↪"setup","msg":"Starting pprof HTTP server","addr":"0.0.0.0:6060"}
  <snipped …>

====== End of Previous Log =====
2023-03-28T12:57:09.854306024Z {"level":"info","ts":"2023-03-28T12:57:09Z","logger":
↪"setup","msg":"Starting CloudNativePG Operator","version":"1.19.1","build":{"Version":
↪"1.19.0+dev107","Commit":"cc9bab17","Date":"2023-03-28"}}
2023-03-28T12:57:09.854363943Z {"level":"info","ts":"2023-03-28T12:57:09Z","logger":
↪"setup","msg":"Starting pprof HTTP server","addr":"0.0.0.0:6060"}
```

If the operator hasn't been restarted, you'll still see the `====== Begin …` and `====== End …` guards, with no content inside.

You can verify that the confidential information is REDACTED by default:

```
cd report_operator_<TIMESTAMP>/manifests/
head cnpg-ca-secret.yaml
```

```yaml
data:
  ca.crt: ""
  ca.key: ""
metadata:
  creationTimestamp: "2022-03-22T10:42:28Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
    fieldsV1:
```

With the `-S` (`--stopRedaction`) option activated, secrets are shown:

```
kubectl cnpg report operator -n <namespace> -f reportNonRedacted.zip -S
```

You'll get a reminder that you're about to view confidential information:

```
WARNING: secret Redaction is OFF. Use it with caution
Successfully written report to "reportNonRedacted.zip" (format: "yaml")
```

```
unzip reportNonRedacted.zip
head cnpg-ca-secret.yaml
```

```yaml
data:
  ca.crt: LS0tLS1CRUdJTiBD…
  ca.key: LS0tLS1CRUdJTiBF…
metadata:
  creationTimestamp: "2022-03-22T10:42:28Z"
```

(continues on next page)

(continued from previous page)

```
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
```

report Cluster

The `cluster` sub-command gathers the following:

- **cluster resources**: the cluster information, same as `kubectl get cluster -o yaml`
- **cluster pods**: pods in the cluster namespace matching the cluster name
- **cluster jobs**: jobs, if any, in the cluster namespace matching the cluster name
- **events**: events in the cluster namespace
- **pod logs**: logs for the cluster Pods (optional, off by default) in JSON-lines format
- **job logs**: logs for the Pods created by jobs (optional, off by default) in JSON-lines format

The `cluster` sub-command accepts the `-f` and `-o` flags, as the `operator` does. If the `-f` flag is not used, a default timestamped report name will be used. Note that the cluster information does not contain configuration Secrets / ConfigMaps, so the `-S` is disabled.

Note: By default, cluster logs are not collected, but you can enable cluster log collection with the *--logs* flag

Usage:

```
kubectl cnpg report cluster <clusterName> [flags]
```

Note that, unlike the `operator` sub-command, for the `cluster` sub-command you need to provide the cluster name, and very likely the namespace, unless the cluster is in the default one.

```
kubectl cnpg report cluster example -f report.zip -n example_namespace
```

and then:

```
unzip report.zip
```

```
Archive:  report.zip
   creating: report_cluster_example_<TIMESTAMP>/
   creating: report_cluster_example_<TIMESTAMP>/manifests/
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster-pods.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster-jobs.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/events.yaml
```

Remember that you can use the `--logs` flag to add the pod and job logs to the ZIP.

```
kubectl cnpg report cluster example -n example_namespace --logs
```

will result in:

```
Successfully written report to "report_cluster_example_<TIMESTAMP>.zip" (format: "yaml")
```

```
unzip report_cluster_<TIMESTAMP>.zip
```

```
Archive:  report_cluster_example_<TIMESTAMP>.zip
   creating: report_cluster_example_<TIMESTAMP>/
   creating: report_cluster_example_<TIMESTAMP>/manifests/
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster-pods.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/cluster-jobs.yaml
  inflating: report_cluster_example_<TIMESTAMP>/manifests/events.yaml
   creating: report_cluster_example_<TIMESTAMP>/logs/
  inflating: report_cluster_example_<TIMESTAMP>/logs/cluster-example-full-1.jsonl
   creating: report_cluster_example_<TIMESTAMP>/job-logs/
  inflating: report_cluster_example_<TIMESTAMP>/job-logs/cluster-example-full-1-initdb-
↪qnnvw.jsonl
  inflating: report_cluster_example_<TIMESTAMP>/job-logs/cluster-example-full-2-join-
↪tvj8r.jsonl
```

Logs

The `kubectl cnpg logs` command allows to follow the logs of a collection of pods related to CloudNativePG in a single go.

It has at the moment one available sub-command: `cluster` .

Cluster logs

The `cluster` sub-command gathers all the pod logs for a cluster in a single stream or file. This means that you can get all the pod logs in a single terminal window, with a single invocation of the command.

As in all the cnpg plugin sub-commands, you can get instructions and help with the `-h` flag:

`kubectl cnpg logs cluster -h`

The `logs` command will display logs in JSON-lines format, unless the `--timestamps` flag is used, in which case, a human readable timestamp will be prepended to each line. In this case, lines will no longer be valid JSON, and tools such as `jq` may not work as desired.

If the `logs cluster` sub-command is given the `-f` flag (aka `--follow`), it will follow the cluster pod logs, and will also watch for any new pods created in the cluster after the command has been invoked. Any new pods found, including pods that have been restarted or re-created, will also have their pods followed. The logs will be displayed in the terminal' s standard-out. This command will only exit when the cluster has no more pods left, or when it is interrupted by the user.

If `logs` is called without the `-f` option, it will read the logs from all cluster pods until the time of invocation and display them in the terminal' s standard-out, then exit. The `-o` or `--output` flag can be provided, to specify the name of the file where the logs should be saved, instead of displaying over standard-out. The `--tail` flag can be used to specify how many log lines will be retrieved from each pod in the cluster. By default, the `logs cluster` sub-command will display all the logs from each pod in the cluster. If combined with the "follow" flag `-f` , the number of logs specified by `--tail` will be retrieved until the current time, and and from then the new logs will be followed.

NOTE: unlike other `cnpg` plugin commands, the `-f` is used to denote "follow" rather than specify a file. This keeps with the convention of `kubectl logs`, which takes `-f` to mean the logs should be followed.

Usage:

```
kubectl cnpg logs cluster <clusterName> [flags]
```

Using the `-f` option to follow:

```
kubectl cnpg report cluster cluster-example -f
```

Using `--tail` option to display 3 lines from each pod and the `-f` option to follow:

```
kubectl cnpg report cluster cluster-example -f --tail 3
```

```
{"level":"info","ts":"2023-06-30T13:37:33Z","logger":"postgres","msg":"2023-06-30␣
↪13:37:33.142 UTC [26] LOG:  ending log output to stderr","source":"/controller/log/
↪postgres","logging_pod":"cluster-example-3"}
{"level":"info","ts":"2023-06-30T13:37:33Z","logger":"postgres","msg":"2023-06-30␣
↪13:37:33.142 UTC [26] HINT:  Future log output will go to log destination \"csvlog\".",
↪"source":"/controller/log/postgres","logging_pod":"cluster-example-3"}
...
...
```

With the `-o` option omitted, and with `--output` specified:

```
kubectl cnpg logs cluster cluster-example --output my-cluster.log

Successfully written logs to "my-cluster.log"
```

Destroy

The `kubectl cnpg destroy` command helps remove an instance and all the associated PVCs from a Kubernetes cluster.

The optional `--keep-pvc` flag, if specified, allows you to keep the PVCs, while removing all `metadata.ownerReferences` that were set by the instance. Additionally, the `cnpg.io/pvcStatus` label on the PVCs will change from `ready` to `detached` to signify that they are no longer in use.

Running again the command without the `--keep-pvc` flag will remove the detached PVCs.

Usage:

```
kubectl cnpg destroy [CLUSTER_NAME] [INSTANCE_ID]
```

The following example removes the `cluster-example-2` pod and the associated PVCs:

```
kubectl cnpg destroy cluster-example 2
```

Cluster hibernation

Sometimes you may want to suspend the execution of a CloudNativePG `Cluster` while retaining its data, then resume its activity at a later time. We' ve called this feature **cluster hibernation**.

Hibernation is only available via the `kubectl cnpg hibernate [on|off]` commands.

Hibernating a CloudNativePG cluster means destroying all the resources generated by the cluster, except the PVCs that belong to the PostgreSQL primary instance.

You can hibernate a cluster with:

```
kubectl cnpg hibernate on <cluster-name>
```

This will:

1. shutdown every PostgreSQL instance
2. detach the PVCs containing the data of the primary instance, and annotate them with the latest database status and the latest cluster configuration
3. delete the `Cluster` resource, including every generated resource - except the aforementioned PVCs

When hibernated, a CloudNativePG cluster is represented by just a group of PVCs, in which the one containing the `PGDATA` is annotated with the latest available status, including content from `pg_controldata` .

> **Warning:** A cluster having fenced instances cannot be hibernated, as fencing is part of the hibernation procedure too.

In case of error the operator will not be able to revert the procedure. You can still force the operation with:

```
kubectl cnpg hibernate on cluster-example --force
```

A hibernated cluster can be resumed with:

```
kubectl cnpg hibernate off <cluster-name>
```

Once the cluster has been hibernated, it' s possible to show the last configuration and the status that PostgreSQL had after it was shut down. That can be done with:

```
kubectl cnpg hibernate status <cluster-name>
```

Benchmarking the database with pgbench

Pgbench can be run against an existing PostgreSQL cluster with following command:

```
kubectl cnpg pgbench <cluster-name> -- --time 30 --client 1 --jobs 1
```

Refer to the *Benchmarking pgbench section* for more details.

Benchmarking the storage with fio

fio can be run on an existing storage class with following command:

```
kubectl cnpg fio <fio-job-name> -n <namespace>
```

Refer to the *Benchmarking fio section* for more details.

Requesting a new physical backup

The `kubectl cnpg backup` command requests a new physical backup for an existing Postgres cluster by creating a new `Backup` resource.

Note: From release 1.21, the *backup* command accepts a new flag, *-m* to specify the backup method. To request a backup using volume snapshots, set *-m volumeSnapshot*

The following example requests an on-demand backup for a given cluster:

```
kubectl cnpg backup [cluster_name]
```

or, if using volume snapshots (from release 1.21)

```
kubectl cnpg backup [cluster_name] -m volumeSnapshot
```

The created backup will be named after the request time:

```
kubectl cnpg backup cluster-example
backup/cluster-example-20230121002300 created
```

By default, a newly created backup will use the backup target policy defined in the cluster to choose which instance to run on. However, you can override this policy with the `--backup-target` option.

In the case of volume snapshot backups, you can also use the `--online` option to request an online/hot backup or an offline/cold one: additionally, you can also tune online backups by explicitly setting the `--immediate-checkpoint` and `--wait-for-archive` options.

The *Appendix A - Common object stores for backups* contains more information about the configuration settings.

Launching psql

The `kubectl cnpg psql` command starts a new PostgreSQL interactive front-end process (psql) connected to an existing Postgres cluster, as if you were running it from the actual pod. This means that you will be using the `postgres` user.

Important: As you will be connecting as *postgres* user, in production environments this method should be used with extreme care, by authorized personnel only.

```
kubectl cnpg psql cluster-example

psql (16.2 (Debian 16.2-1.pgdg110+1))
```

(continues on next page)

(continued from previous page)

```
Type "help" for help.

postgres=#
```

By default, the command will connect to the primary instance. The user can select to work against a replica by using the `--replica` option:

```
kubectl cnpg psql --replica cluster-example
psql (16.2 (Debian 16.2-1.pgdg110+1))

Type "help" for help.

postgres=# select pg_is_in_recovery();
 pg_is_in_recovery
- ------------------
 t
(1 row)

postgres=# \q
```

This command will start `kubectl exec` , and the `kubectl` executable must be reachable in your `PATH` variable to correctly work.

Snapshotting a Postgres cluster

> **Warning:** The *kubectl cnpg snapshot* command has been removed. Please use the *Appendix A - Common object stores for backups* to request backups using volume snapshots.

Using pgAdmin4 for evaluation/demonstration purposes only

pgAdmin stands as the most popular and feature-rich open-source administration and development platform for PostgreSQL. For more information on the project, please refer to the official documentation .

Given that the pgAdmin Development Team maintains official Docker container images, you can install pgAdmin in your environment as a standard Kubernetes deployment.

Important: Deployment of pgAdmin in Kubernetes production environments is beyond the scope of this document and, more broadly, of the CloudNativePG project.

However, **for the purposes of demonstration and evaluation**, CloudNativePG offers a suitable solution. The `cnpg` plugin implements the `pgadmin4` command, providing a straightforward method to connect to a given database `Cluster` and navigate its content in a local environment such as `kind` .

For example, you can install a demo deployment of pgAdmin4 for the `cluster-example` cluster as follows:

```
kubectl cnpg pgadmin4 cluster-example
```

This command will produce:

```
ConfigMap/cluster-example-pgadmin4 created
Deployment/cluster-example-pgadmin4 created
Service/cluster-example-pgadmin4 created
Secret/cluster-example-pgadmin4 created

[...]
```

After deploying pgAdmin, forward the port using kubectl and connect through your browser by following the on-screen instructions.

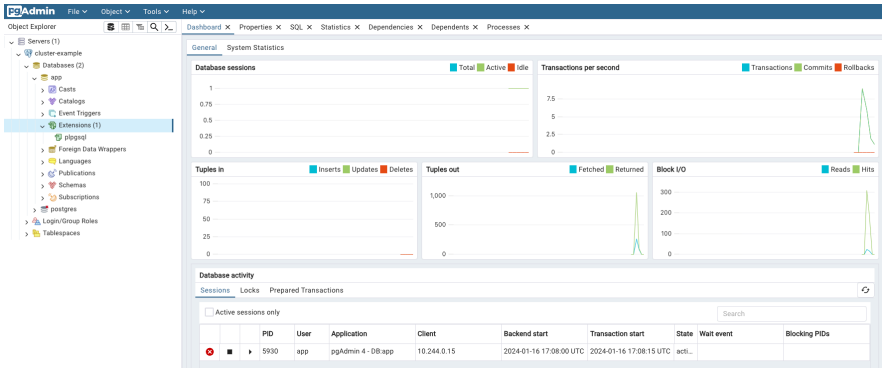


Fig. 17: Screenshot of desktop installation of pgAdmin

As usual, you can use the `--dry-run` option to generate the YAML file:

```
kubectl cnpg pgadmin4 --dry-run cluster-example
```

pgAdmin4 can be installed in either desktop or server mode, with the default being server.

In `server` mode, authentication is required using a randomly generated password, and users must manually specify the database to connect to.

On the other hand, `desktop` mode initiates a pgAdmin web interface without requiring authentication. It automatically connects to the `app` database as the `app` user, making it ideal for quick demos, such as on a local deployment using `kind` :

```
kubectl cnpg pgadmin4 --mode desktop cluster-example
```

After concluding your demo, ensure the termination of the pgAdmin deployment by executing:

```
kubectl cnpg pgadmin4 --dry-run cluster-example | kubectl delete -f -
```

Warning: Never deploy pgAdmin in production using the plugin.

Logical Replication Publications

The `cnpg publication` command group is designed to streamline the creation and removal of *PostgreSQL logical replication publications* . Be aware that these commands are primarily intended for assisting in the creation of logical replication publications, particularly on remote PostgreSQL databases.

> **Warning:** It is crucial to have a solid understanding of both the capabilities and limitations of PostgreSQL's native logical replication system before using these commands. In particular, be mindful of the *logical replication restrictions* .

Creating a new publication

To create a logical replication publication, use the `cnpg publication create` command. The basic structure of this command is as follows:

```
kubectl cnpg publication create \
  --publication <PUBLICATION_NAME> \
  [--external-cluster <EXTERNAL_CLUSTER>]
  <LOCAL_CLUSTER> [options]
```

There are two primary use cases:

- With `--external-cluster` : Use this option to create a publication on an external cluster (i.e. defined in the `externalClusters` stanza). The commands will be issued from the `<LOCAL_CLUSTER>` , but the publication will be for the data in `<EXTERNAL_CLUSTER>` .
- Without `--external-cluster` : Use this option to create a publication in the `<LOCAL_CLUSTER>` PostgreSQL `Cluster` (by default, the `app` database).

> **Warning:** When connecting to an external cluster, ensure that the specified user has sufficient permissions to execute the *CREATE PUBLICATION* command.

You have several options, similar to the CREATE PUBLICATION

command, to define the group of tables to replicate. Notable options include:

- If you specify the `--all-tables` option, you create a publication `FOR ALL TABLES` .
- Alternatively, you can specify multiple occurrences of:
- `--table` : Add a specific table (with an expression) to the publication.
- `--schema` : Include all tables in the specified database schema (available from PostgreSQL 15).

The `--dry-run` option enables you to preview the SQL commands that the plugin will execute.

For additional information and detailed instructions, type the following command:

```
kubectl cnpg publication create --help
```

Example

Given a `source-cluster` and a `destination-cluster` , we would like to create a publication for the data on `source-cluster` . The `destination-cluster` has an entry in the `externalClusters` stanza pointing to `source-cluster` .

We can run:

```
kubectl cnpg publication create destination-cluster  \
  --external-cluster=source-cluster --all-tables
```

which will create a publication for all tables on `source-cluster` , running the SQL commands on the `destination-cluster` .

Or instead, we can run:

```
kubectl cnpg publication create source-cluster \
  --publication=app --all-tables
```

which will create a publication named `app` for all the tables in the `source-cluster` , running the SQL commands on the source cluster.

Note: There are two sample files that have been provided for illustration and inspiration: *logical-source* and *logical-destination* .

Dropping a publication

The `cnpg publication drop` command seamlessly complements the `create` command by offering similar key options, including the publication name, cluster name, and an optional external cluster. You can drop a `PUBLICATION` with the following command structure:

```
kubectl cnpg publication drop \
  --publication <PUBLICATION_NAME> \
  [--external-cluster <EXTERNAL_CLUSTER>]
  <LOCAL_CLUSTER> [options]
```

To access further details and precise instructions, use the following command:

```
kubectl cnpg publication drop --help
```

Logical Replication Subscriptions

The `cnpg subscription` command group is a dedicated set of commands designed to simplify the creation and removal of *PostgreSQL logical replication subscriptions* . These commands are specifically crafted to aid in the establishment of logical replication subscriptions, especially when dealing with remote PostgreSQL databases.

Warning: Before using these commands, it is essential to have a comprehensive understanding of both the capabilities and limitations of PostgreSQL's native logical replication system. In particular, be mindful of the *logical replication restrictions* .

In addition to subscription management, we provide a helpful command for synchronizing all sequences from the source cluster. While its applicability may vary, this command can be particularly useful in scenarios involving major upgrades or data import from remote servers.

Creating a new subscription

To create a logical replication subscription, use the `cnpg subscription create` command. The basic structure of this command is as follows:

```
kubectl cnpg subscription create \
  --subscription <SUBSCRIPTION_NAME> \
  --publication <PUBLICATION_NAME> \
  --external-cluster <EXTERNAL_CLUSTER> \
  <LOCAL_CLUSTER> [options]
```

This command configures a subscription directed towards the specified publication in the designated external cluster, as defined in the `externalClusters` stanza of the `<LOCAL_CLUSTER>` .

For additional information and detailed instructions, type the following command:

```
kubectl cnpg subscription create --help
```

Example

As in the section on publications, we have a `source-cluster` and a `destination-cluster` , and we have already created a publication called `app` .

The following command:

```
kubectl cnpg subscription create destination-cluster \
  --external-cluster=source-cluster \
  --publication=app --subscription=app
```

will create a subscription for `app` on the destination cluster.

> **Warning:** Prioritize testing subscriptions in a non-production environment to ensure their effectiveness and identify any potential issues before implementing them in a production setting.

Note: There are two sample files that have been provided for illustration and inspiration: *logical-source* and *logical-destination* .

Dropping a subscription

The `cnpg subscription drop` command seamlessly complements the `create` command. You can drop a `SUBSCRIPTION` with the following command structure:

```
kubectl cnpg subcription drop \
  --subscription <SUBSCRIPTION_NAME> \
  <LOCAL_CLUSTER> [options]
```

To access further details and precise instructions, use the following command:

```
kubectl cnpg subscription drop --help
```

Synchronizing sequences

One notable constraint of PostgreSQL logical replication, implemented through publications and subscriptions, is the lack of sequence synchronization. This becomes particularly relevant when utilizing logical replication for live database migration, especially to a higher version of PostgreSQL. A crucial step in this process involves updating sequences before transitioning applications to the new database (*cutover*).

To address this limitation, the `cnpg subscription sync-sequences` command offers a solution. This command establishes a connection with the source database, retrieves all relevant sequences, and subsequently updates local sequences with matching identities (based on database schema and sequence name).

You can use the command as shown below:

```
kubectl cnpg subscription sync-sequences \
  --subscription <SUBSCRIPTION_NAME> \
  <LOCAL_CLUSTER>
```

For comprehensive details and specific instructions, utilize the following command:

```
kubectl cnpg subscription sync-sequences --help
```

Example

As in the previous sections for publication and subscription, we have a `source-cluster` and a `destination-cluster` . The publication and the subscription, both called `app` , are already present.

The following command will synchronize the sequences involved in the `app` subscription, from the source cluster into the destination cluster.

```
kubectl cnpg subscription sync-sequences destination-cluster \
  --subscription=app
```

> **Warning:** Prioritize testing subscriptions in a non-production environment to guarantee their effectiveness and detect any potential issues before deploying them in a production setting.

Integration with K9s

The `cnpg` plugin can be easily integrated in K9s , a popular terminal-based UI to interact with Kubernetes clusters.

See *k9s/plugins.yml* for details.

1.4.40 Automated failover

In the case of unexpected errors on the primary for longer than the `.spec.failoverDelay` (by default `0` seconds), the cluster will go into **failover mode**. This may happen, for example, when:

- The primary pod has a disk failure
- The primary pod is deleted
- The `postgres` container on the primary has any kind of sustained failure

In the failover scenario, the primary cannot be assumed to be working properly.

After cases like the ones above, the readiness probe for the primary pod will start failing. This will be picked up in the controller' s reconciliation loop. The controller will initiate the failover process, in two steps:

1. First, it will mark the `TargetPrimary` as `pending` . This change of state will force the primary pod to shutdown, to ensure the WAL receivers on the replicas will stop. The cluster will be marked in failover phase ("Failing over").
2. Once all WAL receivers are stopped, there will be a leader election, and a new primary will be named. The chosen instance will initiate promotion to primary, and, after this is completed, the cluster will resume normal operations. Meanwhile, the former primary pod will restart, detect that it is no longer the primary, and become a replica node.

Important: The two-phase procedure helps ensure the WAL receivers can stop in an orderly fashion, and that the failing primary will not start streaming WALs again upon restart. These safeguards prevent timeline discrepancies between the new primary and the replicas.

During the time the failing primary is being shut down:

1. It will first try a PostgreSQL' s *fast shutdown* with `.spec.switchoverDelay` seconds as timeout. This graceful shutdown will attempt to archive pending WALs.
2. If the fast shutdown fails, or its timeout is exceeded, a PostgreSQL' s *immediate shutdown* is initiated.

Note: "Fast" mode does not wait for PostgreSQL clients to disconnect and will terminate an online backup in progress. All active transactions are rolled back and clients are forcibly disconnected, then the server is shut down. "Immediate" mode will abort all PostgreSQL server processes immediately, without a clean shutdown.

RTO and RPO impact

Failover may result in the service being impacted and/or data being lost:

1. During the time when the primary has started to fail, and before the controller starts failover procedures, queries in transit, WAL writes, checkpoints and similar operations, may fail.
2. Once the fast shutdown command has been issued, the cluster will no longer accept connections, so service will be impacted but no data will be lost.
3. If the fast shutdown fails, the immediate shutdown will stop any pending processes, including WAL writing. Data may be lost.
4. During the time the primary is shutting down and a new primary hasn' t yet started, the cluster will operate without a primary and thus be impaired - but with no data loss.

Note: The timeout that controls fast shutdown is set by *.spec.switchoverDelay* , as in the case of a switchover. Increasing the time for fast shutdown is safer from an RPO point of view, but possibly delays the return to normal operation - negatively affecting RTO.

Warning: As already mentioned in the *In-place updates of the instance manager*

when explaining the switchover process, the `.spec.switchoverDelay` option affects the RPO and RTO of your PostgreSQL database. Setting it to a low value, might favor RTO over RPO but lead to data loss at cluster level and/or backup level. On the contrary, setting it to a high value, might remove the risk of data loss while leaving the cluster without an active primary for a longer time during the switchover.

Delayed failover

As anticipated above, the `.spec.failoverDelay` option allows you to delay the start of the failover procedure by a number of seconds after the primary has been detected to be unhealthy. By default, this setting is set to `0` , triggering the failover procedure immediately.

Sometimes failing over to a new primary can be more disruptive than waiting for the primary to come back online. This is especially true of network disruptions where multiple tiers are affected (i.e., downstream logical subscribers) or when the time to perform the failover is longer than the expected outage.

Enabling a new configuration option to delay failover provides a mechanism to prevent premature failover for short-lived network or node instability.

1.4.41 Troubleshooting

In this page, you can find some basic information on how to troubleshoot CloudNativePG in your Kubernetes cluster deployment.

Hint: As a Kubernetes administrator, you should have the kubectl page bookmarked!

Before you start

Kubernetes environment

What can make a difference in a troubleshooting activity is to provide clear information about the underlying Kubernetes system.

Make sure you know:

- the Kubernetes distribution and version you are using
- the specifications of the nodes where PostgreSQL is running
- as much as you can about the actual *Other S3-compatible Object Storages providers* , including storage class and benchmarks you have done before going into production.
- which relevant Kubernetes applications you are using in your cluster (i.e. Prometheus, Grafana, Istio, Cert-manager, ⋯)
- the situation of continuous backup, in particular if it' s in place and working correctly: in case it is not, make sure you take an *Emergency backup*

before performing any potential disrupting operation

Useful utilities

On top of the mandatory `kubectl` utility, for troubleshooting, we recommend the following plugins/utilities to be available in your system:

- Backup {postgresql-cnpg-io-v1-Backup} for `kubectl`
- jq , a lightweight and flexible command-line JSON processor
- grep , searches one or more input files for lines containing a match to a specified pattern. It is already available in most *nix distros. If you are on Windows OS, you can use findstr as an alternative to `grep` or directly use wsl

and install your preferred *nix distro and use the tools mentioned above.

First steps

To quickly get an overview of the cluster or installation, the `kubectl` plugin is the primary tool to use:

1. the *status subcommand* provides an overview of a cluster
2. the *report subcommand* provides the manifests for clusters and the operator deployment. It can also include logs using the `--logs` option. The report generated via the plugin will include the full cluster manifest.

The plugin can be installed on air-gapped systems via packages. Please refer to the *plugin document* for complete instructions.

Are there backups?

After getting the cluster manifest with the plugin, you should verify if backups are set up and working.

In a cluster with backups set up, you will find, in the cluster Status, the fields `lastSuccessfulBackup` and `firstRecoverabilityPoint` . You should make sure there is a recent `lastSuccessfulBackup` .

A cluster lacking the `.spec.backup` stanza won' t have backups. An insistent message will appear in the PostgreSQL logs:

```
Backup not configured, skip WAL archiving.
```

Before proceeding with troubleshooting operations, it may be advisable to perform an emergency backup depending on your findings regarding backups. Refer to the following section for instructions.

It is **extremely risky** to operate a production database without keeping regular backups.

Emergency backup

In some emergency situations, you might need to take an emergency logical backup of the main `app` database.

Important: The instructions you find below must be executed only in emergency situations and the temporary backup files kept under the data protection policies that are effective in your organization. The dump file is indeed stored in the client machine that runs the *kubectl* command, so make sure that all protections are in place and you have enough space to store the backup file.

The following example shows how to take a logical backup of the `app` database in the `cluster-example` Postgres cluster, from the `cluster-example-1` pod:

```
kubectl exec cluster-example-1 -c postgres \
  -- pg_dump -Fc -d app > app.dump
```

Note: You can easily adapt the above command to backup your cluster, by providing the names of the objects you have used in your environment.

The above command issues a `pg_dump` command in custom format, which is the most versatile way to take logical backups in PostgreSQL .

The next step is to restore the database. We assume that you are operating on a new PostgreSQL cluster that' s been just initialized (so the `app` database is empty).

The following example shows how to restore the above logical backup in the `app` database of the `new-cluster-example` Postgres cluster, by connecting to the primary (`new-cluster-example-1` pod):

```
kubectl exec -i new-cluster-example-1 -c postgres \
  -- pg_restore --no-owner --role=app -d app --verbose < app.dump
```

Important: The example in this section assumes that you have no other global objects (databases and roles) to dump and restore, as per our recommendation. In case you have multiple roles, make sure you have taken a backup using *pg_dumpall -g* and you manually restore them in the new cluster. In case you have multiple databases, you need to

repeat the above operation one database at a time, making sure you assign the right ownership. If you are not familiar with PostgreSQL, we advise that you do these critical operations under the guidance of a professional support company.

The above steps might be integrated into the `cnpg` plugin at some stage in the future.

Logs

Every resource created and controlled by CloudNativePG logs to standard output, as expected by Kubernetes, and directly in JSON format. As a result, you should rely on the `kubectl logs` command to retrieve logs from a given resource.

For more information, type:

```
kubectl logs --help
```

Hint: JSON logs are great for machine reading, but hard to read for human beings. Our recommendation is to use the *jq* command to improve usability. For example, you can *pipe* the *kubectl logs* command with *| jq -C* .

Note: In the sections below, we will show some examples on how to retrieve logs about different resources when it comes to troubleshooting CloudNativePG.

Operator information

By default, the CloudNativePG operator is installed in the `cnpg-system` namespace in Kubernetes as a `Deployment` (see the *Details about the deployment*

for details).

You can get a list of the operator pods by running:

```
kubectl get pods -n cnpg-system
```

Note: Under normal circumstances, you should have one pod where the operator is running, identified by a name starting with *cnpg-controller-manager-* . In case you have set up your operator for high availability, you should have more entries. Those pods are managed by a deployment named *cnpg-controller-manager* .

Collect the relevant information about the operator that is running in pod `<POD>` with:

```
kubectl describe pod -n cnpg-system <POD>
```

Then get the logs from the same pod by running:

```
kubectl logs -n cnpg-system <POD>
```

Gather more information about the operator

Get logs from all pods in CloudNativePG operator Deployment (in case you have a multi operator deployment) by running:

```
kubectl logs -n cnpg-system \
  deployment/cnpg-controller-manager --all-containers=true
```

Tip: You can add *-f* flag to above command to follow logs in real time.

Save logs to a JSON file by running:

```
kubectl logs -n cnpg-system \
  deployment/cnpg-controller-manager --all-containers=true | \
  jq -r . > cnpg_logs.json
```

Get CloudNativePG operator version by using `kubectl-cnpg` plugin:

```
kubectl-cnpg status <CLUSTER>
```

Output:

```
Cluster in healthy state
Name               cluster-example
Namespace          default
System ID          7044925089871458324
PostgreSQL Image   ghcr.io/cloudnative-pg/postgresql:16.2-3
Primary instance   cluster-example-1
Instances          3
Ready instances    3
Current Write LSN  0/5000000 (Timeline: 1 - WAL File: 000000010000000000000004)

Continuous Backup status
Not configured

Streaming Replication status
Name               Sent LSN   Write LSN  Flush LSN  Replay LSN  Write Lag       Flush␣
↪Lag       Replay Lag      State      Sync State  Sync Priority
- ---               --------   ---------  ---------  ----------  ---------       --------
↪-       ----------      -----      ----------  -------------
cluster-example-2  0/5000000  0/5000000  0/5000000  0/5000000   00:00:00        00:00:00
       00:00:00        streaming  async       0
cluster-example-3  0/5000000  0/5000000  0/5000000  0/5000000   00:00:00.10033  00:00:00.
↪10033  00:00:00.10033  streaming  async       0

Instances status
Name               Database Size  Current LSN  Replication role  Status  QoS        ␣
↪Manager Version
- ---               -------------  -----------  ----------------  ------  ---         ---
↪------------
cluster-example-1  33 MB          0/5000000    Primary           OK      BestEffort  1.
↪12.0
```

(continues on next page)

(continued from previous page)

```
cluster-example-2  33 MB          0/5000000    Standby (async)   OK       BestEffort  1.
↪12.0
cluster-example-3  33 MB          0/5000060    Standby (async)   OK       BestEffort  1.
↪12.0
```

Cluster information

You can check the status of the <CLUSTER> cluster in the NAMESPACE namespace with:

```
kubectl get cluster -n <NAMESPACE> <CLUSTER>
```

Output:

```
NAME        AGE        INSTANCES   READY   STATUS                     PRIMARY
<CLUSTER>   10d4h3m    3           3       Cluster in healthy state   <CLUSTER>-1
```

The above example reports a healthy PostgreSQL cluster of 3 instances, all in *ready* state, and with <CLUSTER>-1 being the primary.

In case of unhealthy conditions, you can discover more by getting the manifest of the `Cluster` resource:

```
kubectl get cluster -o yaml -n <NAMESPACE> <CLUSTER>
```

Another important command to gather is the `status` one, as provided by the `cnpg` plugin:

```
kubectl cnpg status -n <NAMESPACE> <CLUSTER>
```

Tip: You can print more information by adding the *--verbose* option.

Note: Besides knowing cluster status, you can also do the following things with the cnpg plugin: Promote a replica.
 Manage certificates.
 Make a rollout restart cluster to apply configuration changes.
 Make a reconciliation loop to reload and apply configuration changes.
 For more information, please see Backup {postgresql-cnpg-io-v1-Backup} documentation.

Get PostgreSQL container image version:

```
kubectl describe cluster <CLUSTER_NAME> -n <NAMESPACE> | grep "Image Name"
```

Output:

```
Image Name:    ghcr.io/cloudnative-pg/postgresql:16.2-3
```

Note: Also you can use *kubectl-cnpg status -n <NAMESPACE> <CLUSTER_NAME>* to get the same information.

Pod information

You can retrieve the list of instances that belong to a given PostgreSQL cluster with:

```
kubectl get pod -l cnpg.io/cluster=<CLUSTER> -L role -n <NAMESPACE>
```

Output:

```
NAME          READY   STATUS    RESTARTS   AGE       ROLE
<CLUSTER>-1   1/1     Running   0          10d4h5m   primary
<CLUSTER>-2   1/1     Running   0          10d4h4m   replica
<CLUSTER>-3   1/1     Running   0          10d4h4m   replica
```

You can check if/how a pod is failing by running:

```
kubectl get pod -n <NAMESPACE> -o yaml <CLUSTER>-<N>
```

You can get all the logs for a given PostgreSQL instance with:

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N>
```

If you want to limit the search to the PostgreSQL process only, you can run:

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq select(.logger=="postgres") | .record.message
```

The following example also adds the timestamp in a user-friendly format:

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq -r select(.logger=="postgres") | [(.ts|strflocaltime("%Y-%m-%dT%H:%M:%S %Z")), .
↪record.message] | @csv
```

Gather and filter extra information about PostgreSQL pods

Check logs from a specific pod that has crashed:

```
kubectl logs -n <NAMESPACE> --previous <CLUSTER>-<N>
```

Get FATAL errors from a specific PostgreSQL pod:

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq -r .record | select(.error_severity == "FATAL")
```

Output:

```
{
  "log_time": "2021-11-08 14:07:44.520 UTC",
  "user_name": "streaming_replica",
  "process_id": "68",
  "connection_from": "10.244.0.10:60616",
  "session_id": "61892f30.44",
  "session_line_num": "1",
  "command_tag": "startup",
```

(continues on next page)

(continued from previous page)

```
  "session_start_time": "2021-11-08 14:07:44 UTC",
  "virtual_transaction_id": "3/75",
  "transaction_id": "0",
  "error_severity": "FATAL",
  "sql_state_code": "28000",
  "message": "role \"streaming_replica\" does not exist",
  "backend_type": "walsender"
}
```

Filter PostgreSQL DB error messages in logs for a specific pod:

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | jq -r .err | select(. != null)
```

Output:

```
dial unix /controller/run/.s.PGSQL.5432: connect: no such file or directory
```

Get messages matching `err` word from a specific pod:

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | jq -r .msg | grep "err"
```

Output:

```
2021-11-08 14:07:39.610 UTC [15] LOG:  ending log output to stderr
```

Get all logs from PostgreSQL process from a specific pod:

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq -r . | select(.logger == "postgres") | select(.msg != "record") | .msg
```  Output:

```shell
2021-11-08 14:07:52.591 UTC [16] LOG:  redirecting log output to logging collector␣
↪process
2021-11-08 14:07:52.591 UTC [16] HINT:  Future log output will appear in directory "/
↪controller/log".
2021-11-08 14:07:52.591 UTC [16] LOG:  ending log output to stderr
2021-11-08 14:07:52.591 UTC [16] HINT:  Future log output will go to log destination
↪"csvlog".
```

Get pod logs filtered by fields with values and join them separated by `|` running:

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq -r [.level, .ts, .logger, .msg] | join(" | ")
```

Output:

```
info | 1636380469.5728037 | wal-archive | Backup not configured, skip WAL archiving
info | 1636383566.0664876 | postgres | record
```

Backup information

You can list the backups that have been created for a named cluster with:

```
kubectl get backup -l cnpg.io/cluster=<CLUSTER>
```

Important: Backup labelling has been introduced in version 1.10.0 of CloudNativePG. So only those resources that have been created with that version or a higher one will contain such a label.

Storage information

Sometimes is useful to double-check the StorageClass used by the cluster to have some more context during investigations or troubleshooting, like this:

```
STORAGECLASS=$(kubectl get pvc <POD> -o jsonpath={.spec.storageClassName})
kubectl get storageclasses $STORAGECLASS -o yaml
```

We are taking the StorageClass from one of the cluster pod here since often clusters are created using the default StorageClass.

Node information

Kubernetes nodes is where ultimately PostgreSQL pods will be running. It's strategically important to know as much as we can about them.

You can get the list of nodes in your Kubernetes cluster with:

```
#  look at the worker nodes and their status

kubectl get nodes -o wide
```

Additionally, you can gather the list of nodes where the pods of a given cluster are running with:

```
kubectl get pod -l cnpg.io/cluster=<CLUSTER> \
  -L role -n <NAMESPACE> -o wide
```

The latter is important to understand where your pods are distributed - very useful if you are using *affinity/anti-affinity rules and/or tolerations* .

Conditions

Like many native kubernetes objects like here , Cluster exposes `status.conditions` as well. This allows one to 'wait' for a particular event to occur instead of relying on the overall cluster health state. Available conditions as of now are:

- LastBackupSucceeded
- ContinuousArchiving
- Ready

`LastBackupSucceeded` is reporting the status of the latest backup. If set to `True` the last backup has been taken correctly, it is set to `False` otherwise.

`ContinuousArchiving` is reporting the status of the WAL archiving. If set to `True` the last WAL archival process has been terminated correctly, it is set to `False` otherwise.

`Ready` is `True` when the cluster has the number of instances specified by the user and the primary instance is ready. This condition can be used in scripts to wait for the cluster to be created.

How to wait for a particular condition

- Backup:

```
$ kubectl wait --for=condition=LastBackupSucceeded cluster/<CLUSTER-NAME> -n <NAMESPACE>
```

- ContinuousArchiving:

```
$ kubectl wait --for=condition=ContinuousArchiving cluster/<CLUSTER-NAME> -n <NAMESPACE>
```

- Ready (Cluster is ready or not):

```
$ kubectl wait --for=condition=Ready cluster/<CLUSTER-NAME> -n <NAMESPACE>
```  Below is a snippet of a `cluster.status`  that contains a failing condition.

```bash
$ kubectl get cluster/<cluster-name> -o yaml
.
.
.
  status:
    conditions:
    - message: unexpected failure invoking barman-cloud-wal-archive: exit status
        2
      reason: ContinuousArchivingFailing
      status: "False"
      type: ContinuousArchiving

    - message: exit status 2
      reason: LastBackupFailed
      status: "False"
      type: LastBackupSucceeded

    - message: Cluster Is Not Ready
      reason: ClusterIsNotReady
      status: "False"
      type: Ready
```

Networking

CloudNativePG requires basic networking and connectivity in place. You can find more information in the *Networking* section.

If installing CloudNativePG in an existing environment, there might be network policies in place, or other network configuration made specifically for the cluster, which could have an impact on the required connectivity between the operator and the cluster pods and/or the between the pods.

You can look for existing network policies with the following command:

```
kubectl get networkpolicies
```

There might be several network policies set up by the Kubernetes network administrator.

```
$ kubectl get networkpolicies
NAME                   POD-SELECTOR                      AGE
allow-prometheus       cnpg.io/cluster=cluster-example   47m
default-deny-ingress   <none>                            57m
```

PostgreSQL core dumps

Although rare, PostgreSQL can sometimes crash and generate a core dump in the `PGDATA` folder. When that happens, normally it is a bug in PostgreSQL (and most likely it has already been solved - this is why it is important to always run the latest minor version of PostgreSQL).

CloudNativePG allows you to control what to include in the core dump through the `cnpg.io/coredumpFilter` annotation.

Note: Please refer to *Labels and annotations*

for more details on the standard annotations that CloudNativePG provides.

By default, the `cnpg.io/coredumpFilter` is set to `0x31` in order to exclude shared memory segments from the dump, as this is the safest approach in most cases.

Note: Please refer to Core dump filtering settings . for more details on how to set the bitmask that controls the core dump filter.

Important: Beware that this setting only takes effect during Pod startup and that changing the annotation doesn't trigger an automated rollout of the instances.

Although you might not personally be involved in inspecting core dumps, you might be asked to provide them so that a Postgres expert can look into them. First, verify that you have a core dump in the `PGDATA` directory with the following command (please run it against the correct pod where the Postgres instance is running):

```
kubectl exec -ti POD -c postgres \
  -- find /var/lib/postgresql/data/pgdata -name core.*
```

Under normal circumstances, this should return an empty set. Suppose, for example, that we have a core dump file:

```
/var/lib/postgresql/data/pgdata/core.14177
```

Once you have verified the space on disk is sufficient, you can collect the core dump on your machine through `kubectl cp` as follows:

```
kubectl cp POD:/var/lib/postgresql/data/pgdata/core.14177 core.14177
```

You now have the file. Make sure you free the space on the server by removing the core dumps.

Some common issues

Storage is full

If one or more pods in the cluster are in `CrashloopBackoff` and logs suggest this could be due to a full disk, you probably have to increase the size of the instance' s `PersistentVolumeClaim` . Please look at the *Volume expansion* in the documentation.

Pods are stuck in `Pending` state

In case a Cluster' s instance is stuck in the `Pending` phase, you should check the pod' s `Events` section to get an idea of the reasons behind this:

```
kubectl describe pod -n <NAMESPACE> <POD>
```

Some of the possible causes for this are:

- No nodes are matching the `nodeSelector`
- Tolerations are not correctly configured to match the nodes' taints
- No nodes are available at all: this could also be related to `cluster-autoscaler` hitting some limits, or having some temporary issues

In this case, it could also be useful to check events in the namespace:

```
kubectl get events -n <NAMESPACE>
#  list events in chronological order

kubectl get events -n <NAMESPACE> --sort-by=.metadata.creationTimestamp
```

Replicas out of sync when no backup is configured

Sometimes replicas might be switched off for a bit of time due to maintenance reasons (think of when a Kubernetes nodes is drained). In case your cluster does not have backup configured, when replicas come back up, they might require a WAL file that is not present anymore on the primary (having been already recycled according to the WAL management policies as mentioned in The `postgresql section <The *postgresql* section>`), and fall out of synchronization.

Similarly, when `pg_rewind` might require a WAL file that is not present anymore in the former primary, reporting `pg_rewind: error: could not open file` .

In these cases, pods cannot become ready anymore, and you are required to delete the PVC and let the operator rebuild the replica.

If you rely on dynamically provisioned Persistent Volumes, and you are confident in deleting the PV itself, you can do so with:

```
PODNAME=<POD>
VOLNAME=$(kubectl get pv -o json | \
  jq -r .items[]|select(.spec.claimRef.name==\"$PODNAME\")|.metadata.name)

kubectl delete pod/$PODNAME pvc/$PODNAME pvc/$PODNAME-wal pv/$VOLNAME
```

Cluster stuck in `Creating new replica`

Cluster is stuck in "Creating a new replica", while pod logs don't show relevant problems. This has been found to be related to the next issue *on connectivity*. From releases 1.20.1, 1.19.3, and 1.18.5, networking issues will be more clearly reflected in the status column as follows:

```
Instance Status Extraction Error: HTTP communication issue
```

Networking is impaired by installed Network Policies

As pointed out in the *networking section*, local network policies could prevent some of the required connectivity.

A tell-tale sign that connectivity is impaired is the presence in the operator logs of messages like:

```
"Cannot extract Pod status", [···snipped···] "Get \"http://<pod IP>:8000/pg/status\": dial␣
↪tcp <pod IP>:8000: i/o timeout"
```

You should list the network policies, and look for any policies restricting connectivity.

```
$ kubectl get networkpolicies
NAME                   POD-SELECTOR                      AGE
allow-prometheus       cnpg.io/cluster=cluster-example   47m
default-deny-ingress   <none>                            57m
```

For example, in the listing above, `default-deny-ingress` seems a likely culprit. You can drill into it:

```
$ kubectl get networkpolicies default-deny-ingress -o yaml
<···snipped···>
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

In the *networking page* you can find a network policy file that you can customize to create a `NetworkPolicy` explicitly allowing the operator to connect cross-namespace to cluster pods.

Error while bootstrapping the data directory

If your Cluster's initialization job crashes with a "Bus error (core dumped) child process exited with exit code 135", you likely need to fix the Cluster hugepages settings.

The reason is the incomplete support of hugepages in the cgroup v1 that should be fixed in v2. For more information, check the PostgreSQL BUG #17757: Not honoring huge_pages setting during initdb causes DB crash in Kubernetes.

To check whether hugepages are enabled, run `grep HugePages /proc/meminfo` on the Kubernetes node and check if hugepages are present, their size, and how many are free.

If the hugepages are present, you need to configure how much hugepages memory every PostgreSQL pod should have available.

For example:

```
postgresql:
  parameters:
    shared_buffers: "128MB"

resources:
  requests:
    memory: "512Mi"
  limits:
    hugepages-2Mi: "512Mi"
```

Please remember that you must have enough hugepages memory available to schedule every Pod in the Cluster (in the example above, at least 512MiB per Pod must be free).

1.4.42 Fencing

Fencing in CloudNativePG is the ultimate process of protecting the data in one, more, or even all instances of a PostgreSQL cluster when they appear to be malfunctioning. When an instance is fenced, the PostgreSQL server process (`postmaster`) is guaranteed to be shut down, while the pod is kept running. This makes sure that, until the fence is lifted, data on the pod is not modified by PostgreSQL and that the file system can be investigated for debugging and troubleshooting purposes.

How to fence instances

In CloudNativePG you can fence:

- a specific instance
- a list of instances
- an entire Postgres `Cluster`

Fencing is controlled through the content of the `cnpg.io/fencedInstances` annotation, which expects a JSON formatted list of instance names. If the annotation is set to `'["*"]'` , a singleton list with a wildcard, the whole cluster is fenced. If the annotation is set to an empty JSON list, the operator behaves as if the annotation was not set.

For example:

- `cnpg.io/fencedInstances: '["cluster-example-1"]'` will fence just the `cluster-example-1` instance

- `cnpg.io/fencedInstances: '["cluster-example-1","cluster-example-2"]'` will fence the `cluster-example-1` and `cluster-example-2` instances
- `cnpg.io/fencedInstances: '["*"]'` will fence every instance in the cluster.

The annotation can be manually set on the Kubernetes object, for example via the `kubectl annotate` command, or in a transparent way using the `kubectl cnpg fencing on` subcommand:

```
#  to fence only one instance

kubectl cnpg fencing on cluster-example 1

#  to fence all the instances in a Cluster

kubectl cnpg fencing on cluster-example "*"
```

Here is an example of a `Cluster` with an instance that was previously fenced:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
    annotations:
      cnpg.io/fencedInstances: ["cluster-example-1"]
[...]
```

How to lift fencing

Fencing can be lifted by clearing the annotation, or set it to a different value.

As for fencing, this can be done either manually with `kubectl annotate` , or using the `kubectl cnpg fencing` subcommand as follows:

```
#  to lift the fencing only for one instance

#  N.B.: at the moment this wont work if the whole cluster was fenced previously,

#        in that case you will have to manually set the annotation as explained above

kubectl cnpg fencing off cluster-example 1

#  to lift the fencing for all the instances in a Cluster

kubectl cnpg fencing off cluster-example "*"
```

How fencing works

Once an instance is set for fencing, the procedure to shut down the `postmaster` process is initiated, identical to the one of the switchover. This consists of an initial fast shutdown with a timeout set to `.spec.switchoverDelay` , followed by an immediate shutdown. Then:

- the Pod will be kept alive
- the Pod won't be marked as *Ready*
- all the changes that don't require the Postgres instance to be up will be reconciled, including:
  - configuration files
  - certificates and all the cryptographic material
- metrics will not be collected, except `cnpg_collector_fencing_on` which will be set to 1

> **Warning:** If a **primary instance** is fenced, its postmaster process is shut down but no failover is performed, interrupting the operativity of the applications. When the fence will be lifted, the primary instance will be started up again without performing a failover.

Given that, we advise users to fence primary instances only if strictly required.

If a fenced instance is deleted, the pod will be recreated normally, but the postmaster won't be started. This can be extremely helpful when instances are `Crashlooping` .

1.4.43 Declarative hibernation

CloudNativePG is designed to keep PostgreSQL clusters up, running and available anytime.

There are some kinds of workloads that require the database to be up only when the workload is active. Batch-driven solutions are one such case.

In batch-driven solutions, the database needs to be up only when the batch process is running.

The declarative hibernation feature enables saving CPU power by removing the database Pods, while keeping the database PVCs.

> **Note:** Declarative hibernation is different from the existing implementation of imperative hibernation via the `cnpg plugin <Kubectl Plugin>` . Imperative hibernation shuts down all Postgres instances in the High Availability cluster, and keeps a static copy of the PVCs of the primary that contain *PGDATA* and WALs. The plugin enables to exit the hibernation phase, by resuming the primary and then recreating all the replicas - if they exist.

Hibernation

To hibernate a cluster, set the `cnpg.io/hibernation=on` annotation:

```
$ kubectl annotate cluster <cluster-name> --overwrite cnpg.io/hibernation=on
```

A hibernated cluster won't have any running Pods, while the PVCs are retained so that the cluster can be rehydrated at a later time. Replica PVCs will be kept in addition to the primary's PVC.

The hibernation procedure will delete the primary Pod and then the replica Pods, avoiding switchover, to ensure the replicas are kept in sync.

The hibernation status can be monitored by looking for the `cnpg.io/hibernation` condition:

```
$ kubectl get cluster <cluster-name> -o "jsonpath={.status.conditions[?(.type==\"cnpg.io/
↪hibernation\")]}"

{
        "lastTransitionTime":"2023-03-05T16:43:35Z",
        "message":"Cluster has been hibernated",
        "reason":"Hibernated",
        "status":"True",
        "type":"cnpg.io/hibernation"
}
```

The hibernation status can also be read with the `status` sub-command of the `cnpg` plugin for `kubectl` :

```
$ kubectl cnpg status <cluster-name>
Cluster Summary
Name:               cluster-example
Namespace:          default
PostgreSQL Image:   ghcr.io/cloudnative-pg/postgresql:16.2
Primary instance:   cluster-example-2
Status:             Cluster in healthy state
Instances:          3
Ready instances:    0

Hibernation
Status   Hibernated
Message  Cluster has been hibernated
Time     2023-03-05 16:43:35 +0000 UTC
[..]
```

Rehydration

To rehydrate a cluster, either set the `cnpg.io/hibernation` annotation to `off` :

```
$ kubectl annotate cluster <cluster-name> --overwrite cnpg.io/hibernation=off
```

Or, just unset it altogether:

```
$ kubectl annotate cluster <cluster-name> cnpg.io/hibernation-
```

The Pods will be recreated and the cluster will resume operation.

1.4.44 PostGIS

PostGIS is a very popular open source extension for PostgreSQL

that introduces support for storing GIS (Geographic Information Systems) objects in the database and be queried via SQL.

Important: This section assumes you are familiar with PostGIS and provides some basic information about how to create a new PostgreSQL cluster with a PostGIS database in Kubernetes via CloudNativePG.

The CloudNativePG Community maintains container images that are built on top of the official *PostGIS images hosted on DockerHub* . For more information please visit:

- The *postgis-containers*
- The *postgis-containers*

Basic concepts about a PostGIS cluster

Conceptually, a PostGIS-based PostgreSQL cluster (or simply a PostGIS cluster) is like any other PostgreSQL cluster. The only differences are:

- the presence in the system of PostGIS and related libraries
- the presence in the database(s) of the PostGIS extension

Since CloudNativePG is based on Immutable Application Containers, the only way to provision PostGIS is to add it to the container image that you use for the operand. The *Container Image Requirements* provides detailed instructions on how this is achieved. More simply, you can just use the PostGIS container images from the Community, as in the examples below.

The second step is to install the extension in the PostgreSQL database. You can do this in two ways:

- install it in the application database, which is the main and supposedly only database you host in the cluster according to the microservice architecture, or
- install it in the `template1` database so as to make it available for all the databases you end up creating in the cluster, in case you adopt the monolith architecture where the instance is shared by multiple databases

Note: For more information on the microservice vs monolith architecture in the database please refer to the *How many databases should be hosted in a single PostgreSQL instance?*

or the *Database import* .

Create a new PostgreSQL cluster with PostGIS

Let' s suppose you want to create a new PostgreSQL 14 cluster with PostGIS 3.2.

The first step is to ensure you use the right PostGIS container image for the operand, and properly set the `.spec.imageName` option in the `Cluster` resource.

The below provides some guidance on how the creation of a PostGIS cluster can be done.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgis-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgis:14
  bootstrap:
    initdb:
      postInitTemplateSQL:
        - CREATE EXTENSION postgis;
        - CREATE EXTENSION postgis_topology;
        - CREATE EXTENSION fuzzystrmatch;
        - CREATE EXTENSION postgis_tiger_geocoder;

  storage:
    size: 1Gi
```

Warning: Please consider that, although convention over configuration applies in CloudNativePG, you should spend time configuring and tuning your system for production. Also the *imageName* in the example below deliberately points to the latest available image for PostgreSQL 14 - you should use a specific image name or, preferably, the SHA256 digest for true immutability.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgis-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgis:14
  bootstrap:
    initdb:
      postInitTemplateSQL:
        - CREATE EXTENSION postgis;
        - CREATE EXTENSION postgis_topology;
        - CREATE EXTENSION fuzzystrmatch;
        - CREATE EXTENSION postgis_tiger_geocoder;

  storage:
    size: 1Gi
```

The example relies on the `postInitTemplateSQL` option which executes a list of queries against the `template1` database, before the actual creation of the application database (called `app`). This means that, once you have applied the manifest and the cluster is up, you will have the above extensions installed in both the template database and the application database, ready for use.

Note: Take some time and look at the available options in *.spec.bootstrap.initdb* from the *API reference* , such as *postInitApplicationSQL* .

You can easily verify the available version of PostGIS that is in the container, by connecting to the `app` database (you

might obtain different values from the ones in this document):

```
$ kubectl exec -ti postgis-example-1 -- psql app
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
psql (16.2 (Debian 16.2-1.pgdg110+1))
Type "help" for help.

app=# SELECT * FROM pg_available_extensions WHERE name ~ ^postgis ORDER BY 1;
           name           | default_version | installed_version |
  comment
- -------------------------+-----------------+-------------------+-----------------------
↪-------------------------------------
 postgis                  | 3.2.2           | 3.2.2             | PostGIS geometry and␣
↪geography spatial types and functions
 postgis-3                | 3.2.2           |                   | PostGIS geometry and␣
↪geography spatial types and functions
 postgis_raster           | 3.2.2           |                   | PostGIS raster types␣
↪and functions
 postgis_raster-3         | 3.2.2           |                   | PostGIS raster types␣
↪and functions
 postgis_sfcgal           | 3.2.2           |                   | PostGIS SFCGAL␣
↪functions
 postgis_sfcgal-3         | 3.2.2           |                   | PostGIS SFCGAL␣
↪functions
 postgis_tiger_geocoder   | 3.2.2           | 3.2.2             | PostGIS tiger geocoder␣
↪and reverse geocoder
 postgis_tiger_geocoder-3 | 3.2.2           |                   | PostGIS tiger geocoder␣
↪and reverse geocoder
 postgis_topology         | 3.2.2           | 3.2.2             | PostGIS topology␣
↪spatial types and functions
 postgis_topology-3       | 3.2.2           |                   | PostGIS topology␣
↪spatial types and functions
(10 rows)
```

The next step is to verify that the extensions listed in the `postInitTemplateSQL` section have been correctly installed in the `app` database.

```
app=# \dx
                                        List of installed extensions
          Name          | Version |   Schema   |                        Description
- ------------------------+---------+------------+---------------------------------------
↪--------------------
 fuzzystrmatch          | 1.1     | public     | determine similarities and distance␣
↪between strings
 plpgsql                | 1.0     | pg_catalog | PL/pgSQL procedural language
 postgis                | 3.2.2   | public     | PostGIS geometry and geography spatial␣
↪types and functions
 postgis_tiger_geocoder | 3.2.2   | tiger      | PostGIS tiger geocoder and reverse␣
↪geocoder
 postgis_topology       | 3.2.2   | topology   | PostGIS topology spatial types and␣
↪functions
(5 rows)
```

Finally:

```
app=# SELECT postgis_full_version();
                                                                                postgis_full_
↪version
- ---------------------------------------------------------------------------------------
↪-----------------------------------------------------------------------------------
 POSTGIS="3.2.2 628da50" [EXTENSION] PGSQL="140" GEOS="3.9.0-CAPI-1.16.2" PROJ="7.2.1"␣
↪LIBXML="2.9.10" LIBJSON="0.15" LIBPROTOBUF="1.3.3" WAGYU="0.5.0 (Internal)" TOPOLOGY
(1 row)
```

1.4.45 End-to-End Tests

CloudNativePG is automatically tested after each commit via a suite of **End-to-end (E2E) tests** (or integration tests) which ensure that the operator correctly deploys and manages PostgreSQL clusters.

Kubernetes versions 1.23 through 1.29, and PostgreSQL versions 12 through 16, are tested for each commit, helping detect bugs at an early stage of the development process.

For each tested version of Kubernetes and PostgreSQL, a Kubernetes cluster is created using kind , run on the GitHub Actions platform, and the following suite of E2E tests are performed on that cluster:

- **Basic:**
- Installation of the operator
- Creation of a Cluster
- Usage of a persistent volume for data storage
- **Service connectivity:**
- Connection via services, including read-only
- Connection via user-provided server and/or client certificates
- PgBouncer
- **Self-healing:**
- Failover
- Switchover
- Primary endpoint switch in case of failover in less than 10 seconds
- Primary endpoint switch in case of switchover in less than 20 seconds
- Recover from a degraded state in less than 60 seconds
- PVC Deletion
- Corrupted PVC
- **Backup and Restore:**
- Backup and restore from Volume Snapshots
- Backup and ScheduledBackups execution using Barman Cloud on S3
- Backup and ScheduledBackups execution using Barman Cloud on Azure blob storage
- Restore from backup using Barman Cloud on S3
- Restore from backup using Barman Cloud on Azure blob storage

- Point-in-time recovery (PITR) on Azure, S3 storage
- Wal-Restore (sequential / parallel)
- **Operator:**
- Operator Deployment
- Operator configuration via ConfigMap
- Operator pod deletion
- Operator pod eviction
- Operator upgrade
- Operator High Availability
- **Observability:**
- Metrics collection
- PgBouncer Metrics
- JSON log format
- **Replication:**
- Replication Slots
- Synchronous replication
- Scale-up and scale-down of a Cluster
- **Replica clusters**
- Bootstrapping a replica cluster from backup
- Bootstrapping a replica cluster via streaming
- Bootstrapping via volume snapshots
- Detaching a replica cluster
- **Plugin:**
- Cluster Hibernation using CNPG plugin
- Fencing
- Creation of a connection certificate
- **Postgres Configuration:**
- Manage PostgreSQL configuration changes
- Rolling updates when changing PostgreSQL images
- Rolling updates when changing ImageCatalog/ClusterImageCatalog images
- Rolling updates on hot standby sensitive parameter changes
- Database initialization via InitDB
- **Pod Scheduling:**
- Tolerations and taints
- Pod affinity using `NodeSelector`
- Rolling updates on PodSpec drift detection

- In-place upgrades
- Multi-Arch availability
- **Cluster Metadata:**
- ConfigMap for Cluster Labels and Annotations
- Object metadata
- **Recovery:**
- Data corruption
- pg_basebackup
- **Importing Databases:**
- Microservice approach
- Monolith approach
- **Storage:**
- Storage expansion
- Dedicated PG_WAL persistent volume
- **Security:** - AppArmor annotation propagation. Executed only on Azure environment
- **Maintenance:**
- Node Drain with maintenance window
- Node Drain with single-instance cluster with/without Pod Disruption Budgets
- **Hibernation** - Declarative hibernation / rehydration
- **Volume snapshots**
- Backup/restore for cold and online snapshots
- Point-in-time recovery (PITR) for cold and online snapshots
- Backups via plugin for cold and online snapshots
- Declarative backups for cold and online snapshots
- **Managed Roles**
- Creation and update of managed roles
- Password maintenance using Kubernetes secrets
- **Tablespaces** - Declarative creation of tablespaces - Declarative creation of temporary tablespaces - Backup / recovery from object storage - Backup / recovery from volume snapshots

1.4.46 Container Image Requirements

The CloudNativePG operator for Kubernetes is designed to work with any compatible container image of PostgreSQL that complies with the following requirements:

- PostgreSQL executables that must be in the path:
  - `initdb`
  - `postgres`
  - `pg_ctl`
  - `pg_controldata`
  - `pg_basebackup`
- Barman Cloud executables that must be in the path:
  - `barman-cloud-backup`
  - `barman-cloud-backup-delete`
  - `barman-cloud-backup-list`
  - `barman-cloud-check-wal-archive`
  - `barman-cloud-restore`
  - `barman-cloud-wal-archive`
  - `barman-cloud-wal-restore`
- PGAudit extension installed (optional - only if PGAudit is required in the deployed clusters)
- Appropriate locale settings

Important: Only PostgreSQL versions supported by the PGDG are allowed.

No entry point and/or command is required in the image definition, as CloudNativePG overrides it with its instance manager.

Warning: Application Container Images will be used by CloudNativePG in a **Primary with multiple/optional Hot Standby Servers Architecture** only.

The CloudNativePG community provides and supports public PostgreSQL container images

that work with CloudNativePG, and publishes them on ghcr.io .

Image Tag Requirements

To ensure the operator makes informed decisions, it must accurately detect the PostgreSQL major version. This detection can occur in two ways:

1. Utilizing the `major` field of the `imageCatalogRef` , if defined.
2. Auto-detecting the major version from the image tag of the `imageName` if not explicitly specified.

For auto-detection to work, the image tag must adhere to a specific format. It should commence with a valid PostgreSQL major version number (e.g., 15.6 or 16), optionally followed by a dot and the patch level.

Following this, the tag can include any character combination valid and accepted in a Docker tag, preceded by a dot, an underscore, or a minus sign.

Examples of accepted image tags:

- `12.1`
- `13.3.2.1-1`
- `13.4`
- `14`
- `15.5-10`
- `16.0`

Warning: *latest* is not considered a valid tag for the image.

Note: Image tag requirements do no apply for images defined in a catalog.

1.4.47 Operator capability levels

These capabilities were implemented by CloudNativePG, classified using the Operator SDK definition of Capability Levels

framework.

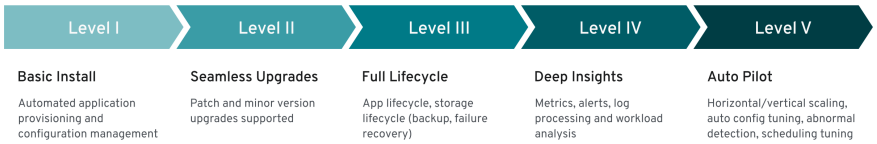


Fig. 18: Operator Capability Levels

Important: Based on the *Operator Capability Levels model* , you can expect a "Level V - Auto Pilot" set of capabilities from the CloudNativePG operator.

Each capability level is associated with a certain set of management features the operator offers:

1. Basic install

2. Seamless upgrades
3. Full lifecycle
4. Deep insights
5. Auto pilot

Note: We consider this framework as a guide for future work and implementations in the operator.

Level 1: Basic install

Capability level 1 involves installing and configuring the operator. This category includes usability and user experience enhancements, such as improvements in how you interact with the operator and a PostgreSQL cluster configuration.

Important: We consider information security part of this level.

Operator deployment via declarative configuration

The operator is installed in a declarative way using a Kubernetes manifest that defines four major `CustomResourceDefinition` objects: `Cluster` , `Pooler` , `Backup` , and `ScheduledBackup` .

PostgreSQL cluster deployment via declarative configuration

You define a PostgreSQL cluster (operand) using the `Cluster` custom resource in a fully declarative way. The PostgreSQL version is determined by the operand container image defined in the CR, which is automatically fetched from the requested registry. When deploying an operand, the operator also creates the following resources: `Pod` , `Service` , `Secret` , `ConfigMap` ,`PersistentVolumeClaim` , `PodDisruptionBudget` , `ServiceAccount` , `RoleBinding` , and `Role` .

Override of operand images through the CRD

The operator is designed to support any operand container image with PostgreSQL inside. By default, the operator uses the latest available minor version of the latest stable major version supported by the PostgreSQL community and published on ghcr.io. You can use any compatible image of PostgreSQL supporting the primary/standby architecture directly by setting the `imageName` attribute in the CR. The operator also supports `imagePullSecrets` to access private container registries, and it supports digests and tags for finer control of container image immutability. If you prefer not to specify an image name, you can leverage *image catalogs* by simply referencing the PostgreSQL major version. Moreover, image catalogs enable you to effortlessly create custom catalogs, directing to images based on your specific requirements.

Labels and annotations

You can configure the operator to support inheriting labels and annotations that are defined in a cluster's metadata. The goal is to improve the organization of the CloudNativePG deployment in your Kubernetes infrastructure.

Self-contained instance manager

Instead of relying on an external tool to coordinate PostgreSQL instances in the Kubernetes cluster pods, such as Patroni or Stolon, the operator injects the operator executable inside each pod, in a file named `/controller/manager`. The application is used to control the underlying PostgreSQL instance and to reconcile the pod status with the instance based on the PostgreSQL cluster topology. The instance manager also starts a web server that's invoked by the `kubelet` for probes. Unix signals invoked by the `kubelet` are filtered by the instance manager. Where appropriate, they're forwarded to the `postgres` process for fast and controlled reactions to external events. The instance manager is written in Go and has no external dependencies.

Storage configuration

Storage is a critical component in a database workload. Taking advantage of the Kubernetes native capabilities and resources in terms of storage, the operator gives you enough flexibility to choose the right storage for your workload requirements, based on what the underlying Kubernetes environment can offer. This implies choosing a particular storage class in a public cloud environment or fine-tuning the generated PVC through a PVC template in the CR's `storage` parameter.

For better performance and finer control, you can also choose to host your cluster's write-ahead log (WAL, also known as `pg_wal`) on a separate volume, preferably on different storage. The cnp-bench open source project can be used to benchmark both the storage and the database prior to production.

Replica configuration

The operator detects replicas in a cluster through a single parameter, called `instances`. If set to `1`, the cluster comprises a single primary PostgreSQL instance with no replica. If higher than `1`, the operator manages `instances -1` replicas, including high availability (HA) through automated failover and rolling updates through switchover operations.

CloudNativePG manages replication slots for all the replicas in the HA cluster. The implementation is inspired by the previously proposed patch for PostgreSQL, called failover slots, and also supports user defined physical replication slots on the primary.

Database configuration

The operator is designed to manage a PostgreSQL cluster with a single database. The operator transparently manages access to the database through three Kubernetes services provisioned and managed for read-write, read, and read-only workloads. Using the convention-over-configuration approach, the operator creates a database called `app`, by default owned by a regular Postgres user with the same name. You can specify both the database name and the user name, if required.

Although no configuration is required to run the cluster, you can customize both PostgreSQL runtime configuration and PostgreSQL host-based authentication rules in the `postgresql` section of the CR.

Configuration of Postgres roles, users, and groups

CloudNativePG supports *management of PostgreSQL roles, users, and groups through declarative configuration* using the `.spec.managed.roles` stanza.

Pod security policies

For InfoSec requirements, the operator doesn't require privileged mode for any container. It enforces a read-only root filesystem to guarantee containers immutability for both the operator and the operand pods. It also explicitly sets the required security contexts.

Affinity

The cluster's `affinity` section enables fine-tuning of how pods and related resources, such as persistent volumes, are scheduled across the nodes of a Kubernetes cluster. In particular, the operator supports:

- Pod affinity and anti-affinity
- Node selector
- Taints and tolerations

Topology spread constraints

The cluster's `topologySpreadConstraints` section enables additional control of scheduling pods across topologies, enhancing what affinity and anti-affinity can offer.

Command-line interface

CloudNativePG doesn't have its own command-line interface. It relies on the best command-line interface for Kubernetes, kubectl, by providing a plugin called `cnpg` . This plugin enhances and simplifies your PostgreSQL cluster management experience.

Current status of the cluster

The operator continuously updates the status section of the CR with the observed status of the cluster. The entire PostgreSQL cluster status is continuously monitored by the instance manager running in each pod. The instance manager is responsible for applying the required changes to the controlled PostgreSQL instance to converge to the required status of the cluster. (For example, if the cluster status reports that pod `-1` is the primary, pod `-1` needs to promote itself while the other pods need to follow pod `-1` .) The same status is used by the `cnpg` plugin for kubectl to provide details.

Operator's certification authority

The operator creates a certification authority for itself. It creates and signs with the operator certification authority a leaf certificate for the webhook server to use. This certificate ensures safe communication between the Kubernetes API server and the operator.

Cluster's certification authority

The operator creates a certification authority for every PostgreSQL cluster. This certification authority is used to issue and renew TLS certificates for clients' authentication, including streaming replication standby servers (instead of passwords). Support for a custom certification authority for client certificates is available through secrets, which also includes integration with cert-manager. Certificates can be issued with the `cnpg` plugin for kubectl.

TLS connections

The operator transparently and natively supports TLS/SSL connections to encrypt client/server communications for increased security using the cluster's certification authority. Support for custom server certificates is available through secrets, which also includes integration with cert-manager.

Certificate authentication for streaming replication

To authorize streaming replication connections from the standby servers, the operator relies on TLS client certificate authentication. This method is used instead of relying on a password (and therefore a secret).

Continuous configuration management

The operator enables you to apply changes to the `Cluster` resource YAML section of the PostgreSQL configuration. Depending on the configuration option, it also makes sure that all instances are properly reloaded or restarted.

Import of existing PostgreSQL databases

The operator provides a declarative way to import existing Postgres databases in a new CloudNativePG cluster in Kubernetes, using offline migrations. The same feature also covers offline major upgrades of PostgreSQL databases. Offline means that applications must stop their write operations at the source until the database is imported. The feature extends the `initdb` bootstrap method to create a new PostgreSQL cluster using a logical snapshot of the data available in another PostgreSQL database. This data can be accessed by way of the network through a superuser connection. Import is from any supported version of Postgres. It relies on `pg_dump` and `pg_restore` being executed from the new cluster primary for all databases that are part of the operation and, if requested, for roles.

PostGIS clusters

CloudNativePG supports the installation of clusters with the
PostGIS Container Images

open source extension for geographical databases. This extension is one of the most popular extensions for PostgreSQL.

Basic LDAP authentication for PostgreSQL

The operator allows you to configure LDAP authentication for your PostgreSQL clients, using either the *simple bind* or *search+bind* mode, as described in the LDAP authentication section of the PostgreSQL documentation .

Multiple installation methods

The operator can be installed through a Kubernetes manifest by way of `kubectl apply` , to be used in a traditional Kubernetes installation in public and private cloud environments. CloudNativePG also supports installation by way of a Helm chart or OLM bundle from OperatorHub.io.

Convention over configuration

The operator supports the convention-over-configuration paradigm, deciding standard default values while allowing you to override them and customize them. You can specify a deployment of a PostgreSQL cluster using the `Cluster` CRD in a couple of lines of YAML code.

Level 2: Seamless upgrades

Capability level 2 is about enabling updates of the operator and the actual workload, in this case PostgreSQL servers. This includes PostgreSQL minor release updates (security and bug fixes normally) as well as major online upgrades.

Upgrade of the operator

You can upgrade the operator seamlessly as a new deployment. Because of the instance manager' s injection, a change in the operator doesn' t require a change in the operand. The operator can manage older versions of the operand.

CloudNativePG also supports *In-place updates of the instance manager*

following an upgrade of the operator. In-place updates don' t require a rolling update (and subsequent switchover) of the cluster.

Upgrade of the managed workload

The operand can be upgraded using a declarative configuration approach as part of changing the CR and, in particular, the `imageName` parameter. The operator prevents major upgrades of PostgreSQL while making it possible to go in both directions in terms of minor PostgreSQL releases within a major version, enabling updates and rollbacks.

In the presence of standby servers, the operator performs rolling updates starting from the replicas. It does this by dropping the existing pod and creating a new one with the new requested operand image that reuses the underlying storage. Depending on the value of the `primaryUpdateStrategy` , the operator proceeds with a switchover before updating the former primary (`unsupervised`). Or, it waits for the user to manually issue the switchover procedure

(`supervised`) by way of the `cnpg` plugin for kubectl. The setting to use depends on the business requirements, as the operation might generate some downtime for the applications. This downtime can range from a few seconds to minutes, based on the actual database workload.

Display cluster availability status during upgrade

At any time, convey the cluster' s high availability status, for example, `Setting up primary` , `Creating a new replica` , `Cluster in healthy state` , `Switchover in progress` , `Failing over` , and `Upgrading cluster` .

Level 3: Full lifecycle

Capability level 3 requires the operator to manage aspects of business continuity and scalability.

- Disaster recovery* is a business continuity component that requires that both backup and recovery of a database work correctly. While as a starting point, the goal is to achieve RPO < 5 minutes, the long-term goal is to implement RPO=0 backup solutions. *High availability* is the other important component of business continuity. Through PostgreSQL native physical replication and hot standby replicas, it allows the operator to perform failover and switchover operations. This area includes enhancements in:
- Control of PostgreSQL physical replication, such as synchronous replication, (cascading) replication clusters, and so on
- Connection pooling, to improve performance and control through a connection pooling layer with pgBouncer

PostgreSQL WAL archive

The operator supports PostgreSQL continuous archiving of WAL files to an object store (AWS S3 and S3-compatible, Azure Blob Storage, Google Cloud Storage, and gateways like MinIO).

WAL archiving is defined at the cluster level, declaratively, through the `backup` parameter in the cluster definition. This is done by specifying an S3 protocol destination URL (for example, to point to a specific folder in an AWS S3 bucket) and, optionally, a generic endpoint URL.

WAL archiving, a prerequisite for continuous backup, doesn' t require any further user action. The operator transparently sets the `archive_command` to rely on `barman-cloud-wal-archive` to ship WAL files to the defined endpoint. You can decide the compression algorithm, as well as the number of parallel jobs to concurrently upload WAL files in the archive. In addition, `Instance Manager` checks the correctness of the archive destination by performing the `barman-cloud-check-wal-archive` command before beginning to ship the first set of WAL files.

PostgreSQL backups

The operator was designed to provide application-level backups using PostgreSQL' s native continuous hot backup technology based on physical base backups and continuous WAL archiving. Base backups can be saved on:

- Kubernetes volume snapshots
- Object stores (AWS S3 and S3-compatible, Azure Blob Storage, Google Cloud Storage, and gateways like MinIO)

Base backups are defined at the cluster level, declaratively, through the `backup` parameter in the cluster definition.

You can define base backups in two ways:

- On-demand, through the `Backup` custom resource definition

- Scheduled, through the `ScheduledBackup` custom resource definition, using a cron-like syntax

Volume snapshots rely directly on the Kubernetes API, which delegates this capability to the underlying storage classes and CSI drivers. Volume snapshot backups are suitable for very large database (VLDB) contexts.

Object store backups rely on `barman-cloud-backup` for the job (distributed as part of the application container image) to relay backups in the same endpoint, alongside WAL files.

Both `barman-cloud-wal-restore` and `barman-cloud-backup` are distributed in the application container image under GNU GPL 3 terms.

Object store backups and volume snapshot backups are taken while PostgreSQL is up and running (hot backups). Volume snapshots also support taking consistent database snapshots with cold backups.

Backups from a standby

The operator supports offloading base backups onto a standby without impacting the RPO of the database. This allows resources to be preserved on the primary, in particular I/O, for standard database operations.

Full restore from a backup

The operator enables you to bootstrap a new cluster (with its settings) starting from an existing and accessible backup, either on a volume snapshot or in an object store.

Once the bootstrap process is completed, the operator initiates the instance in recovery mode. It replays all available WAL files from the specified archive, exiting recovery and starting as a primary. Subsequently, the operator clones the requested number of standby instances from the primary. CloudNativePG supports parallel WAL fetching from the archive.

Point-in-time recovery (PITR) from a backup

The operator enables you to create a new PostgreSQL cluster by recovering an existing backup to a specific point in time, defined with a timestamp, a label, or a transaction ID. This capability is built on top of the full restore one and supports all the options available in PostgreSQL for PITR .

Zero-data-loss clusters through synchronous replication

Achieve *zero data loss* (RPO=0) in your local high-availability CloudNativePG cluster through quorum-based synchronous replication support. The operator provides two configuration options that control the minimum and maximum number of expected synchronous standby replicas available at any time. The operator reacts accordingly, based on the number of available and ready PostgreSQL instances in the cluster. It uses the following formula for the quorum (`q`):

```
1 <= minSyncReplicas <= q <= maxSyncReplicas <= readyReplicas
```

Replica clusters

Establish a robust cross-Kubernetes cluster topology for PostgreSQL clusters, harnessing the power of native streaming and cascading replication. With the `replica` option, you can configure an autonomous cluster to consistently replicate data from another PostgreSQL source of the same major version. This source can be located anywhere, provided you have access to a WAL archive for fetching WAL files or a direct streaming connection via TLS between the two endpoints.

Notably, the source PostgreSQL instance can exist outside the Kubernetes environment, whether in a physical or virtual setting.

Replica clusters can be instantiated through various methods, including volume snapshots, a recovery object store (utilizing the Barman Cloud backup format), or streaming using `pg_basebackup` . Both WAL file shipping and WAL streaming are supported. The deployment of replica clusters significantly elevates the business continuity posture of PostgreSQL databases within Kubernetes, extending across multiple data centers and facilitating hybrid and multi-cloud setups. (While anticipating Kubernetes federation native capabilities, manual switchover across data centers remains necessary.)

Additionally, the flexibility extends to creating delayed replica clusters intentionally lagging behind the primary cluster. This intentional lag aims to minimize the Recovery Time Objective (RTO) in the event of unintended errors, such as incorrect `DELETE` or `UPDATE` SQL operations.

Tablespace support

CloudNativePG seamlessly integrates robust support for PostgreSQL tablespaces by facilitating the declarative definition of individual persistent volumes. This innovative feature empowers you to efficiently distribute I/O operations across a diverse array of storage devices. Through the transparent orchestration of tablespaces, CloudNativePG enhances the performance and scalability of PostgreSQL databases, ensuring a streamlined and optimized experience for managing large scale data storage in cloud-native environments. Support for temporary tablespaces is also included.

Liveness and readiness probes

The operator defines liveness and readiness probes for the Postgres containers that are then invoked by the kubelet. They' re mapped respectively to the `/healthz` and `/readyz` endpoints of the web server managed directly by the instance manager.

The liveness probe is based on the `pg_isready` executable, and the pod is considered healthy with exit codes 0 (server accepting connections normally) and 1 (server is rejecting connections, for example, during startup). The readiness probe issues a simple query (`;`) to verify that the server is ready to accept connections.

Rolling deployments

The operator supports rolling deployments to minimize the downtime. If a PostgreSQL cluster is exposed publicly, the service load-balances the read-only traffic only to available pods during the initialization or the update.

Scale up and down of replicas

The operator allows you to scale up and down the number of instances in a PostgreSQL cluster. New replicas are started up from the primary server and participate in the cluster's HA infrastructure. The CRD declares a "scale" subresource that allows you to use the `kubectl scale` command.

Maintenance window and PodDisruptionBudget for Kubernetes nodes

The operator creates a `PodDisruptionBudget` resource to limit the number of concurrent disruptions to one primary instance. This configuration prevents the maintenance operation from deleting all the pods in a cluster, allowing the specified number of instances to be created. The PodDisruptionBudget is applied during the node-draining operation, preventing any disruption of the cluster service.

While this strategy is correct for Kubernetes clusters where storage is shared among all the worker nodes, it might not be the best solution for clusters using local storage or for clusters installed in a private cloud. The operator allows you to specify a maintenance window and configure the reaction to any underlying node eviction. The `ReusePVC` option in the maintenance window section enables to specify the strategy to use. Allocate new storage in a different PVC for the evicted instance, or wait for the underlying node to be available again.

Fencing

Fencing is the process of protecting the data in one, more, or even all instances of a PostgreSQL cluster when they appear to be malfunctioning. When an instance is fenced, the PostgreSQL server process is guaranteed to be shut down, while the pod is kept running. This ensures that, until the fence is lifted, data on the pod isn't modified by PostgreSQL and that you can investigate file system for debugging and troubleshooting purposes.

Hibernation (declarative)

CloudNativePG supports *hibernation of a running PostgreSQL cluster*

in a declarative manner, through the `cnpg.io/hibernation` annotation. Hibernation enables saving CPU power by removing the database pods while keeping the database PVCs. This feature simulates scaling to 0 instances.

Hibernation (imperative)

CloudNativePG supports *hibernation of a running PostgreSQL cluster*

by way of the `cnpg` plugin. Hibernation shuts down all Postgres instances in the high-availability cluster and keeps a static copy of the PVC group of the primary. The copy contains `PGDATA` and WALs. The plugin enables you to exit the hibernation phase by resuming the primary and then recreating all the replicas, if they exist.

Reuse of persistent volumes storage in pods

When the operator needs to create a pod that was deleted by the user or was evicted by a Kubernetes maintenance operation, it reuses the `PersistentVolumeClaim`, if available. This ability avoids the need to clone the data from the primary again.

CPU and memory requests and limits

The operator allows administrators to control and manage resource usage by the cluster's pods in the `resources` section of the manifest. In particular, you can set `requests` and `limits` values for both CPU and RAM.

Connection pooling with PgBouncer

CloudNativePG provides native support for connection pooling with

PgBouncerIntegrationStatus {postgresql-cnpg-io-v1-PgBouncerIntegrationStatus}, one of the most popular open source connection poolers

for PostgreSQL. From an architectural point of view, the native implementation of a PgBouncer connection pooler introduces a new layer to access the database. This optimizes the query flow toward the instances and makes the use of the underlying PostgreSQL resources more efficient. Instead of connecting directly to a PostgreSQL service, applications can now connect to the PgBouncer service and start reusing any existing connection.

Level 4: Deep insights

Capability level 4 is about *observability*: monitoring, alerting, trending, and log processing. This might involve the use of external tools, such as Prometheus, Grafana, and Fluent Bit, as well as extensions in the PostgreSQL engine for the output of error logs directly in JSON format.

CloudNativePG was designed to provide everything needed to easily integrate with industry-standard and community-accepted tools for flexible monitoring and logging.

Prometheus exporter with configurable queries

The instance manager provides a pluggable framework. By way of its own web server listening on the `metrics` port (9187), it exposes an endpoint to export metrics for the Prometheus monitoring and alerting tool. The operator supports custom monitoring queries defined as `ConfigMap` or `Secret` objects using a syntax that's compatible with postgres_exporter. CloudNativePG provides a set of basic monitoring queries for PostgreSQL that can be integrated and adapted to your context.

Grafana dashboard

CloudNativePG comes with a Grafana dashboard that you can use as a base to monitor all critical aspects of a PostgreSQL cluster, and customize.

Standard output logging of PostgreSQL error messages in JSON format

Every log message is delivered to standard output in JSON format. The first level is the definition of the timestamp, the log level, and the type of log entry, such as `postgres` for the canonical PostgreSQL error message channel. As a result, every pod managed by CloudNativePG can be easily and directly integrated with any downstream log processing stack that supports JSON as source data type.

Real-time query monitoring

CloudNativePG transparently and natively supports:

- The essential pg_stat_statements , which enables tracking of planning and execution statistics of all SQL statements executed by a PostgreSQL server
- The auto_explain , which provides a means for logging execution plans of slow statements automatically, without having to manually run `EXPLAIN` (helpful for tracking down un-optimized queries)
- The pg_failover_slots , which makes logical replication slots usable across a physical failover, ensuring resilience in change data capture (CDC) contexts based on PostgreSQL' s native logical replication

Audit

CloudNativePG allows database and security administrators, auditors, and operators to track and analyze database activities using PGAudit for PostgreSQL. Such activities flow directly in the JSON log and can be properly routed to the correct downstream target using common log brokers like Fluentd.

Kubernetes events

Record major events as expected by the Kubernetes API, such as creating resources, removing nodes, and upgrading. Events can be displayed by using the `kubectl describe` and `kubectl get events` commands.

Level 5: Auto pilot

Capability level 5 is focused on automated scaling, healing, and tuning through the discovery of anomalies and insights that emerged from the observability layer.

Automated failover for self-healing

In case of detected failure on the primary, the operator changes the status of the cluster by setting the most aligned replica as the new target primary. As a consequence, the instance manager in each alive pod initiates the required procedures to align itself with the requested status of the cluster. It does this by either becoming the new primary or by following it. In case the former primary comes back up, the same mechanism avoids a split-brain by preventing applications from reaching it, running `pg_rewind` on the server and restarting it as a standby.

Automated recreation of a standby

If the pod hosting a standby is removed, the operator initiates the procedure to re-create a standby server.

1.4.48 Custom Pod Controller

Kubernetes uses the *Controller pattern*

to align the current cluster state with the desired one.

Stateful applications are usually managed with the *StatefulSet*

controller, which creates and reconciles a set of Pods built from the same specification, and assigns them a sticky identity.

CloudNativePG implements its own custom controller to manage PostgreSQL instances, instead of relying on the `StatefulSet` controller. While bringing more complexity to the implementation, this design choice provides the operator with more flexibility on how we manage the cluster, while being transparent on the topology of PostgreSQL clusters.

Like many choices in the design realm, different ones lead to other compromises. The following sections discuss a few points where we believe this design choice has made the implementation of CloudNativePG more reliable, and easier to understand.

PVC resizing

This is a well known limitation of `StatefulSet` : it does not support resizing PVCs. This is inconvenient for a database. Resizing volumes requires convoluted workarounds.

In contrast, CloudNativePG leverages the configured storage class to manage the underlying PVCs directly, and can handle PVC resizing if the storage class supports it.

Primary Instances versus Replicas

The `StatefulSet` controller is designed to create a set of Pods from just one template. Given that we use one `Pod` per PostgreSQL instance, we have two kinds of Pods:

1. primary instance (only one)
2. replicas (multiple, optional)

This difference is relevant when deciding the correct deployment strategy to execute for a given operation.

Some operations should be performed on the replicas first, and then on the primary, but only after an updated replica is promoted as the new primary. For example, when you want to apply a different PostgreSQL image version, or when you increase configuration parameters like `max_connections` (which are treated specially by PostgreSQL because CloudNativePG uses hot standby replicas).

While doing that, CloudNativePG considers the PostgreSQL instance' s role - and not just its serial number.

Sometimes the operator needs to follow the opposite process: work on the primary first and then on the replicas. For example, when you lower `max_connections` . In that case, CloudNativePG will:

- apply the new setting to the primary instance
- restart it
- apply the new setting on the replicas

The `StatefulSet` controller, being application-independent, can't incorporate this behavior, which is specific to PostgreSQL's native replication technology.

Coherence of PVCs

PostgreSQL instances can be configured to work with multiple PVCs: this is how WAL storage can be separated from `PGDATA`.

The two data stores need to be coherent from the PostgreSQL point of view, as they're used simultaneously. If you delete the PVC corresponding to the WAL storage of an instance, the PVC where `PGDATA` is stored will not be usable anymore.

This behavior is specific to PostgreSQL and is not implemented in the `StatefulSet` controller - the latter not being application specific.

After the user dropped a PVC, a `StatefulSet` would just recreate it, leading to a corrupted PostgreSQL instance.

CloudNativePG would instead classify the remaining PVC as unusable, and start creating a new pair of PVCs for another instance to join the cluster correctly.

Local storage, remote storage, and database size

Sometimes you need to take down a Kubernetes node to do an upgrade. After the upgrade, depending on your upgrade strategy, the updated node could go up again, or a new node could replace it.

Supposing the unavailable node was hosting a PostgreSQL instance, depending on your database size and your cloud infrastructure, you may prefer to choose one of the following actions:

1. drop the PVC and the Pod residing on the downed node; create a new PVC cloning the data from another PVC; after that, schedule a Pod for it
2. drop the Pod, schedule the Pod in a different node, and mount the PVC from there
3. leave the Pod and the PVC as they are, and wait for the node to be back up.

The first solution is practical when your database size permits, allowing you to immediately bring back the desired number of replicas.

The second solution is only feasible when you're not using the storage of the local node, and re-mounting the PVC in another host is possible in a reasonable amount of time (which only you and your organization know).

The third solution is appropriate when the database is big and uses local node storage for maximum performance and data durability.

The CloudNativePG controller implements all these strategies so that the user can select the preferred behavior at the cluster level (read the *Kubernetes Upgrade and Maintenance* section for details).

Being generic, the `StatefulSet` doesn't allow this level of customization.

1.4.49 Examples

The examples show configuration files for setting up your PostgreSQL cluster.

Important: These examples are for demonstration and experimentation purposes. You can execute them on a personal Kubernetes cluster with Minikube or Kind, as described in *Quick start* .

See also:

For a list of available options, see *API reference* .

Basics

Basic cluster :

A basic example of a cluster.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3

  storage:
    size: 1Gi
```

Custom cluster :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-custom
spec:
  instances: 3

  # Parameters and pg_hba configuration will be append
  # to the default ones to make the cluster work
  postgresql:
    parameters:
      max_worker_processes: "60"
    pg_hba:
      # To access through TCP/IP you will need to get username
      # and password from the secret cluster-example-custom-app
      - host all all all md5

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised
```

(continues on next page)

(continued from previous page)

```
  # Require 1Gi of space per instance using default storage class
  storage:
    size: 1Gi
```

A basic cluster that uses the default storage class and custom parameters for the `postgresql.conf` and `pg_hba.conf` files.

Cluster with customized storage class :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-class
spec:
  instances: 3

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi
```

: A basic cluster that uses a specified storage class of `standard` .

Cluster with persistent volume claim (PVC) template configured :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-pvc-template
spec:
  instances: 3

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    size: 1Gi
    pvcTemplate:
      accessModes:
        - ReadWriteOnce
```

(continues on next page)

(continued from previous page)

```
      resources:
        requests:
          storage: 1Gi
      storageClassName: standard
      volumeMode: Filesystem
```

: A basic cluster with an explicit persistent volume claim template.

Extended configuration example :

```
#  Example of definition of a test cluster using all the elements available
#  in the CRD. Please change values appropriately for your environment.
#  Remember that you can take advantage of convention over configuration
#  and normally you dont need to use all these definitions.

apiVersion: v1
data:
  password:␣
↪VHhWZVE0bk44MlNTaVlIb3N3cU9VUlp2UURhTDRLcE5FbHNDRUVlOWJ3RHhNZDczS2NrSWVYelM1Y1U2TGlDMg==
  username: YXBw
kind: Secret
metadata:
  name: cluster-example-app-user
type: kubernetes.io/basic-auth
- --
apiVersion: v1
data:
  password:␣
↪dU4zaTFIaDBiWWJDYzRUeVZBYWNCaG1TemdxdHpxeG1PVmpBbjBRSUNoc0pyU211OVBZMmZ3MnE4RUtLTHBaOQ==
  username: cG9zdGdyZXM=
kind: Secret
metadata:
  name: cluster-example-superuser
type: kubernetes.io/basic-auth
- --
apiVersion: v1
kind: Secret
metadata:
  name: backup-creds
data:
  ACCESS_KEY_ID: a2V5X2lk
  ACCESS_SECRET_KEY: c2VjcmV0X2tleQ==
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-full
spec:
  description: "Example of cluster"
  imageName: ghcr.io/cloudnative-pg/postgresql:16.2
  # imagePullSecret is only required if the images are located in a private registry
  # imagePullSecrets:
```

(continues on next page)

(continued from previous page)

```
#   - name: private_registry_access
instances: 3
startDelay: 300
stopDelay: 300
primaryUpdateStrategy: unsupervised

postgresql:
  parameters:
    shared_buffers: 256MB
    pg_stat_statements.max: 10000
    pg_stat_statements.track: all
    auto_explain.log_min_duration: 10s
  pg_hba:
    - host all all 10.244.0.0/16 md5

bootstrap:
  initdb:
    database: app
    owner: app
    secret:
      name: cluster-example-app-user
  # Alternative bootstrap method: start from a backup
  #recovery:
  #  backup:
  #    name: backup-example

enableSuperuserAccess: true
superuserSecret:
  name: cluster-example-superuser

storage:
  storageClass: standard
  size: 1Gi

backup:
  barmanObjectStore:
    destinationPath: s3://cluster-example-full-backup/
    endpointURL: http://custom-endpoint:1234
    s3Credentials:
      accessKeyId:
        name: backup-creds
        key: ACCESS_KEY_ID
      secretAccessKey:
        name: backup-creds
        key: ACCESS_SECRET_KEY
    wal:
      compression: gzip
      encryption: AES256
    data:
      compression: gzip
      encryption: AES256
      immediateCheckpoint: false
```

(continues on next page)

(continued from previous page)

```
        jobs: 2
    retentionPolicy: "30d"

  resources:
    requests:
      memory: "512Mi"
      cpu: "1"
    limits:
      memory: "1Gi"
      cpu: "2"

  affinity:
    enablePodAntiAffinity: true
    topologyKey: failure-domain.beta.kubernetes.io/zone

  nodeMaintenanceWindow:
    inProgress: false
    reusePVC: false
```

: A cluster that sets most of the available options.

Bootstrap cluster with SQL files :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: post-init-sql-configmap
data:
  configmap.sql: |
    create table configmaps (i integer);
    insert into configmaps (select generate_series(1,10000));
- --
apiVersion: v1
kind: Secret
metadata:
  name: post-init-sql-secret
stringData:
  secret.sql: |
    create table secrets (i integer);
    insert into secrets (select generate_series(1,10000));
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    initdb:
      database: appdb
      owner: appuser
```

(continues on next page)

(continued from previous page)

```
      postInitSQL:
        - create table numbers (i integer)
        - insert into numbers (select generate_series(1,10000))
      postInitTemplateSQL:
        - create extension intarray
      postInitApplicationSQL:
        - create table application_numbers (i integer)
        - insert into application_numbers (select generate_series(1,10000))
      postInitApplicationSQLRefs:
        configMapRefs:
        - name: post-init-sql-configmap
          key: configmap.sql
        secretRefs:
        - name: post-init-sql-secret
          key: secret.sql

  storage:
    size: 1Gi
```

: A cluster example that executes a set of queries defined in a secret and a `ConfigMap` right after the database is created.

Sample cluster with customized ``pg_hba`` configuration :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  postgresql:
    pg_hba:
      - hostssl app all all cert

  storage:
    size: 1Gi
```

: A basic cluster that enables the user app to authenticate using certificates.

Sample cluster with Secret and ConfigMap mounted using projected volume template :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-projected-volume
spec:
  instances: 3
  projectedVolumeTemplate:
    sources:
      - secret:
          name: sample-secret
          items:
            - key: tls.crt
              path: certificate/tls.crt
```

(continues on next page)

(continued from previous page)

```
            - key: tls.key
              path: certificate/tls.key
      - configMap:
          name: sample-configmap
          items:
            - key: key1
              path: config/key1
            - key: key2
              path: config/key2
  storage:
    size: 1Gi

- --
apiVersion: v1
data:
  tls.crt:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNQekNDQWVXZ0F3SUJBZ0lRSHVDY2lKcDVkZis0dHZBcDBrTk9rekFLQmdncW
  tls.key:␣
↪LS0tLS1CRUdJTiBFQyBQUklWQVRFIEtFWS0tLS0tCk1IY0NBUUVFSUtucWxVQjFqU2sxWW5VZzAyb0tGbXlRdDJLUEZwaFc0K1lmQU
kind: Secret
metadata:
  name: sample-secret
type: kubernetes.io/tls
- --
apiVersion: v1
data:
  key1: value1
  key2: value2
  key3: value3
kind: ConfigMap
metadata:
  name: sample-configmap
```

A basic cluster with the existing `Secret` and `ConfigMap` mounted into Postgres pod using projected volume mount.

Backups

Customized storage class and backups : *Prerequisites*: Bucket storage must be available. The sample config is for AWS. Change it to suit your setup. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg-backup
spec:
  instances: 3

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
```

(continues on next page)

(continued from previous page)

```
  #                the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi

  # Backup properties
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/path/to/folder
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

A cluster with backups configured.

Backup : *Prerequisites*:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg-backup
spec:
  instances: 3

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #                the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi

  # Backup properties
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/path/to/folder
      s3Credentials:
        accessKeyId:
          name: aws-creds
```

(continues on next page)

(continued from previous page)

```
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

applied and healthy. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: pg-backup-example
spec:
  cluster:
    name: pg-backup
```

: An example of a backup that runs against the previous sample.

Simple cluster with backup configured : *Prerequisites*: The configuration assumes minio is running and working. Update `backup.barmanObjectStore` with your minio parameters or your cloud solution. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-backup
spec:
  instances: 3
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: csi-hostpath-sc
    size: 1Gi

  # Backup properties
  # This assumes a local minio setup
  backup:
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio:9000
      s3Credentials:
        accessKeyId:
          name: minio
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
      data:
        additionalCommandArgs:
          - "--min-chunk-size=5MB"
```

(continues on next page)

(continued from previous page)

```
          - "--read-timeout=60"
          - "-vv"
```

A basic cluster with backups configured.

Replica clusters

Replica cluster by way of backup from an object store : *Prerequisites*:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg-backup
spec:
  instances: 3

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi

  # Backup properties
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/path/to/folder
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

applied and healthy, and a backup

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: cluster-example-trigger-backup
spec:
  cluster:
    name: cluster-example-with-backup
```

applied and completed. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-replica-from-backup-simple
spec:
  instances: 1

  bootstrap:
    recovery:
      source: cluster-example-backup

  replica:
    enabled: true
    source: cluster-example-backup

  storage:
    size: 1Gi

  externalClusters:
  - name: cluster-example-backup
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio:9000
      s3Credentials:
        accessKeyId:
          name: minio
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio
          key: ACCESS_SECRET_KEY
```

: A replica cluster following a cluster with backup configured.

Replica cluster by way of volume snapshot : *Prerequisites*:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-volume-snapshot
spec:
  instances: 3
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: csi-hostpath-sc
    size: 1Gi
  walStorage:
    storageClass: csi-hostpath-sc
    size: 1Gi

  # Backup properties
```

(continues on next page)

(continued from previous page)

```
  backup:
    volumeSnapshot:
       className: csi-hostpath-snapclass
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio:9000
      s3Credentials:
        accessKeyId:
          name: minio
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

applied and healthy, and a volume snapshot

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: backup-with-volume-snapshot
spec:
  method: volumeSnapshot
  cluster:
    name: cluster-example-with-volume-snapshot
```

applied and completed. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-replica-from-snapshot
spec:
  instances: 1

  storage:
    storageClass: csi-hostpath-sc
    size: 1Gi
  walStorage:
    storageClass: csi-hostpath-sc
    size: 1Gi

  bootstrap:
    recovery:
      source: cluster-example-with-volume-snapshot
      volumeSnapshots:
        storage:
          name: cluster-example-with-volume-snapshot-2-1692618163
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io
        walStorage:
```

(continues on next page)

(continued from previous page)

```
          name: cluster-example-with-volume-snapshot-2-wal-1692618163
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io

  replica:
    enabled: true
    source: cluster-example-with-volume-snapshot

  externalClusters:
    - name: cluster-example-with-volume-snapshot

      connectionParameters:
        host: cluster-example-with-volume-snapshot-rw.default.svc
        user: postgres
        dbname: postgres
      password:
        name: cluster-example-with-volume-snapshot-superuser
        key: password

      barmanObjectStore:
        destinationPath: s3://backups/
        endpointURL: http://minio:9000
        s3Credentials:
          accessKeyId:
            name: minio
            key: ACCESS_KEY_ID
          secretAccessKey:
            name: minio
            key: ACCESS_SECRET_KEY
        wal:
          maxParallel: 8
```

: A replica cluster following a cluster with volume snapshot configured.

Replica cluster by way of streaming (pg_basebackup) : *Prerequisites*:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3

  storage:
    size: 1Gi
```

applied and healthy. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-replica-example
spec:
```

(continues on next page)

(continued from previous page)

```
  instances: 1

  bootstrap:
    pg_basebackup:
      source: cluster-example

  replica:
    enabled: true
    source: cluster-example

  storage:
    size: 1Gi
  # note the namespace default in the host name `cluster-example-rw.default.svc`
  # remember to change accordingly with the namespace of the main cluster
  externalClusters:
  - name: cluster-example
    connectionParameters:
      host: cluster-example-rw.default.svc
      user: streaming_replica
      sslmode: verify-full
      dbname: postgres
    # NOTE: if this cluster is created in a different namespace than the main cluster
    # remember to create the `-replication` and `-ca` secrets in the follower namespace
    # before creating the follower cluster
    sslKey:
      name: cluster-example-replication
      key: tls.key
    sslCert:
      name: cluster-example-replication
      key: tls.crt
    sslRootCert:
      name: cluster-example-ca
      key: ca.crt
```

: A replica cluster following `cluster-example` with streaming replication.

PostGIS

PostGIS example :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgis-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgis:14
  bootstrap:
    initdb:
      postInitTemplateSQL:
        - CREATE EXTENSION postgis;
```

(continues on next page)

(continued from previous page)

```
        - CREATE EXTENSION postgis_topology;
        - CREATE EXTENSION fuzzystrmatch;
        - CREATE EXTENSION postgis_tiger_geocoder;

  storage:
    size: 1Gi
```

: An example of a PostGIS cluster. See *PostGIS Container Images* for details.

Managed roles

Cluster with declarative role management :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-roles
spec:
  instances: 3
  storage:
    size: 1Gi

  managed:
    roles:
    - name: app
      createdb: true
      login: true
    - name: dante
      ensure: present
      comment: my database-side comment
      login: true
      superuser: false
      createdb: true
      createrole: false
      inherit: false
      replication: false
      bypassrls: false
      connectionLimit: 4
      validUntil: "2053-04-12T15:04:05Z"
      inRoles:
        - pg_monitor
        - pg_signal_backend
      passwordSecret:
        name: cluster-example-dante
- --
apiVersion: v1
data:
  username: ZGFudGU=
  password: ZGFudGU=
kind: Secret
metadata:
```

(continues on next page)

(continued from previous page)

```
  name: cluster-example-dante
type: kubernetes.io/basic-auth
```

: Declares a role with the `managed` stanza. Includes password management with Kubernetes secrets.

Declarative tablespaces

Cluster with declarative tablespaces :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-tablespaces
spec:
  instances: 3

  storage:
    size: 1Gi

  tablespaces:
    - name: atablespace
      storage:
        size: 1Gi
        storageClass: standard
      temporary: true
    - name: another_tablespace
      storage:
        size: 2Gi
        storageClass: standard
      temporary: true
    - name: tablespacea1
      storage:
        size: 2Gi
        storageClass: standard
```

Cluster with declarative tablespaces and backup : *Prerequisites*: The configuration assumes minio is running and working. Update `backup.barmanObjectStore` with your minio parameters or your cloud solution. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-tablespaces
spec:
  instances: 3

  storage:
    size: 1Gi

  backup:
    barmanObjectStore:
      destinationPath: s3://backups/
```

(continues on next page)

(continued from previous page)

```
      endpointURL: http://minio:9000
      s3Credentials:
        accessKeyId:
          name: minio
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip

  tablespaces:
    - name: atablespace
      storage:
        size: 1Gi
        storageClass: standard
    - name: another_tablespace
      storage:
        size: 2Gi
        storageClass: standard
    - name: tablespacea1
      storage:
        size: 2Gi
        storageClass: standard
```

Restored cluster with tablespaces from object store : *Prerequisites*: The previous cluster applied and a base backup completed. Remember to update `bootstrap.recovery.backup.name` with the backup name. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-restore-with-tablespaces
spec:
  instances: 3

  storage:
    size: 1Gi

  bootstrap:
    recovery:
      backup:
        name: cluster-example-with-tablespaces-20231128093940

  tablespaces:
    atablespace:
      storage:
        size: 1Gi
        storageClass: standard
    another_tablespace:
      storage:
        size: 2Gi
        storageClass: standard
```

(continues on next page)

(continued from previous page)

```
    tablespacea1:
      storage:
        size: 2Gi
        storageClass: standard
```

For a list of available options, see *API reference* .

Pooler configuration

Pooler with custom service config :

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example
  instances: 3
  type: rw
  serviceTemplate:
    metadata:
      labels:
        app: pooler
    spec:
      type: LoadBalancer
  pgbouncer:
    poolMode: session
    parameters:
      max_client_conn: "1000"
      default_pool_size: "10"
```

1.4.50 Networking

CloudNativePG assumes the underlying Kubernetes cluster has the required connectivity already set up. Networking on Kubernetes is an important and extended topic; please refer to the *Kubernetes documentation* for further information.

If you' re following the quickstart guide to install CloudNativePG on a local KinD or K3d cluster, you should not encounter any networking issues as neither platform will add any networking restrictions by default.

However, when deploying CloudNativePG on existing infrastructure, networking restrictions might be in place that could impair the communication of the operator with PostgreSQL clusters. Specifically, existing Network Policies

might restrict certain types of traffic.

Or, you might be interested in adding network policies in your environment for increased security. As mentioned in the *security document* , please ensure the operator can reach every cluster pod on ports 8000 and 5432, and that pods can connect to each other.

Cross-namespace network policy for the operator

Following the quickstart guide or using helm chart for deployment will install the operator in a dedicated namespace (`cnpg-system` by default). We recommend that you create clusters in a different namespace.

The operator *must* be able to connect to cluster pods. This might be precluded if there is a `NetworkPolicy` restricting cross-namespace traffic.

For example, the *kubernetes guide on network policies*

contains an example policy denying all ingress traffic by default.

If your local kubernetes setup has this kind of restrictive network policy, you will need to create a `NetworkPolicy` to explicitly allow connection from the operator namespace and pod to the cluster namespace and pods. You can find an example in the file in this repository.

```
#  This network policy allows ingress from the CNPG operator

#  installed in namespace cnpg-system into the pods corresponding

#  to a cluster named cluster-example

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-operator
spec:
  podSelector:
    matchLabels:
      cnpg.io/cluster: cluster-example # The label value must be the cluster name
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: cnpg-system # Namespace where the operator is␣
↪deployed
          podSelector:
            matchLabels:
              app.kubernetes.io/name: cloudnative-pg # Matches the Operator pod
      ports:
        - port: 8000
        - port: 5432
```

Please note, you'll need to adjust the cluster name and cluster namespace to match your specific setup, and also the operator namespace if it is not the default namespace.

Cross-cluster networking

While *Error while bootstrapping the data directory* from another cluster or when using the `externalClusters` section, ensure connectivity among all clusters, object stores, and namespaces involved.

Again, we refer you to the *Kubernetes documentation*

for setup information.

1.4.51 Benchmarking

The CNPG kubectl plugin provides an easy way for benchmarking a PostgreSQL deployment in Kubernetes using CloudNativePG.

Benchmarking is focused on two aspects:

- the **database**, by relying on pgbench
- the **storage**, by relying on fio

Important: *pgbench* and *fio* must be run in a staging or pre-production environment. Do not use these plugins in a production environment, as it might have catastrophic consequences on your databases and the other workloads/applications that run in the same shared environment.

pgbench

The `kubectl` CNPG plugin command `pgbench` executes a user-defined `pgbench` job against an existing Postgres Cluster.

Through the `--dry-run` flag you can generate the manifest of the job for later modification/execution.

A common command structure with `pgbench` is the following:

```
kubectl cnpg pgbench \
  -n <namespace> <cluster-name> \
  --job-name <pgbench-job> \
  --db-name <db-name> \
  -- <pgbench options>
```

Important: Please refer to the pgbench

for information about the specific options to be used in your jobs.

This example creates a job called `pgbench-init` that initializes for `pgbench` OLTP-like purposes the `app` database in a `Cluster` named `cluster-example` , using a scale factor of 1000:

```
kubectl cnpg pgbench \
  --job-name pgbench-init \
  cluster-example \
  -- --initialize --scale 1000
```

Note: This will generate a database with 100000000 records, taking approximately 13GB of space on disk.

You can see the progress of the job with:

```
kubectl logs jobs/pgbench-run
```

The following example creates a job called `pgbench-run` executing `pgbench` against the previously initialized database for 30 seconds, using a single connection:

```
kubectl cnpg pgbench \
  --job-name pgbench-run \
  cluster-example \
  -- --time 30 --client 1 --jobs 1
```

The next example runs `pgbench` against an existing database by using the `--db-name` flag and the `pgbench` namespace:

```
kubectl cnpg pgbench \
  --db-name pgbench \
  --job-name pgbench-job \
  cluster-example \
  -- --time 30 --client 1 --jobs 1
```

If you want to run a `pgbench` job on a specific worker node, you can use the `--node-selector` option. Suppose you want to run the previous initialization job on a node having the `workload=pgbench` label, you can run:

```
kubectl cnpg pgbench \
  --db-name pgbench \
  --job-name pgbench-init \
  --node-selector workload=pgbench \
  cluster-example \
  -- --initialize --scale 1000
```

The job status can be fetched by running:

```
kubectl get job/pgbench-job -n <namespace>

NAME       COMPLETIONS   DURATION   AGE
job-name   1/1           15s        41s
```

Once the job is completed the results can be gathered by executing:

```
kubectl logs job/pgbench-job -n <namespace>
```

fio

The kubectl CNPG plugin command `fio` executes a fio job with default values and read operations. Through the `--dry-run` flag you can generate the manifest of the job for later modification/execution.

Note: The kubectl plugin command *fio* will create a deployment with predefined fio job values using a ConfigMap. If you want to provide custom job values, we recommend generating a manifest using the *--dry-run* flag and providing your custom job values in the generated ConfigMap.

Example of default usage:

```
kubectl cnpg fio <fio-name>
```  Example with custom values:

```shell
kubectl cnpg fio <fio-name> \
  -n <namespace>  \
  --storageClass <name> \
  --pvcSize <size>
```

Example of how to run the `fio` command against a `StorageClass` named `standard` and `pvcSize: 2Gi` in the `fio` namespace:

```
kubectl cnpg fio fio-job \
  -n fio  \
  --storageClass standard \
  --pvcSize 2Gi
```

The deployment status can be fetched by running:

```
kubectl get deployment/fio-job -n fio

NAME          READY   UP-TO-DATE   AVAILABLE   AGE
fio-job        1/1     1            1           14s
```

After running kubectl plugin command `fio` .

It will:

1. Create a PVC
2. Create a ConfigMap representing the configuration of a fio job
3. Create a fio deployment composed by a single Pod, which will run fio on the PVC, create graphs after completing the benchmark and start serving the generated files with a webserver. We use the *fio-tools <https://github.com/wallnerryan/fio-tools>`_* image for that.

The Pod created by the deployment will be ready when it starts serving the results. You can forward the port of the pod created by the deployment

```
kubectl port-forward -n <namespace> deployment/<fio-name> 8000
```

and then use a browser and connect to `http://localhost:8000/` to get the data.

The default 8k block size has been chosen to emulate a PostgreSQL workload. Disks that cap the amount of available IOPS can show very different throughput values when changing this parameter.

Below is an example diagram of sequential writes on a local disk mounted on a dedicated Kubernetes node (1 hour benchmark):

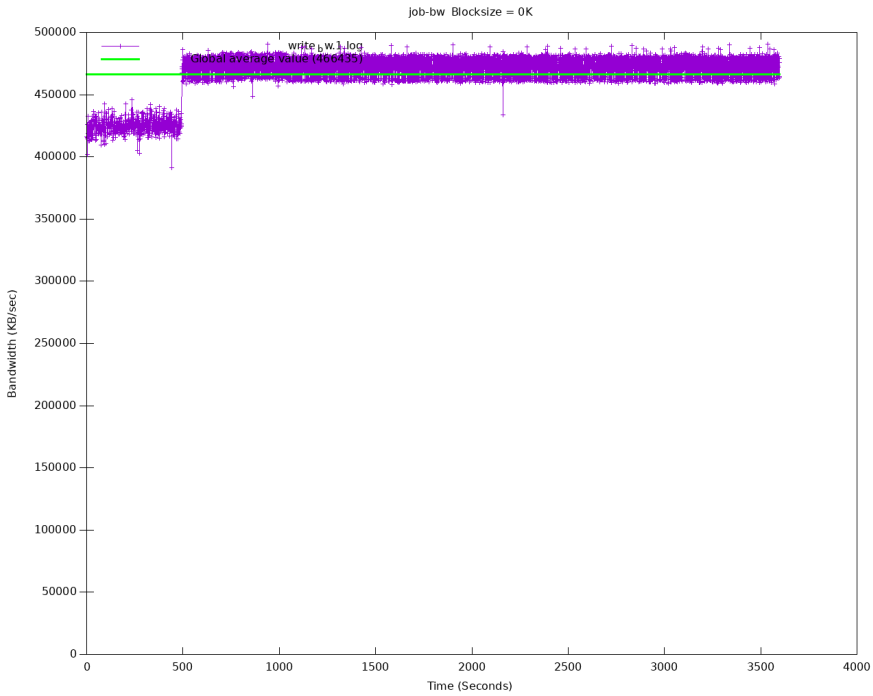


Fig. 19: Sequential writes bandwidth

After all testing is done, fio deployment and resources can be deleted by:

`shell kubectl cnpg fio <fio-job-name> --dry-run | kubectl delete -f -` make sure use the same name which was used to create the fio deployment and add namespace if applicable.

1.4.52 Commercial support

CloudNativePG is an independent open source project that doesn't endorse any company.

The Support website lists third-party companies and individuals that provide products or services related to CloudNativePG.

If you're providing commercial support for CloudNativePG, add yourself or your organization to that list.

1.4.53 Frequently Asked Questions (FAQ)

Running PostgreSQL in Kubernetes

Everyone knows that stateful workloads like PostgreSQL cannot run in Kubernetes. Why do you say the contrary?

An *independent research survey commissioned by the Data on Kubernetes Community* in September 2021 revealed that half of the respondents run most of their production workloads on Kubernetes. 90% of them believe that Kubernetes is ready for stateful workloads, and 70% of them run databases in production. Databases like Postgres. However, according to them, significant challenges remain, such as the knowledge gap (Kubernetes and Cloud Native, in general,

have a steep learning curve) and the quality of Kubernetes operators. The latter is the reason why we believe that an operator like CloudNativePG highly contributes to the success of your project.

For database fanatics like us, a real game-changer has been the introduction of the support for local persistent volumes in *Kubernetes 1.14 in April 2019* .

CloudNativePG is built on immutable application containers. What does it mean?

According to the microservice architectural pattern, a container is designed to run a single application or process. As a result, such container images are built to run the main application as the single entry point (the so-called PID 1 process).

In Kubernetes terms, the application is referred to as workload. Workloads can be stateless like a web application server or stateful like a database. Mapping this concept to PostgreSQL, an immutable application container is a single "postgres" process that is running and tied to a single and specific version - the one in the immutable container image.

No other processes such as SSH or systemd, or syslog are allowed.

Immutable Application Containers are in contrast with Mutable System Containers, which are still a very common way to interpret and use containers.

Immutable means that a container won't be modified during its life: no updates, no patches, no configuration changes. If you must update the application code or apply a patch, you build a new image and redeploy it. Immutability makes deployments safer and more repeatable.

For more information, please refer to *"Why EDB chose immutable application containers"* .

What does Cloud Native mean?

The Cloud Native Computing Foundation defines the term "*Cloud Native*". However, since the start of the Cloud Native PostgreSQL/CloudNativePG operator at 2ndQuadrant, the development team has been interpreting Cloud Native as three main concepts:

1. An existing, healthy, genuine, and prosperous DevOps culture, founded on people, as well as principles and processes, which enables teams and organizations (as teams of teams) to continuously change so to innovate and accelerate the delivery of outcomes and produce value for the business in safer, more efficient, and more engaging ways
2. A microservice architecture that is based on Immutable Application Containers
3. A way to manage and orchestrate these containers, such as Kubernetes

Currently, the standard de facto for container orchestration is Kubernetes, which automates the deployment, administration and scalability of Cloud Native Applications.

Another definition of Cloud Native that resonates with us is the one defined by Ibryam and Huß in *"Kubernetes Patterns", published by O'Reilly* :

> Principles, Patterns, Tools to automate containerized microservices at scale

Can I run CloudNativePG on bare metal Kubernetes?

Yes, definitely. You can run Kubernetes on bare metal. And you can dedicate one or more physical worker nodes with locally attached storage to PostgreSQL workloads for maximum and predictable I/O performance.

The actual Cloud Native PostgreSQL project, from which CloudNativePG originated, was born after a pilot project in 2019 that benchmarked storage and PostgreSQL on the same bare metal server, first directly in Linux, and then inside Kubernetes. As expected, the experiment showed only negligible performance impact introduced by the container running in Kubernetes through local persistent volumes, allowing the Cloud Native initiative to continue.

Why should I use PostgreSQL replication instead of file system replication?

Please read the *Architecture: Synchronizing the state*

section.

Why should I use an operator instead of running PostgreSQL as a container?

The most basic approach to running PostgreSQL in Kubernetes is to have a pod, which is the smallest unit of deployment in Kubernetes, running a Postgres container with no replica. The volume hosting the Postgres data directory is mounted on the pod, and it usually resides on network storage. In this case, Kubernetes restarts the pod in case of a problem or moves it to another Kubernetes node.

The most sophisticated approach is to run PostgreSQL using an operator. An operator is an extension of the Kubernetes controller and defines how a complex application works in business continuity contexts. The operator pattern is currently state of the art in Kubernetes for this purpose. An operator simulates the work of a human operator in an automated and programmatic way.

Postgres is a complex application, and an operator not only needs to deploy a cluster (the first step), but also properly react after unexpected events. The typical example is that of a failover.

An operator relies on Kubernetes for capabilities like self-healing, scalability, replication, high availability, backup, recovery, updates, access, resource control, storage management, and so on. It also facilitates the integration of a PostgreSQL cluster in the log management and monitoring infrastructure.

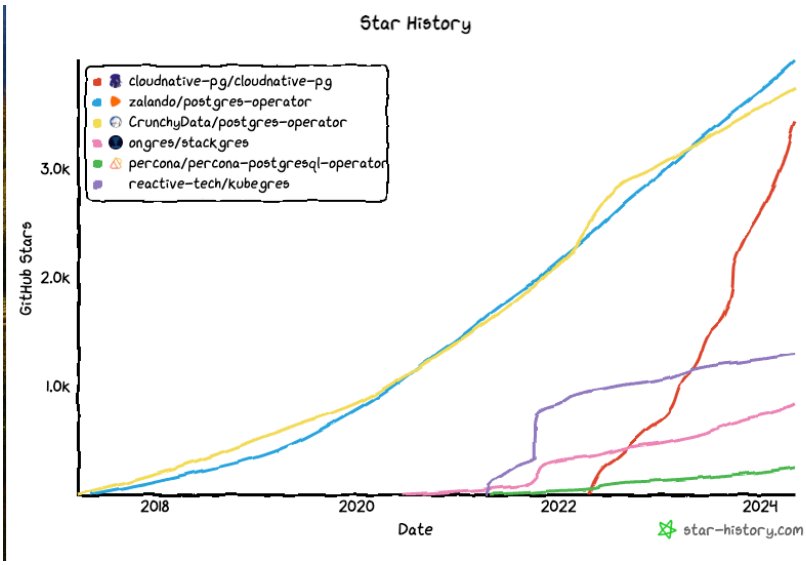
CloudNativePG enables the definition of the desired state of a PostgreSQL cluster via declarative configuration. Kubernetes continuously makes sure that the current state of the infrastructure matches the desired one through reconciliation loops initiated by the Kubernetes controller. If the desired state and the actual state don't match, reconciliation loops trigger self-healing procedures. That's where an operator like CloudNativePG comes into play.

Are there any other operators for Postgres out there?

Yes, of course. And our advice is that you look at all of them and compare them with CloudNativePG before making your decision. You will see that most of these operators use an external failover management tool (Patroni or similar) and rely on StatefulSets.

Here is a non exhaustive list, in chronological order from their publication on GitHub:

- Crunchy Data Postgres Operator (2017)
- Zalando Postgres Operator (2017)
- Stackgres (2020)
- Percona Operator for PostgreSQL (2021)
- Kubegres (2021)



Feel free to report any relevant missing entry as a PR.

Note: The Data on Kubernetes Community

(which includes some of our maintainers) is working on an independent and vendor neutral project to list the operators called Operator Feature Matrix .

You say that CloudNativePG is a fully declarative operator. What do you mean by that?

The easiest way is to explain declarative configuration through an example that highlights the differences with imperative configuration. In an imperative context, the state is defined as a series of tasks to be executed in sequence. So, we can get a three-node PostgreSQL cluster by creating the first instance, configuring the replication, cloning a second instance, and the third one.

In a declarative approach, the state of a system is defined using configuration, namely: there' s a PostgreSQL 13 cluster with two replicas. This approach highly simplifies change management operations, and when these are stored in source control systems like Git, it enables the Infrastructure as Code capability. And Kubernetes takes it farther than deployment, as it makes sure that our request is fulfilled at any time.

What are the required skills to run PostgreSQL on Kubernetes?

Running PostgreSQL on Kubernetes requires both PostgreSQL and Kubernetes skills in your DevOps team. The best experience is when database administrators familiarize themselves with Kubernetes core concepts and are able to interact with Kubernetes administrators.

Our advice is for everyone that wants to fully exploit Cloud Native PostgreSQL to acquire the "Certified Kubernetes Administrator (CKA)" status from the CNCF certification program.

Why isn' t CloudNativePG using StatefulSets?

CloudNativePG does not rely on `StatefulSet` resources, and instead manages the underlying PVCs directly by leveraging the selected storage class for dynamic provisioning. Please refer to the *Custom Pod Controller* section for details and reasons behind this decision.

High availability

What happens to the PostgreSQL clusters when the operator pod dies or it is not available for a certain amount of time?

The CloudNativePG operator, among other things, is responsible for self-healing capabilities. As such, they might not be available during an outage of the operator.

However, assuming that the outage does not affect the nodes where PostgreSQL clusters are running, the database will continue to serve normal operations, through the relevant Kubernetes services. Moreover, the *In-place updates of the instance manager* , which runs inside each PostgreSQL pod will still work, making sure that the database server is up, including accessory services like logging, export of metrics, continuous archiving of WAL files, etc.

To summarize:

an outage of the operator does not necessarily imply a PostgreSQL database outage; it' s like running a database without a DBA or system administrator.

What are the reasons behind CloudNativePG not relying on a failover management tool like Patroni, repmgr, or Stolon?

Although part of the team that develops CloudNativePG has been heavily involved in repmgr in the past, we decided to take a different approach and directly extend the Kubernetes controller and rely on the Kubernetes API server to hold the status of a Postgres cluster, and use it as the only source of truth to:

- control High Availability of a Postgres cluster primarily via automated failover and switchover, coordinating itself with the *In-place updates of the instance manager*
- control the Kubernetes services, that is the entry points for your applications

Should I manually resync a former primary with the new one following a failover?

No. The operator does that automatically for you, and relies on `pg_rewind` to synchronize the former primary with the new one.

Database management

Why should I use PostgreSQL?

We believe that PostgreSQL is the equivalent in the database area of what Linux represents in the operating system space. The current latest major version of Postgres is version 16, which ships out of the box:

- native streaming replication, both physical and logical
- continuous hot backup and point in time recovery
- declarative partitioning for horizontal table partitioning, which is a very well-known technique in the database area to improve vertical scalability on a single instance
- extensibility, with extensions like *PostGIS Container Images* for geographical databases
- parallel queries for vertical scalability
- JSON support, unleashing the multi-model hybrid database for both structured and unstructured data queried via standard SQL

And so on ⋯

How many databases should be hosted in a single PostgreSQL instance?

Our recommendation is to dedicate a single PostgreSQL cluster (intended as primary and multiple standby servers) to a single database, entirely managed by a single microservice application. However, by leveraging the "postgres" superuser, it is possible to create as many users and databases as desired (subject to the available resources).

The reason for this recommendation lies in the Cloud Native concept, based on microservices. In a pure microservice architecture, the microservice itself should own the data it manages exclusively. These could be flat files, queues, key-value stores, or, in our case, a PostgreSQL relational database containing both structured and unstructured data. The general idea is that only the microservice can access the database, including schema management and migrations.

CloudNativePG has been designed to work this way out of the box, by default creating an application user and an application database owned by the aforementioned application user.

Reserving a PostgreSQL instance to a single microservice owned database, enhances:

- resource management: in PostgreSQL, CPU, and memory constrained resources are generally handled at the instance level, not the database level, making it easier to integrate it with Kubernetes resource management policies at the pod level
- physical continuous backup and Point-In-Time-Recovery (PITR): given that PostgreSQL handles continuous backup and recovery at the instance level, having one database per instance simplifies PITR operations, differentiates retention policy management, and increases data protection of backups
- application updates: enable each application to decide their update policies without impacting other databases owned by different applications
- database updates: each application can decide which PostgreSQL version to use, and independently, when to upgrade to a different major version of PostgreSQL and at what conditions (e.g., cutover time)

Is there an upper limit in database size for not considering Kubernetes?

No, as Kubernetes is no different from virtual machines and bare metal as far as this is regarded. Practically, however, it depends on the available resources of your Kubernetes cluster. Our advice with very large databases (VLDB) is to consider a shared nothing architecture, where a Kubernetes worker node is dedicated to a single Postgres instance, with dedicated storage. We proved that this extreme architectural pattern works when we benchmarked running PostgreSQL on bare metal Kubernetes with local persistent volumes . Tablespaces and horizontal partitioning are data modeling techniques that you can use to improve the vertical scalability of you databases.

How can I specify a time zone in the PostgreSQL cluster?

PostgreSQL has an extensive support for time zones, as explained in the official documentation:

- Date time data types
- Client connections config options

Although time zones can even be used at session, transaction and even as part of a query in PostgreSQL, a very common way is to set them up globally. With CloudNativePG you can configure the cluster level time zone in the `.spec.postgresql.parameters` section as in the following example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg-italy
spec:
  instances: 1

  postgresql:
    parameters:
      timezone: "Europe/Rome"

  storage:
    size: 1Gi
```

The time zone can be verified with:

```
$ kubectl exec -ti pg-italy-1 -c postgres -- psql -x -c "SHOW timezone"
- [ RECORD 1 ]---------
TimeZone | Europe/Rome
```

What is the recommended architecture for best business continuity outcomes?

As covered in the *Architecture* section, the main recommendation is to adopt shared nothing architectures as much as possible, by leveraging the native capabilities and resources that Kubernetes provides in a single cluster, namely:

- availability zones: spread your instances across different availability zones in the same Kubernetes cluster
- worker nodes: as a consequence, make sure that your Postgres instances reside on different Kubernetes worker nodes
- storage: use dedicated storage for each worker node running Postgres

Use at least one standby, preferably at least two, so that you can configure synchronous replication in the cluster, introducing RPO=0 for high availability.

If you do not have availability zones - normally the case of on-premise installations - separate on worker nodes and storage.

Properly setup continuous backup on a local/regional object store.

The same architecture that is in a single Kubernetes cluster can be replicated in another Kubernetes cluster (normally in another geographical area or region) through the *Replica clusters* feature, providing disaster recovery and high availability at global scale.

You can use the WAL archive in the primary object store to feed the replica in the other region, without having to provide a streaming connection, if the default maximum RPO of 5 minutes is enough for you.

How can instances be stopped or started?

Please look at *Fencing* or *Declarative hibernation* .

What are the global objects such as roles and databases that are automatically created by CloudNativePG?

The operator automatically creates a user for the application (by default called `app`) and a database for the application (by default called `app`) which is owned by the aforementioned user.

This way, the database is ready for a microservice adoption, as developers can control migrations using the `app` user, without requiring *superuser* access.

Teams can then create another user for read-write operations through the
Declarative role management feature and assign the required `GRANT` to the tables.

1.4.54 API Reference

Package v1 contains API Schema definitions for the postgresql v1 API group

Resource Types

- *Appendix A - Common object stores for backups*
- *Multi-availability zone Kubernetes clusters*
- ClusterImageCatalog {postgresql-cnpg-io-v1-ClusterImageCatalog}
- ClusterImageCatalog {postgresql-cnpg-io-v1-ClusterImageCatalog}
- Pooler {postgresql-cnpg-io-v1-Pooler}
- ScheduledBackup {postgresql-cnpg-io-v1-ScheduledBackup}

Backup

Backup is the Schema for the backups API

Field	Description
apiVersion **[Required]** string	postgresql.cnpg.io/v1
kind **[Required]** string	Backup
metadata ** Required]** [*meta/v1.ObjectMeta*	No description provided.Refer to the Kubernetes API documentation for the fields of the metadata field.
spec ** *archive_timeout*	Specification of the desired behavior of the backup. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

continues on next page

Table 5 – continued from previous page

Field	Description
status *archive_timeout*	Most recently observed status of the backup. This data may not be up to date. Populated by the system. Read-only. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

Cluster

Cluster is the Schema for the PostgreSQL API

Field	Description
apiVersion **[Required]** string	postgresql.cnpg.io/v1
kind **[Required]** string	Cluster
metadata ** Required]** [*meta/v1.ObjectMeta*	No description provided.Refer to the Kubernetes API documentation for the fields of the metadata field.
spec ** Required]** [*ClusterSpec*	Specification of the desired behavior of the cluster. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
status *ClusterStatus*	Most recently observed status of the cluster. This data may not be up to date. Populated by the system. Read-only. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

ClusterImageCatalog

ClusterImageCatalog is the Schema for the clusterimagecatalogs API

Field	Description
apiVersion **[Required]** string	postgresql.cnpg.io/v1
kind **[Required]** string	ClusterImageCatalog
metadata ** Required]** [*meta/v1.ObjectMeta*	No description provided.Refer to the Kubernetes API documentation for the fields of the metadata field.
spec ** Required]** [*ImageCatalogSpec*	Specification of the desired behavior of the ClusterImageCatalog. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

ImageCatalog

ImageCatalog is the Schema for the imagecatalogs API

Field	Description
apiVersion **[Required]** string	postgresql.cnpg.io/v1
kind **[Required]** string	ImageCatalog
metadata ** Required]** [*meta/v1.ObjectMeta*	No description provided.Refer to the Kubernetes API documentation for the fields of the metadata field.
spec ** Required]** [*ImageCatalogSpec*	Specification of the desired behavior of the ImageCatalog. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

Pooler

Pooler is the Schema for the poolers API

Field	Description
apiVersion **[Required]** string	postgresql.cnpg.io/v1
kind **[Required]** string	Pooler
metadata ** Required]** [*meta/v1.ObjectMeta*	No description provided.Refer to the Kubernetes API documentation for the fields of the metadata field.
spec ** Required]** [*PoolerSpec*	Specification of the desired behavior of the Pooler. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
status *PoolerStatus*	Most recently observed status of the Pooler. This data may not be up to date. Populated by the system. Read-only. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

ScheduledBackup

ScheduledBackup is the Schema for the scheduledbackups API

Field	Description
apiVersion **[Required]** string	postgresql.cnpg.io/v1
kind **[Required]** string	ScheduledBackup
metadata ** Required]** [*meta/v1.ObjectMeta*	No description provided.Refer to the Kubernetes API documentation for the fields of the metadata field.
spec ** *archive_timeout*	Specification of the desired behavior of the ScheduledBackup. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

continues on next page

Table 10 – continued from previous page

Field	Description
status *archive_timeout*	Most recently observed status of the ScheduledBackup. This data may not be up to date. Populated by the system. Read-only. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

AffinityConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

AffinityConfiguration contains the info we need to create the affinity rules for Pods

Field	Description
enablePodAntiAffinity *bool*	Activates anti-affinity for the pods. The operator will define pods anti-affinity unless this field is explicitly set to false
topologyKey *string*	TopologyKey to use for anti-affinity configuration. See k8s documentation for more info on that
nodeSelector *map[string]string*	NodeSelector is map of key-value pairs used to define the nodes on which the pods can run. More info: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/
nodeAffinity *core/v1.NodeAffinity*	NodeAffinity describes node affinity scheduling rules for the pod. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity
tolerations *core/v1.Toleration*	Tolerations is a list of Tolerations that should be set for all the pods, in order to allow them to run on tainted nodes. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/
podAntiAffinityType *string*	PodAntiAffinityType allows the user to decide whether pod anti-affinity between cluster instance has to be considered a strong requirement during scheduling or not. Allowed values are: "preferred" (default if empty) or "required". Setting it to "required", could lead to instances remaining pending until new kubernetes nodes are added if all the existing nodes don't match the required pod anti-affinity rule. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#inter-pod-affinity-and-anti-affinity
additionalPodAntiAffinity *core/v1.PodAntiAffinity*	AdditionalPodAntiAffinity allows to specify pod anti-affinity terms to be added to the ones generated by the operator if EnablePodAntiAffinity is set to true (default) or to be used exclusively if set to false.
additionalPodAffinity *core/v1.PodAffinity*	AdditionalPodAffinity allows to specify pod affinity terms to be passed to all the cluster's pods.

AvailableArchitecture

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

AvailableArchitecture represents the state of a cluster’s architecture

Field	Description
goArch **[Required]** *string*	GoArch is the name of the executable architecture
hash **[Required]** *string*	Hash is the hash of the executable

AzureCredentials

Appears in:

- BarmanCredentials {postgresql-cnpg-io-v1-BarmanCredentials}

AzureCredentials is the type for the credentials to be used to upload files to Azure Blob Storage. The connection string contains every needed information. If the connection string is not specified, we’ll need the storage account name and also one (and only one) of:

- storageKey
- storageSasToken
- inheriting the credentials from the pod environment by setting inheritFromAzureAD to true

Field	Description
connectionString *SecretKeySelector*	The connection string to be used
storageAccount *SecretKeySelector*	The storage account where to upload data
storageKey *SecretKeySelector*	The storage account key to be used in conjunction with the storage account name
storageSasToken *SecretKeySelector*	A shared-access-signature to be used in conjunction with the storage account name
inheritFromAzureAD *bool*	Use the Azure AD based authentication without providing explicitly the keys.

BackupConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

BackupConfiguration defines how the backup of the cluster are taken. The supported backup methods are BarmanObjectStore and VolumeSnapshot. For details and examples refer to the Backup and Recovery section of the documentation

Field	Description
volumeSnapshot *VolumeSnapshotConfiguration*	VolumeSnapshot provides the configuration for the execution of volume snapshot backups.
barmanObjectStore *BarmanObjectStoreConfiguration*	The configuration for the barman-cloud tool suite
retentionPolicy *string*	RetentionPolicy is the retention policy to be used for backups and WALs (i.e. '60d'). The retention policy is expressed in the form of XXu where XX is a positive integer and u is in [dwm] - days, weeks, months. It's currently only applicable when using the BarmanObjectStore method.
target *archive_timeout*	The policy to decide which instance should perform backups. Available options are empty string, which will default to prefer-standby policy, primary to have backups run always on primary instances, prefer-standby to have backups run preferably on the most updated standby, if available.

BackupMethod

(Alias of `string`)

Appears in:

- BackupSpec {postgresql-cnpg-io-v1-BackupSpec}
- BackupStatus {postgresql-cnpg-io-v1-BackupStatus}
- ScheduledBackupSpec {postgresql-cnpg-io-v1-ScheduledBackupSpec}

BackupMethod defines the way of executing the physical base backups of the selected PostgreSQL instance

BackupPhase

(Alias of `string`)

Appears in:

- BackupStatus {postgresql-cnpg-io-v1-BackupStatus}

BackupPhase is the phase of the backup

BackupPluginConfiguration

Appears in:

- BackupSpec {postgresql-cnpg-io-v1-BackupSpec}
- ScheduledBackupSpec {postgresql-cnpg-io-v1-ScheduledBackupSpec}

BackupPluginConfiguration contains the backup configuration used by the backup plugin

Field	Description
name **[Required]** *string*	Name is the name of the plugin managing this backup

continues on next page

Table 15 – continued from previous page

Field	Description
parameters *map[string]string*	Parameters are the configuration parameters passed to the backup plugin for this backup

BackupSnapshotElementStatus

Appears in:

- BackupSnapshotStatus {postgresql-cnpg-io-v1-BackupSnapshotStatus}

BackupSnapshotElementStatus is a volume snapshot that is part of a volume snapshot method backup

Field	Description
name **[Required]** *string*	Name is the snapshot resource name
type **[Required]** *string*	Type is tho role of the snapshot in the cluster, such as PG_DATA, PG_WAL and PG_TABLESPACE
tablespaceName **[Required]** *string*	TablespaceName is the name of the snapshotted tablespace. Only set when type is PG_TABLESPACE

BackupSnapshotStatus

Appears in:

- BackupStatus {postgresql-cnpg-io-v1-BackupStatus}

BackupSnapshotStatus the fields exclusive to the volumeSnapshot method backup

Field	Description
elements *archive_timeout*	The elements list, populated with the gathered volume snapshots

BackupSource

Appears in:

- BootstrapRecovery {postgresql-cnpg-io-v1-BootstrapRecovery}

BackupSource contains the backup we need to restore from, plus some information that could be needed to correctly restore it.

Field	Description
LocalObjectReference *LocalObjectReference*	(Members of LocalObjectReference are embedded into this type.) No description provided.

continues on next page

Table 18 – continued from previous page

Field	Description
endpointCA *SecretKeySelector*	EndpointCA store the CA bundle of the barman endpoint. Useful when using self-signed certificates to avoid errors with certificate issuer and barman-cloud-wal-archive.

BackupSpec

Appears in:

- *Appendix A - Common object stores for backups*

BackupSpec defines the desired state of Backup

Field	Description
cluster ** Required]** [*LocalObjectReference*	The cluster to backup
target *archive_timeout*	The policy to decide which instance should perform this backup. If empty, it defaults to cluster.spec.backup.target. Available options are empty string, primary and prefer-standby. primary to have backups run always on primary instances, prefer-standby to have backups run preferably on the most updated standby, if available.
method *archive_timeout*	The backup method to be used, possible options are barmanObjectStore, volumeSnapshot or plugin. Defaults to: barmanObjectStore.
pluginConfiguration *archive_timeout*	Configuration parameters passed to the plugin managing this backup
online *bool*	Whether the default type of backup with volume snapshots is online/hot (true, default) or offline/cold (false) Overrides the default setting specified in the cluster field '.spec.backup.volumeSnapshot.online'
onlineConfiguration *OnlineConfiguration*	Configuration parameters to control the online/hot backup with volume snapshots Overrides the default settings specified in the cluster '.backup.volumeSnapshot.onlineConfiguration' stanza

BackupStatus

Appears in:

- *Appendix A - Common object stores for backups*

BackupStatus defines the observed state of Backup

Field	Description
BarmanCredentials *BarmanCredentials*	(Members of BarmanCredentials are embedded into this type.) The potential credentials for each cloud provider
endpointCA *SecretKeySelector*	EndpointCA store the CA bundle of the barman endpoint. Useful when using self-signed certificates to avoid errors with certificate issuer and barman-cloud-wal-archive.

continues on next page

Table 20 – continued from previous page

Field	Description
endpointURL *string*	Endpoint to be used to upload data to the cloud, overriding the automatic endpoint discovery
destinationPath *string*	The path where to store the backup (i.e. s3://bucket/path/to/folder) this path, with different destination folders, will be used for WALs and for data. This may not be populated in case of errors.
serverName *string*	The server name on S3, the cluster name is used if this parameter is omitted
encryption *string*	Encryption method required to S3 API
backupId *string*	The ID of the Barman backup
backupName *string*	The Name of the Barman backup
phase *archive_timeout*	The last backup status
startedAt *meta/v1.Time*	When the backup was started
stoppedAt *meta/v1.Time*	When the backup was terminated
beginWal *string*	The starting WAL
endWal *string*	The ending WAL
beginLSN *string*	The starting xlog
endLSN *string*	The ending xlog
error *string*	The detected error
commandOutput *string*	Unused. Retained for compatibility with old versions.
commandError *string*	The backup command output in case of error
backupLabelFile *byte*	Backup label file content as returned by Postgres in case of online (hot) backups
tablespaceMapFile *byte*	Tablespace map file content as returned by Postgres in case of online (hot) backups
instanceID *InstanceID*	Information to identify the instance where the backup has been taken from
snapshotBackupStatus *archive_timeout*	Status of the volumeSnapshot backup
method *archive_timeout*	The backup method being used
online **[Required]** *bool*	Whether the backup was online/hot (true) or offline/cold (false)

BackupTarget

(Alias of `string`)

Appears in:

- BackupConfiguration {postgresql-cnpg-io-v1-BackupConfiguration}
- BackupSpec {postgresql-cnpg-io-v1-BackupSpec}
- ScheduledBackupSpec {postgresql-cnpg-io-v1-ScheduledBackupSpec}

BackupTarget describes the preferred targets for a backup

BarmanCredentials

Appears in:

- BackupStatus {postgresql-cnpg-io-v1-BackupStatus}
- BarmanObjectStoreConfiguration {postgresql-cnpg-io-v1-BarmanObjectStoreConfiguration}

BarmanCredentials an object containing the potential credentials for each cloud provider

Field	Description
googleCredentials *GoogleCredentials*	The credentials to use to upload data to Google Cloud Storage
s3Credentials *S3Credentials*	The credentials to use to upload data to S3
azureCredentials *AzureCredentials*	The credentials to use to upload data to Azure Blob Storage

BarmanObjectStoreConfiguration

Appears in:

- BackupConfiguration {postgresql-cnpg-io-v1-BackupConfiguration}
- The `externalClusters section <The *externalClusters* section>`

BarmanObjectStoreConfiguration contains the backup configuration using Barman against an S3-compatible object storage

Field	Description
BarmanCredentials *BarmanCredentials*	(Members of BarmanCredentials are embedded into this type.) The potential credentials for each cloud provider
endpointURL *string*	Endpoint to be used to upload data to the cloud, overriding the automatic endpoint discovery
endpointCA *SecretKeySelector*	EndpointCA store the CA bundle of the barman endpoint. Useful when using self-signed certificates to avoid errors with certificate issuer and barman-cloud-wal-archive
destinationPath **[Required]** *string*	The path where to store the backup (i.e. s3://bucket/path/to/folder) this path, with different destination folders, will be used for WALs and for data
serverName *string*	The server name on S3, the cluster name is used if this parameter is omitted
wal *archive_timeout*	The configuration for the backup of the WAL stream. When not defined, WAL files will be stored uncompressed and may be unencrypted in the object store, according to the bucket default policy.
data *archive_timeout*	The configuration to be used to backup the data files When not defined, base backups files will be stored uncompressed and may be unencrypted in the object store, according to the bucket default policy.
tags *map[string]string*	Tags is a list of key value pairs that will be passed to the Barman --tags option.
historyTags *map[string]string*	HistoryTags is a list of key value pairs that will be passed to the Barman --history-tags option.

BootstrapConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

BootstrapConfiguration contains information about how to create the PostgreSQL cluster. Only a single bootstrap method can be defined among the supported ones. initdb will be used as the bootstrap method if left unspecified. Refer to the Bootstrap page of the documentation for more information.

Field	Description
initdb *archive_timeout*	Bootstrap the cluster via initdb
recovery *archive_timeout*	Bootstrap the cluster from a backup
pg_basebackup *archive_timeout*	Bootstrap the cluster taking a physical backup of another compatible PostgreSQL instance

BootstrapInitDB

Appears in:

- BootstrapConfiguration {postgresql-cnpg-io-v1-BootstrapConfiguration}

BootstrapInitDB is the configuration of the bootstrap process when initdb is used Refer to the Bootstrap page of the documentation for more information.

Field	Description
database *string*	Name of the database used by the application. Default: app.
owner *string*	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the database key.
secret *LocalObjectReference*	Name of the secret containing the initial credentials for the owner of the user database. If empty a new secret will be created from scratch
options *string*	The list of options that must be passed to initdb when creating the cluster. Deprecated: This could lead to inconsistent configurations, please use the explicit provided parameters instead. If defined, explicit values will be ignored.
dataChecksums *bool*	Whether the -k option should be passed to initdb, enabling checksums on data pages (default: false)
encoding *string*	The value to be passed as option --encoding for initdb (default:UTF8)
localeCollate *string*	The value to be passed as option --lc-collate for initdb (default:C)
localeCType *string*	The value to be passed as option --lc-ctype for initdb (default:C)
walSegmentSize *int*	The value in megabytes (1 to 1024) to be passed to the --wal-segsize option for initdb (default: empty, resulting in PostgreSQL default: 16MB)
postInitSQL *string*	List of SQL queries to be executed as a superuser immediately after the cluster has been created - to be used with extreme care (by default empty)
postInitApplicationSQL *string*	List of SQL queries to be executed as a superuser in the application database right after is created - to be used with extreme care (by default empty)

continues on next page

Table 24 – continued from previous page

Field	Description
postInitTemplateSQL *string*	List of SQL queries to be executed as a superuser in the template1 after the cluster has been created - to be used with extreme care (by default empty)
import *Import*	Bootstraps the new cluster by importing data from an existing PostgreSQL instance using logical backup (pg_dump and pg_restore)
postInitApplicationSQLRefs *archive_timeout*	PostInitApplicationSQLRefs points references to ConfigMaps or Secrets which contain SQL files, the general implementation order to these references is from all Secrets to all ConfigMaps, and inside Secrets or ConfigMaps, the implementation order is same as the order of each array (by default empty)

BootstrapPgBaseBackup

Appears in:

- BootstrapConfiguration {postgresql-cnpg-io-v1-BootstrapConfiguration}

BootstrapPgBaseBackup contains the configuration required to take a physical backup of an existing PostgreSQL cluster

Field	Description
source **[Required]** *string*	The name of the server of which we need to take a physical backup
database *string*	Name of the database used by the application. Default: app.
owner *string*	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the database key.
secret *LocalObjectReference*	Name of the secret containing the initial credentials for the owner of the user database. If empty a new secret will be created from scratch

BootstrapRecovery

Appears in:

- BootstrapConfiguration {postgresql-cnpg-io-v1-BootstrapConfiguration}

BootstrapRecovery contains the configuration required to restore from an existing cluster using 3 methodologies: external cluster, volume snapshots or backup objects. Full recovery and Point-In-Time Recovery are supported. The method can be also be used to create clusters in continuous recovery (replica clusters), also supporting cascading replication when instances >

1. Once the cluster exits recovery, the password for the superuser will be changed through the provided secret. Refer to the Bootstrap page of the documentation for more information.

Field	Description
backup *archive_timeout*	The backup object containing the physical base backup from which to initiate the recovery procedure. Mutually exclusive with source and volumeSnapshots.

continues on next page

Table 26 – continued from previous page

Field	Description
source *string*	The external cluster whose backup we will restore. This is also used as the name of the folder under which the backup is stored, so it must be set to the name of the source cluster Mutually exclusive with backup.
volumeSnapshots *DataSource*	The static PVC data source(s) from which to initiate the recovery procedure. Currently supporting VolumeSnapshot and PersistentVolumeClaim resources that map an existing PVC group, compatible with CloudNativePG, and taken with a cold backup copy on a fenced Postgres instance (limitation which will be removed in the future when online backup will be implemented). Mutually exclusive with backup.
recoveryTarget *archive_timeout*	By default, the recovery process applies all the available WAL files in the archive (full recovery). However, you can also end the recovery as soon as a consistent state is reached or recover to a point-in-time (PITR) by specifying a RecoveryTarget object, as expected by PostgreSQL (i.e., timestamp, transaction Id, LSN, ...). More info: https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET
database *string*	Name of the database used by the application. Default: app.
owner *string*	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the database key.
secret *LocalObjectReference*	Name of the secret containing the initial credentials for the owner of the user database. If empty a new secret will be created from scratch

CatalogImage

Appears in:

- ImageCatalogSpec {postgresql-cnpg-io-v1-ImageCatalogSpec}

CatalogImage defines the image and major version

Field	Description
image **[Required]** *string*	The image reference
major **[Required]** *int*	The PostgreSQL major version of the image. Must be unique within the catalog.

CertificatesConfiguration

Appears in:

- CertificatesStatus {postgresql-cnpg-io-v1-CertificatesStatus}
- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

CertificatesConfiguration contains the needed configurations to handle server certificates.

Field	Description
serverCASecret *string*	The secret containing the Server CA certificate. If not defined, a new secret will be created with a self-signed CA and will be used to generate the TLS certificate ServerTLSSecret. Contains: - ca.crt: CA that should be used to validate the server certificate, used as sslrootcert in client connection strings. - ca.key: key used to generate Server SSL certs, if ServerTLSSecret is provided, this can be omitted.
serverTLSSecret *string*	The secret of type kubernetes.io/tls containing the server TLS certificate and key that will be set as ssl_cert_file and ssl_key_file so that clients can connect to postgres securely. If not defined, ServerCASecret must provide also ca.key and a new secret will be created using the provided CA.
replicationTLSSecret *string*	The secret of type kubernetes.io/tls containing the client certificate to authenticate as the streaming_replica user. If not defined, ClientCASecret must provide also ca.key, and a new secret will be created using the provided CA.
clientCASecret *string*	The secret containing the Client CA certificate. If not defined, a new secret will be created with a self-signed CA and will be used to generate all the client certificates. Contains: - ca.crt: CA that should be used to validate the client certificates, used as ssl_ca_file of all the instances. - ca.key: key used to generate client certificates, if ReplicationTLSSecret is provided, this can be omitted.
serverAltDNSNames *string*	The list of the server alternative DNS names to be added to the generated server TLS certificates, when required.

CertificatesStatus

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

CertificatesStatus contains configuration certificates and related expiration dates.

Field	Description
CertificatesConfiguration *archive_timeout*	(Members of CertificatesConfiguration are embedded into this type.) Needed configurations to handle server certificates, initialized with default values, if needed.
expirations *map[string]string*	Expiration dates for all certificates.

ClusterSpec

Appears in:

- *Multi-availability zone Kubernetes clusters*

ClusterSpec defines the desired state of Cluster

Field	Description
description *string*	Description of this PostgreSQL cluster
inheritedMetadata *EmbeddedObjectMetadata*	Metadata that will be inherited by all objects related to the Cluster
imageName *string*	Name of the container image, supporting both tags (<image>:<tag>) and digests for deterministic and repeatable deployments (<image>:<tag>@sha256:<digestValue>)
imageCatalogRef *ImageCatalogRef*	Defines the major PostgreSQL version we want to use within an ImageCatalog
imagePullPolicy *core/v1.PullPolicy*	Image pull policy. One of Always, Never or IfNotPresent. If not defined, it defaults to IfNotPresent. Cannot be updated. More info: https://kubernetes.io/docs/concepts/containerimages#updating-images
schedulerName *string*	If specified, the pod will be dispatched by specified Kubernetes scheduler. If not specified, the pod will be dispatched by the default scheduler. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
postgresUID *int64*	The UID of the postgres user inside the image, defaults to 26
postgresGID *int64*	The GID of the postgres user inside the image, defaults to 26
instances **[Required]** *int*	Number of instances required in the cluster
minSyncReplicas *int*	Minimum number of instances required in synchronous replication with the primary. Undefined or 0 allow writes to complete when no standby is available.
maxSyncReplicas *int*	The target value for the synchronous replication quorum, that can be decreased if the number of ready standbys is lower than this. Undefined or 0 disable synchronous replication.
postgresql *PostgresConfiguration*	Configuration of the PostgreSQL server
replicationSlots *archive_timeout*	Replication slots management configuration
bootstrap *archive_timeout*	Instructions to bootstrap this cluster
replica *ReplicaClusterConfiguration*	Replica cluster configuration
superuserSecret *LocalObjectReference*	The secret containing the superuser password. If not defined a new secret will be created with a randomly generated password
enableSuperuserAccess *bool*	When this option is enabled, the operator will use the SuperuserSecret to update the postgres user password (if the secret is not present, the operator will automatically create one). When this option is disabled, the operator will ignore the SuperuserSecret content, delete it when automatically created, and then blank the password of the postgres user by setting it to NULL. Disabled by default.
certificates *archive_timeout*	The configuration for the CA and related certificates
imagePullSecrets *LocalObjectReference*	The list of pull secrets to be used to pull the images
storage *archive_timeout*	Configuration of the storage of the instances
serviceAccountTemplate *ServiceAccountTemplate*	Configure the generation of the service account
walStorage *archive_timeout*	Configuration of the storage for PostgreSQL WAL (Write-Ahead Log)

continues on next page

Table 30 – continued from previous page

Field	Description
ephemeralVolumeSource *core/v1.EphemeralVolumeSource*	EphemeralVolumeSource allows the user to configure the source of ephemeral volumes.
startDelay *int32*	The time in seconds that is allowed for a PostgreSQL instance to successfully start up (default 3600). The startup probe failure threshold is derived from this value using the formula: ceiling(startDelay / 10).
stopDelay *int32*	The time in seconds that is allowed for a PostgreSQL instance to gracefully shutdown (default 1800)
smartShutdownTimeout *int32*	The time in seconds that controls the window of time reserved for the smart shutdown of Postgres to complete. Make sure you reserve enough time for the operator to request a fast shutdown of Postgres (that is: stopDelay - smartShutdownTimeout).
switchoverDelay *int32*	The time in seconds that is allowed for a primary PostgreSQL instance to gracefully shutdown during a switchover. Default value is 3600 seconds (1 hour).
failoverDelay *int32*	The amount of time (in seconds) to wait before triggering a failover after the primary PostgreSQL instance in the cluster was detected to be unhealthy
affinity *AffinityConfiguration*	Affinity/Anti-affinity rules for Pods
topologySpreadConstraints *core/v1.TopologySpreadConstraint*	TopologySpreadConstraints specifies how to spread matching pods among the given topology. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/
resources *core/v1.ResourceRequirements*	Resources requirements of every generated Pod. Please refer to https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ for more information.
ephemeralVolumesSizeLimit ** Required]** [*EphemeralVolumesSizeLimitConfiguration*	EphemeralVolumesSizeLimit allows the user to set the limits for the ephemeral volumes
priorityClassName *string*	Name of the priority class which will be used in every generated Pod, if the PriorityClass specified does not exist, the pod will not be able to schedule. Please refer to https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/#priorityclass for more information
primaryUpdateStrategy *PrimaryUpdateStrategy*	Deployment strategy to follow to upgrade the primary server during a rolling update procedure, after all replicas have been successfully updated: it can be automated (unsupervised- default) or manual (supervised)
primaryUpdateMethod *PrimaryUpdateMethod*	Method to follow to upgrade the primary server during a rolling update procedure, after all replicas have been successfully updated: it can be with a switchover (switchover) or in-place (restart- default)
backup *archive_timeout*	The configuration to be used for backups
nodeMaintenanceWindow *NodeMaintenanceWindow*	Define a maintenance window for the Kubernetes nodes
monitoring *archive_timeout*	The configuration of the monitoring infrastructure of this cluster
externalClusters *ExternalCluster*	The list of external clusters which are used in the configuration
logLevel *string*	The instances' log level, one of the following values: error, warning, info (default), debug, trace
projectedVolumeTemplate *core/v1.ProjectedVolumeSource*	Template to be used to define projected volumes, projected volumes will be mounted under /projected base folder
env *core/v1.EnvVar*	Env follows the Env format to pass environment variables to the pods created in the cluster

continues on next page

Table 30 – continued from previous page

Field	Description
envFrom *core/v1.EnvFromSource*	EnvFrom follows the EnvFrom format to pass environment variables sources to the pods to be used by Env
managed *ManagedConfiguration*	The configuration that is used by the portions of PostgreSQL that are managed by the instance manager
seccompProfile *core/v1.SeccompProfile*	The SeccompProfile applied to every Pod and Container. Defaults to: RuntimeDefault
tablespaces *TablespaceConfiguration*	The tablespaces configuration
enablePDB *bool*	Manage the PodDisruptionBudget resources within the cluster. When configured as true (default setting), the pod disruption budgets will safeguard the primary node from being terminated. Conversely, setting it to false will result in the absence of any PodDisruptionBudget resource, permitting the shutdown of all nodes hosting the PostgreSQL cluster. This latter configuration is advisable for any PostgreSQL cluster employed for development/staging purposes.
plugins ** Required]** [*PluginConfigurationList*	The plugins configuration, containing any plugin to be loaded with the corresponding configuration

ClusterStatus

Appears in:

- *Multi-availability zone Kubernetes clusters*

ClusterStatus defines the observed state of Cluster

Field	Description
instances *int*	The total number of PVC Groups detected in the cluster. It may differ from the number of existing instance pods.
readyInstances *int*	The total number of ready instances in the cluster. It is equal to the number of ready instance pods.
instancesStatus *map[PodStatus][]string*	InstancesStatus indicates in which status the instances are
instancesReportedState *map[PodName]InstanceReportedState*	The reported state of the instances during the last reconciliation loop
managedRolesStatus *ManagedRoles*	ManagedRolesStatus reports the state of the managed roles in the cluster
tablespacesStatus *archive_timeout*	TablespacesStatus reports the state of the declarative tablespaces in the cluster
timelineID *int*	The timeline of the Postgres cluster
topology *Topology*	Instances topology.
latestGeneratedNode *int*	ID of the latest generated node (used to avoid node name clashing)
currentPrimary *string*	Current primary instance
targetPrimary *string*	Target primary instance, this is different from the previous one during a switchover or a failover
pvcCount *int32*	How many PVCs have been created by this cluster
jobCount *int32*	How many Jobs have been created by this cluster
danglingPVC *string*	List of all the PVCs created by this cluster and still available which are not attached to a Pod
resizingPVC *string*	List of all the PVCs that have ResizingPVC condition.
initializingPVC *string*	List of all the PVCs that are being initialized by this cluster

continues on next page

Table 31 – continued from previous page

Field	Description
healthyPVC *string*	List of all the PVCs not dangling nor initializing
unusablePVC *string*	List of all the PVCs that are unusable because another PVC is missing
writeService *string*	Current write pod
readService *string*	Current list of read pods
phase *string*	Current phase of the cluster
phaseReason *string*	Reason for the current phase
secretsResourceVersion *SecretsResourceVersion*	The list of resource versions of the secrets managed by the operator. Every change here is done in the interest of the instance manager, which will refresh the secret data
configMapResourceVersion *ConfigMapResourceVersion*	The list of resource versions of the configmaps, managed by the operator. Every change here is done in the interest of the instance manager, which will refresh the configmap data
certificates *archive_timeout*	The configuration for the CA and related certificates, initialized with defaults.
firstRecoverabilityPoint *string*	The first recoverability point, stored as a date in RFC3339 format. This field is calculated from the content of FirstRecoverabilityPointByMethod
firstRecoverabilityPointByMethod *map[BackupMethod]meta/v1.Time*	The first recoverability point, stored as a date in RFC3339 format, per backup method type
lastSuccessfulBackup *string*	Last successful backup, stored as a date in RFC3339 format This field is calculated from the content of LastSuccessfulBackupByMethod
lastSuccessfulBackupByMethod *map[BackupMethod]meta/v1.Time*	Last successful backup, stored as a date in RFC3339 format, per backup method type
lastFailedBackup *string*	Stored as a date in RFC3339 format
cloudNativePGCommitHash *string*	The commit hash number of which this operator running
currentPrimaryTimestamp *string*	The timestamp when the last actual promotion to primary has occurred
currentPrimaryFailingSinceTimestamp *string*	The timestamp when the primary was detected to be unhealthy This field is reported when .spec.failoverDelay is populated or during online upgrades
targetPrimaryTimestamp *string*	The timestamp when the last request for a new primary has occurred
poolerIntegrations *PoolerIntegrations*	The integration needed by poolers referencing the cluster
cloudNativePGOperatorHash *string*	The hash of the binary of the operator
availableArchitectures *archive_timeout*	AvailableArchitectures reports the available architectures of a cluster
conditions *meta/v1.Condition*	Conditions for cluster object
instanceNames *string*	List of instance names in the cluster
onlineUpdateEnabled *bool*	OnlineUpdateEnabled shows if the online upgrade is enabled inside the cluster
azurePVCUpdateEnabled *bool*	AzurePVCUpdateEnabled shows if the PVC online upgrade is enabled for this cluster
image *string*	Image contains the image name used by the pods
pluginStatus ** Required]** [*PluginStatus*	PluginStatus is the status of the loaded plugins
switchReplicaClusterStatus *SwitchReplicaClusterStatus*	SwitchReplicaClusterStatus is the status of the switch to replica cluster

CompressionType

(Alias of `string`)

Appears in:

- DataBackupConfiguration {postgresql-cnpg-io-v1-DataBackupConfiguration}
- WalBackupConfiguration {postgresql-cnpg-io-v1-WalBackupConfiguration}

CompressionType encapsulates the available types of compression

ConfigMapKeySelector

Appears in:

- MonitoringConfiguration {postgresql-cnpg-io-v1-MonitoringConfiguration}
- PostInitApplicationSQLRefs {postgresql-cnpg-io-v1-PostInitApplicationSQLRefs}

ConfigMapKeySelector contains enough information to let you locate the key of a ConfigMap

Field	Description
LocalObjectReference *LocalObjectReference*	(Members of LocalObjectReference are embedded into this type.) The name of the secret in the pod's namespace to select from.
key **[Required]** *string*	The key to select

ConfigMapResourceVersion

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

ConfigMapResourceVersion is the resource versions of the secrets managed by the operator

Field	Description
metrics *map[string]string*	A map with the versions of all the config maps used to pass metrics. Map keys are the config map names, map values are the versions

DataBackupConfiguration

Appears in:

- BarmanObjectStoreConfiguration {postgresql-cnpg-io-v1-BarmanObjectStoreConfiguration}

DataBackupConfiguration is the configuration of the backup of the data directory

Field	Description
compression *CompressionType*	Compress a backup file (a tar file per tablespace) while streaming it to the object store. Available options are empty string (no compression, default), gzip, bzip2 or snappy.
encryption *EncryptionType*	Whenever to force the encryption of files (if the bucket is not already configured for that). Allowed options are empty string (use the bucket policy, default), AES256 and aws:kms
jobs *int32*	The number of parallel jobs to be used to upload the backup, defaults to 2
immediateCheckpoint *bool*	Control whether the I/O workload for the backup initial checkpoint will be limited, according to the checkpoint_completion_target setting on the PostgreSQL server. If set to true, an immediate checkpoint will be used, meaning PostgreSQL will complete the checkpoint as soon as possible. false by default.
additionalCommandArgs **[Required]** *string*	AdditionalCommandArgs represents additional arguments that can be appended to the 'barman-cloud-backup' command-line invocation. These arguments provide flexibility to customize the backup process further according to specific requirements or configurations. Example: In a scenario where specialized backup options are required, such as setting a specific timeout or defining custom behavior, users can use this field to specify additional command arguments. Note: It's essential to ensure that the provided arguments are valid and supported by the 'barman-cloud-backup' command, to avoid potential errors or unintended behavior during execution.

DataSource

Appears in:

- BootstrapRecovery {postgresql-cnpg-io-v1-BootstrapRecovery}

DataSource contains the configuration required to bootstrap a PostgreSQL cluster from an existing storage

Field	Description
storage ** Required]** [*core/v1.TypedLocalObjectReference*	Configuration of the storage of the instances
walStorage *core/v1.TypedLocalObjectReference*	Configuration of the storage for PostgreSQL WAL (Write-Ahead Log)
tablespaceStorage *map[string]core/v1.TypedLocalObjectRe	Configuration of the storage for PostgreSQL tablespaces

DatabaseRoleRef

Appears in:

- TablespaceConfiguration {postgresql-cnpg-io-v1-TablespaceConfiguration}

DatabaseRoleRef is a reference an a role available inside PostgreSQL

Field	Description
name *string*	No description provided.

EmbeddedObjectMetadata

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

EmbeddedObjectMetadata contains metadata to be inherited by all resources related to a Cluster

Field	Description
labels *map[string]string*	No description provided.
annotations *map[string]string*	No description provided.

EncryptionType

(Alias of `string`)

Appears in:

- DataBackupConfiguration {postgresql-cnpg-io-v1-DataBackupConfiguration}
- WalBackupConfiguration {postgresql-cnpg-io-v1-WalBackupConfiguration}

EncryptionType encapsulated the available types of encryption

EnsureOption

(Alias of `string`)

Appears in:

- RoleConfiguration {postgresql-cnpg-io-v1-RoleConfiguration}

EnsureOption represents whether we should enforce the presence or absence of a Role in a PostgreSQL instance

EphemeralVolumesSizeLimitConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

EphemeralVolumesSizeLimitConfiguration contains the configuration of the ephemeral storage

Field	Description
shm ** Required]** [*k8s.io/apimachinery/pkg/api/resource.Qu	Shm is the size limit of the shared memory volume
temporaryData ** Required]** [*k8s.io/apimachinery/pkg/api/resource.Qu	TemporaryData is the size limit of the temporary data volume

ExternalCluster

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ExternalCluster represents the connection parameters to an external cluster which is used in the other sections of the configuration

Field	Description
name **[Required]** *string*	The server name, required
connectionParameters *map[string]string*	The list of connection parameters, such as dbname, host, username, etc
sslCert *core/v1.SecretKeySelector*	The reference to an SSL certificate to be used to connect to this instance
sslKey *core/v1.SecretKeySelector*	The reference to an SSL private key to be used to connect to this instance
sslRootCert *core/v1.SecretKeySelector*	The reference to an SSL CA public key to be used to connect to this instance
password *core/v1.SecretKeySelector*	The reference to the password to be used to connect to the server. If a password is provided, CloudNativePG creates a PostgreSQL passfile at /controller/external/NAME/pass (where "NAME" is the cluster's name). This passfile is automatically referenced in the connection string when establishing a connection to the remote PostgreSQL server from the current PostgreSQL Cluster. This ensures secure and efficient password management for external clusters.
barmanObjectStore *BarmanObjectStoreConfiguration*	The configuration for the barman-cloud tool suite

GoogleCredentials

Appears in:

- BarmanCredentials {postgresql-cnpg-io-v1-BarmanCredentials}

GoogleCredentials is the type for the Google Cloud Storage credentials. This needs to be specified even if we run inside a GKE environment.

Field	Description
applicationCredentials *SecretKeySelector*	The secret containing the Google Cloud Storage JSON file with the credentials
gkeEnvironment *bool*	If set to true, will presume that it's running inside a GKE environment, default to false.

ImageCatalogRef

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ImageCatalogRef defines the reference to a major version in an ImageCatalog

Field	Description
TypedLocalObjectReference *core/v1.TypedLocalObjectReference*	(Members of TypedLocalObjectReference are embedded into this type.) No description provided.
major **[Required]** *int*	The major version of PostgreSQL we want to use from the ImageCatalog

ImageCatalogSpec

Appears in:

- ClusterImageCatalog {postgresql-cnpg-io-v1-ClusterImageCatalog}
- ClusterImageCatalog {postgresql-cnpg-io-v1-ClusterImageCatalog}

ImageCatalogSpec defines the desired ImageCatalog

Field	Description
images ** Required]** [*CatalogImage*	List of CatalogImages available in the catalog

Import

Appears in:

- BootstrapInitDB {postgresql-cnpg-io-v1-BootstrapInitDB}

Import contains the configuration to init a database from a logic snapshot of an externalCluster

Field	Description
source ** Required]** [*ImportSource*	The source of the import
type ** Required]** [*SnapshotType*	The import type. Can be microservice or monolith.
databases **[Required]** *string*	The databases to import
roles *string*	The roles to import
postImportApplicationSQL *string*	List of SQL queries to be executed as a superuser in the application database right after is imported - to be used with extreme care (by default empty). Only available in microservice type.
schemaOnly *bool*	When set to true, only the pre-data and post-data sections of pg_restore are invoked, avoiding data import. Default: false.

ImportSource

Appears in:

- Import {postgresql-cnpg-io-v1-Import}

ImportSource describes the source for the logical snapshot

Field	Description
externalCluster **[Required]** *string*	The name of the externalCluster used for import

InstanceID

Appears in:

- BackupStatus {postgresql-cnpg-io-v1-BackupStatus}

InstanceID contains the information to identify an instance

Field	Description
podName *string*	The pod name
ContainerID *string*	The container ID

InstanceReportedState

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

InstanceReportedState describes the last reported state of an instance during a reconciliation loop

Field	Description
isPrimary **[Required]** *bool*	indicates if an instance is the primary one
timeLineID *int*	indicates on which TimelineId the instance is

LDAPBindAsAuth

Appears in:

- LDAPConfig {postgresql-cnpg-io-v1-LDAPConfig}

LDAPBindAsAuth provides the required fields to use the bind authentication for LDAP

Field	Description
prefix *string*	Prefix for the bind authentication option
suffix *string*	Suffix for the bind authentication option

LDAPBindSearchAuth

Appears in:

- LDAPConfig {postgresql-cnpg-io-v1-LDAPConfig}

LDAPBindSearchAuth provides the required fields to use the bind+search LDAP authentication process

Field	Description
baseDN *string*	Root DN to begin the user search
bindDN *string*	DN of the user to bind to the directory
bindPassword *core/v1.SecretKeySelector*	Secret with the password for the user to bind to the directory
searchAttribute *string*	Attribute to match against the username
searchFilter *string*	Search filter to use when doing the search+bind authentication

LDAPConfig

Appears in:

- PostgresConfiguration {postgresql-cnpg-io-v1-PostgresConfiguration}

LDAPConfig contains the parameters needed for LDAP authentication

Field	Description
server *string*	LDAP hostname or IP address
port *int*	LDAP server port
scheme *LDAPScheme*	LDAP schema to be used, possible options are ldap and ldaps
bindAsAuth *LDAPBindAsAuth*	Bind as authentication configuration
bindSearchAuth *LDAPBindSearchAuth*	Bind+Search authentication configuration
tls *bool*	Set to 'true' to enable LDAP over TLS. 'false' is default

LDAPScheme

(Alias of `string`)

Appears in:

- LDAPConfig {postgresql-cnpg-io-v1-LDAPConfig}

LDAPScheme defines the possible schemes for LDAP

LocalObjectReference

Appears in:

- BackupSource {postgresql-cnpg-io-v1-BackupSource}
- BackupSpec {postgresql-cnpg-io-v1-BackupSpec}
- BootstrapInitDB {postgresql-cnpg-io-v1-BootstrapInitDB}
- BootstrapPgBaseBackup {postgresql-cnpg-io-v1-BootstrapPgBaseBackup}
- BootstrapRecovery {postgresql-cnpg-io-v1-BootstrapRecovery}
- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}
- ConfigMapKeySelector {postgresql-cnpg-io-v1-ConfigMapKeySelector}
- PgBouncerSpec {postgresql-cnpg-io-v1-PgBouncerSpec}
- PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec}
- RoleConfiguration {postgresql-cnpg-io-v1-RoleConfiguration}
- ScheduledBackupSpec {postgresql-cnpg-io-v1-ScheduledBackupSpec}
- SecretKeySelector {postgresql-cnpg-io-v1-SecretKeySelector}

LocalObjectReference contains enough information to let you locate a local object with a known type inside the same namespace

Field	Description
name **[Required]** *string*	Name of the referent.

ManagedConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ManagedConfiguration represents the portions of PostgreSQL that are managed by the instance manager

Field	Description
roles *RoleConfiguration*	Database roles managed by the Cluster

ManagedRoles

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

ManagedRoles tracks the status of a cluster' s managed roles

Field	Description
byStatus *map[RoleStatus][]string*	ByStatus gives the list of roles in each state
cannotReconcile *map[string][]string*	CannotReconcile lists roles that cannot be reconciled in PostgreSQL, with an explanation of the cause
passwordStatus *map[string]PasswordState*	PasswordStatus gives the last transaction id and password secret version for each managed role

Metadata

Appears in:

- PodTemplateSpec {postgresql-cnpg-io-v1-PodTemplateSpec}
- ServiceAccountTemplate {postgresql-cnpg-io-v1-ServiceAccountTemplate}
- ServiceTemplateSpec {postgresql-cnpg-io-v1-ServiceTemplateSpec}

Metadata is a structure similar to the metav1.ObjectMeta, but still parseable by controller-gen to create a suitable CRD for the user. The comment of PodTemplateSpec has an explanation of why we are not using the core data types.

Field	Description
labels *map[string]string*	Map of string keys and values that can be used to organize and categorize (scope and select) objects. May match selectors of replication controllers and services. More info: http://kubernetes.io/docs/user-guide/labels
annotations *map[string]string*	Annotations is an unstructured key value map stored with a resource that may be set by external tools to store and retrieve arbitrary metadata. They are not queryable and should be preserved when modifying objects. More info: http://kubernetes.io/docs/user-guide/annotations

MonitoringConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

MonitoringConfiguration is the type containing all the monitoring configuration for a certain cluster

Field	Description
disableDefaultQueries *bool*	Whether the default queries should be injected. Set it to true if you don't want to inject default queries into the cluster. Default: false.
customQueriesConfigMap *ConfigMapKeySelector*	The list of config maps containing the custom queries
customQueriesSecret *SecretKeySelector*	The list of secrets containing the custom queries
enablePodMonitor *bool*	Enable or disable the PodMonitor
podMonitorMetricRelabelings *archive_timeout*	The list of metric relabelings for the PodMonitor. Applied to samples before ingestion.
podMonitorRelabelings *archive_timeout*	The list of relabelings for the PodMonitor. Applied to samples before scraping.

NodeMaintenanceWindow

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

NodeMaintenanceWindow contains information that the operator will use while upgrading the underlying node.

This option is only useful when the chosen storage prevents the Pods from being freely moved across nodes.

Field	Description
reusePVC *bool*	Reuse the existing PVC (wait for the node to come up again) or not (recreate it elsewhere- when instances >1)
inProgress *bool*	Is there a node maintenance activity in progress?

OnlineConfiguration

Appears in:

- BackupSpec {postgresql-cnpg-io-v1-BackupSpec}
- ScheduledBackupSpec {postgresql-cnpg-io-v1-ScheduledBackupSpec}
- VolumeSnapshotConfiguration {postgresql-cnpg-io-v1-VolumeSnapshotConfiguration}

OnlineConfiguration contains the configuration parameters for the online volume snapshot

Field	Description
waitForArchive *bool*	If false, the function will return immediately after the backup is completed, without waiting for WAL to be archived. This behavior is only useful with backup software that independently monitors WAL archiving. Otherwise, WAL required to make the backup consistent might be missing and make the backup useless. By default, or when this parameter is true, pg_backup_stop will wait for WAL to be archived when archiving is enabled. On a standby, this means that it will wait only when archive_mode = always. If write activity on the primary is low, it may be useful to run pg_switch_wal on the primary in order to trigger an immediate segment switch.
immediateCheckpoint *bool*	Control whether the I/O workload for the backup initial checkpoint will be limited, according to the checkpoint_completion_target setting on the PostgreSQL server. If set to true, an immediate checkpoint will be used, meaning PostgreSQL will complete the checkpoint as soon as possible. false by default.

PasswordState

Appears in:

- ManagedRoles {postgresql-cnpg-io-v1-ManagedRoles}

PasswordState represents the state of the password of a managed RoleConfiguration

Field	Description
transactionID *int64*	the last transaction ID to affect the role definition in PostgreSQL
resourceVersion *string*	the resource version of the password secret

PgBouncerIntegrationStatus

Appears in:

- PoolerIntegrations {postgresql-cnpg-io-v1-PoolerIntegrations}

PgBouncerIntegrationStatus encapsulates the needed integration for the pgbouncer poolers referencing the cluster

Field	Description
secrets *string*	No description provided.

PgBouncerPoolMode

(Alias of `string`)

Appears in:

- PgBouncerSpec {postgresql-cnpg-io-v1-PgBouncerSpec}

PgBouncerPoolMode is the mode of PgBouncer

PgBouncerSecrets

Appears in:

- PoolerSecrets {postgresql-cnpg-io-v1-PoolerSecrets}

PgBouncerSecrets contains the versions of the secrets used by pgbouncer

Field	Description
authQuery *SecretVersion*	The auth query secret version

PgBouncerSpec

Appears in:

- PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec}

PgBouncerSpec defines how to configure PgBouncer

Field	Description
poolMode *PgBouncerPoolMode*	The pool mode. Default: session.
authQuerySecret *LocalObjectReference*	The credentials of the user that need to be used for the authentication query. In case it is specified, also an AuthQuery (e.g. "SELECT username, passwd FROM pg_catalog.pg_shadow WHERE usename=$1") has to be specified and no automatic CNPG Cluster integration will be triggered.

continues on next page

Table 60 – continued from previous page

Field	Description
authQuery *string*	The query that will be used to download the hash of the password of a certain user. Default: "SELECT usename, passwd FROM public.user_search($1)". In case it is specified, also an AuthQuerySecret has to be specified and no automatic CNPG Cluster integration will be triggered.
parameters *map[string]string*	Additional parameters to be passed to PgBouncer- please check the CNPG documentation for a list of options you can configure
pg_hba *string*	PostgreSQL Host Based Authentication rules (lines to be appended to the pg_hba.conf file)
paused *bool*	When set to true, PgBouncer will disconnect from the PostgreSQL server, first waiting for all queries to complete, and pause all new client connections until this value is set to false (default). Internally, the operator calls PgBouncer's PAUSE and RESUME commands.

PluginStatus

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

PluginStatus is the status of a loaded plugin

Field	Description
name **[Required]** *string*	Name is the name of the plugin
version **[Required]** *string*	Version is the version of the plugin loaded by the latest reconciliation loop
capabilities **[Required]** *string*	Capabilities are the list of capabilities of the plugin
operatorCapabilities **[Required]** *string*	OperatorCapabilities are the list of capabilities of the plugin regarding the reconciler
walCapabilities **[Required]** *string*	WALCapabilities are the list of capabilities of the plugin regarding the WAL management
backupCapabilities **[Required]** *string*	BackupCapabilities are the list of capabilities of the plugin regarding the Backup management

PodTemplateSpec

Appears in:

- PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec}

PodTemplateSpec is a structure allowing the user to set a template for Pod generation.

Unfortunately we can’t use the corev1.PodTemplateSpec type because the generated CRD won’t have the field for the metadata section.

References: https://github.com/kubernetes-sigs/controller-tools/issues/385 https://github.com/kubernetes-sigs/controller-tools/issues/448 https://github.com/prometheus-operator/prometheus-operator/issues/3041

Field	Description
metadata *Metadata*	Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata
spec *core/v1.PodSpec*	Specification of the desired behavior of the pod. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

PodTopologyLabels

(Alias of `map[string]string`)

Appears in:

- PodTopologyLabels {postgresql-cnpg-io-v1-PodTopologyLabels}

PodTopologyLabels represent the topology of a Pod. map[labelName]labelValue

PoolerIntegrations

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

PoolerIntegrations encapsulates the needed integration for the poolers referencing the cluster

Field	Description
pgBouncerIntegration *PgBouncerIntegrationStatus*	No description provided.

PoolerMonitoringConfiguration

Appears in:

- PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec}

PoolerMonitoringConfiguration is the type containing all the monitoring configuration for a certain Pooler.

Mirrors the Cluster' s MonitoringConfiguration but without the custom queries part for now.

Field	Description
enablePodMonitor *bool*	Enable or disable the PodMonitor
podMonitorMetricRelabelings *archive_timeout*	The list of metric relabelings for the PodMonitor. Applied to samples before ingestion.
podMonitorRelabelings *archive_timeout*	The list of relabelings for the PodMonitor. Applied to samples before scraping.

PoolerSecrets

Appears in:

- PoolerStatus {postgresql-cnpg-io-v1-PoolerStatus}

PoolerSecrets contains the versions of all the secrets used

Field	Description
serverTLS *SecretVersion*	The server TLS secret version
serverCA *SecretVersion*	The server CA secret version
clientCA *SecretVersion*	The client CA secret version
pgBouncerSecrets *PgBouncerSecrets*	The version of the secrets used by PgBouncer

PoolerSpec

Appears in:

- Pooler {postgresql-cnpg-io-v1-Pooler}

PoolerSpec defines the desired state of Pooler

Field	Description
cluster ** Required]** [*LocalObjectReference*	This is the cluster reference on which the Pooler will work. Pooler name should never match with any cluster name within the same namespace.
type *PoolerType*	Type of service to forward traffic to. Default: rw.
instances *int32*	The number of replicas we want. Default: 1.
template *PodTemplateSpec*	The template of the Pod to be created
pgbouncer ** Required]** [*PgBouncerSpec*	The PgBouncer configuration
deploymentStrategy *apps/v1.DeploymentStrategy*	The deployment strategy to use for pgbouncer to replace existing pods with new ones
monitoring *archive_timeout*	The configuration of the monitoring infrastructure of this pooler.
serviceTemplate *ServiceTemplateSpec*	Template for the Service to be created

PoolerStatus

Appears in:

- Pooler {postgresql-cnpg-io-v1-Pooler}

PoolerStatus defines the observed state of Pooler

Field	Description
secrets *PoolerSecrets*	The resource version of the config object
instances *int32*	The number of pods trying to be scheduled

PoolerType

(Alias of `string`)

Appears in:

- PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec}

PoolerType is the type of the connection pool, meaning the service we are targeting. Allowed values are rw and ro.

PostInitApplicationSQLRefs

Appears in:

- BootstrapInitDB {postgresql-cnpg-io-v1-BootstrapInitDB}

PostInitApplicationSQLRefs points references to ConfigMaps or Secrets which contain SQL files, the general implementation order to these references is from all Secrets to all ConfigMaps, and inside Secrets or ConfigMaps, the implementation order is same as the order of each array

Field	Description
secretRefs *SecretKeySelector*	SecretRefs holds a list of references to Secrets
configMapRefs *ConfigMapKeySelector*	ConfigMapRefs holds a list of references to ConfigMaps

PostgresConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

PostgresConfiguration defines the PostgreSQL configuration

Field	Description
parameters *map[string]string*	PostgreSQL configuration options (postgresql.conf)
pg_hba *string*	PostgreSQL Host Based Authentication rules (lines to be appended to the pg_hba.conf file)
pg_ident *string*	PostgreSQL User Name Maps rules (lines to be appended to the pg_ident.conf file)
syncReplicaElectionConstraint *SyncReplicaElectionConstraints*	Requirements to be met by sync replicas. This will affect how the "synchronous_standby_names" parameter will be set up.
shared_preload_libraries *string*	Lists of shared preload libraries to add to the default ones
ldap *LDAPConfig*	Options to specify LDAP configuration
promotionTimeout *int32*	Specifies the maximum number of seconds to wait when promoting an instance to primary. Default value is 40000000, greater than one year in seconds, big enough to simulate an infinite timeout
enableAlterSystem *bool*	If this parameter is true, the user will be able to invoke ALTER SYSTEM on this CloudNativePG Cluster. This should only be used for debugging and troubleshooting. Defaults to false.

PrimaryUpdateMethod

(Alias of `string`)

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

PrimaryUpdateMethod contains the method to use when upgrading the primary server of the cluster as part of rolling updates

PrimaryUpdateStrategy

(Alias of `string`)

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

PrimaryUpdateStrategy contains the strategy to follow when upgrading the primary server of the cluster as part of rolling updates

RecoveryTarget

Appears in:

- BootstrapRecovery {postgresql-cnpg-io-v1-BootstrapRecovery}

RecoveryTarget allows to configure the moment where the recovery process will stop. All the target options except TargetTLI are mutually exclusive.

Field	Description
backupID *string*	The ID of the backup from which to start the recovery process. If empty (default) the operator will automatically detect the backup based on targetTime or targetLSN if specified. Otherwise use the latest available backup in chronological order.
targetTLI *string*	The target timeline ("latest" or a positive integer)
targetXID *string*	The target transaction ID
targetName *string*	The target name (to be previously created with pg_create_restore_point)
targetLSN *string*	The target LSN (Log Sequence Number)
targetTime *string*	The target time as a timestamp in the RFC3339 standard
targetImmediate *bool*	End recovery as soon as a consistent state is reached
exclusive *bool*	Set the target to be exclusive. If omitted, defaults to false, so that in Postgres, recovery_target_inclusive will be true

ReplicaClusterConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ReplicaClusterConfiguration encapsulates the configuration of a replica cluster

Field	Description
source **[Required]** *string*	The name of the external cluster which is the replication origin
enabled **[Required]** *bool*	If replica mode is enabled, this cluster will be a replica of an existing cluster. Replica cluster can be created from a recovery object store or via streaming through pg_basebackup. Refer to the Replica clusters page of the documentation for more information.

ReplicationSlotsConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ReplicationSlotsConfiguration encapsulates the configuration of replication slots

Field	Description
highAvailability *archive_timeout*	Replication slots for high availability configuration
updateInterval *int*	Standby will update the status of the local replication slots every updateInterval seconds (default 30).
synchronizeReplicas *SynchronizeReplicasConfiguration*	Configures the synchronization of the user defined physical replication slots

ReplicationSlotsHAConfiguration

Appears in:

- ReplicationSlotsConfiguration {postgresql-cnpg-io-v1-ReplicationSlotsConfiguration}

ReplicationSlotsHAConfiguration encapsulates the configuration of the replication slots that are automatically managed by the operator to control the streaming replication connections with the standby instances for high availability (HA) purposes. Replication slots are a PostgreSQL feature that makes sure that PostgreSQL automatically keeps WAL files in the primary when a streaming client (in this specific case a replica that is part of the HA cluster) gets disconnected.

Field	Description
enabled *bool*	If enabled (default), the operator will automatically manage replication slots on the primary instance and use them in streaming replication connections with all the standby instances that are part of the HA cluster. If disabled, the operator will not take advantage of replication slots in streaming connections with the replicas. This feature also controls replication slots in replica cluster, from the designated primary to its cascading replicas.
slotPrefix *string*	Prefix for replication slots managed by the operator for HA. It may only contain lower case letters, numbers, and the underscore character. This can only be set at creation time. By default set to _cnpg_.

RoleConfiguration

Appears in:

- ManagedConfiguration {postgresql-cnpg-io-v1-ManagedConfiguration}

RoleConfiguration is the representation, in Kubernetes, of a PostgreSQL role with the additional field Ensure specifying whether to ensure the presence or absence of the role in the database

The defaults of the CREATE ROLE command are applied Reference: https://www.postgresql.org/docs/current/sql-createrole.html

Field	Description
name **[Required]** *string*	Name of the role
comment *string*	Description of the role
ensure *EnsureOption*	Ensure the role is present or absent- defaults to "present"
passwordSecret *LocalObjectReference*	Secret containing the password of the role (if present) If null, the password will be ignored unless DisablePassword is set
connectionLimit *int64*	If the role can log in, this specifies how many concurrent connections the role can make. -1 (the default) means no limit.
validUntil *meta/v1.Time*	Date and time after which the role's password is no longer valid. When omitted, the password will never expire (default).
inRoles *string*	List of one or more existing roles to which this role will be immediately added as a new member. Default empty.
inherit *bool*	Whether a role "inherits" the privileges of roles it is a member of. Defaults is true.
disablePassword *bool*	DisablePassword indicates that a role's password should be set to NULL in Postgres
superuser *bool*	Whether the role is a superuser who can override all access restrictions within the database- superuser status is dangerous and should be used only when really needed. You must yourself be a superuser to create a new superuser. Defaults is false.
createdb *bool*	When set to true, the role being defined will be allowed to create new databases. Specifying false (default) will deny a role the ability to create databases.
createrole *bool*	Whether the role will be permitted to create, alter, drop, comment on, change the security label for, and grant or revoke membership in other roles. Default is false.

continues on next page

Table 74 – continued from previous page

Field	Description
login *bool*	Whether the role is allowed to log in. A role having the login attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges, but are not users in the usual sense of the word. Default is false.
replication *bool*	Whether a role is a replication role. A role must have this attribute (or be a superuser) in order to be able to connect to the server in replication mode (physical or logical replication) and in order to be able to create or drop replication slots. A role having the replication attribute is a very highly privileged role, and should only be used on roles actually used for replication. Default is false.
bypassrls *bool*	Whether a role bypasses every row-level security (RLS) policy. Default is false.

S3Credentials

Appears in:

- BarmanCredentials {postgresql-cnpg-io-v1-BarmanCredentials}

S3Credentials is the type for the credentials to be used to upload files to S3. It can be provided in two alternative ways:

- explicitly passing accessKeyId and secretAccessKey
- inheriting the role from the pod environment by setting inheritFromIAMRole to true

Field	Description
accessKeyId *SecretKeySelector*	The reference to the access key id
secretAccessKey *SecretKeySelector*	The reference to the secret access key
region *SecretKeySelector*	The reference to the secret containing the region name
sessionToken *SecretKeySelector*	The references to the session key
inheritFromIAMRole *bool*	Use the role based authentication without providing explicitly the keys.

ScheduledBackupSpec

Appears in:

- ScheduledBackup {postgresql-cnpg-io-v1-ScheduledBackup}

ScheduledBackupSpec defines the desired state of ScheduledBackup

Field	Description
suspend *bool*	If this backup is suspended or not
immediate *bool*	If the first backup has to be immediately start after creation or not

continues on next page

Table 76 – continued from previous page

Field	Description
schedule **[Required]** *string*	The schedule does not follow the same format used in Kubernetes CronJobs as it includes an additional seconds specifier, see https://pkg.go.dev/github.com/robfig/cron#hdr-CRON_Expression_Format
cluster ** Required]** [*LocalObjectReference*	The cluster to backup
backupOwnerReference *string*	Indicates which ownerReference should be put inside the created backup resources. - none: no owner reference for created backup objects (same behavior as before the field was introduced) - self: sets the Scheduled backup object as owner of the backup - cluster: set the cluster as owner of the backup
target *archive_timeout*	The policy to decide which instance should perform this backup. If empty, it defaults to cluster.spec.backup.target. Available options are empty string, primary and prefer-standby. primary to have backups run always on primary instances, prefer-standby to have backups run preferably on the most updated standby, if available.
method *archive_timeout*	The backup method to be used, possible options are barmanObjectStore and volumeSnapshot. Defaults to: barmanObjectStore.
pluginConfiguration *archive_timeout*	Configuration parameters passed to the plugin managing this backup
online *bool*	Whether the default type of backup with volume snapshots is online/hot (true, default) or offline/cold (false) Overrides the default setting specified in the cluster field '.spec.backup.volumeSnapshot.online'
onlineConfiguration *OnlineConfiguration*	Configuration parameters to control the online/hot backup with volume snapshots Overrides the default settings specified in the cluster '.backup.volumeSnapshot.onlineConfiguration' stanza

ScheduledBackupStatus

Appears in:

- ScheduledBackup {postgresql-cnpg-io-v1-ScheduledBackup}

ScheduledBackupStatus defines the observed state of ScheduledBackup

Field	Description
lastCheckTime *meta/v1.Time*	The latest time the schedule
lastScheduleTime *meta/v1.Time*	Information when was the last time that backup was successfully scheduled.
nextScheduleTime *meta/v1.Time*	Next time we will run a backup

SecretKeySelector

Appears in:

- AzureCredentials {postgresql-cnpg-io-v1-AzureCredentials}
- BackupSource {postgresql-cnpg-io-v1-BackupSource}
- BackupStatus {postgresql-cnpg-io-v1-BackupStatus}
- BarmanObjectStoreConfiguration {postgresql-cnpg-io-v1-BarmanObjectStoreConfiguration}
- GoogleCredentials {postgresql-cnpg-io-v1-GoogleCredentials}
- MonitoringConfiguration {postgresql-cnpg-io-v1-MonitoringConfiguration}
- PostInitApplicationSQLRefs {postgresql-cnpg-io-v1-PostInitApplicationSQLRefs}
- S3Credentials {postgresql-cnpg-io-v1-S3Credentials}

SecretKeySelector contains enough information to let you locate the key of a Secret

Field	Description
LocalObjectReference *LocalObjectReference*	(Members of LocalObjectReference are embedded into this type.) The name of the secret in the pod's namespace to select from.
key **[Required]** *string*	The key to select

SecretVersion

Appears in:

- PgBouncerSecrets {postgresql-cnpg-io-v1-PgBouncerSecrets}
- PoolerSecrets {postgresql-cnpg-io-v1-PoolerSecrets}

SecretVersion contains a secret name and its ResourceVersion

Field	Description
name *string*	The name of the secret
version *string*	The ResourceVersion of the secret

SecretsResourceVersion

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

SecretsResourceVersion is the resource versions of the secrets managed by the operator

Field	Description
superuserSecretVersion *string*	The resource version of the "postgres" user secret

continues on next page

Table 80 – continued from previous page

Field	Description
replicationSecretVersion *string*	The resource version of the "streaming_replica" user secret
applicationSecretVersion *string*	The resource version of the "app" user secret
managedRoleSecretVersion *map[string]string*	The resource versions of the managed roles secrets
caSecretVersion *string*	Unused. Retained for compatibility with old versions.
clientCaSecretVersion *string*	The resource version of the PostgreSQL client-side CA secret version
serverCaSecretVersion *string*	The resource version of the PostgreSQL server-side CA secret version
serverSecretVersion *string*	The resource version of the PostgreSQL server-side secret version
barmanEndpointCA *string*	The resource version of the Barman Endpoint CA if provided
externalClusterSecretVersion *map[string]string*	The resource versions of the external cluster secrets
metrics *map[string]string*	A map with the versions of all the secrets used to pass metrics. Map keys are the secret names, map values are the versions

ServiceAccountTemplate

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ServiceAccountTemplate contains the template needed to generate the service accounts

Field	Description
metadata ** Required]** [*Metadata*	Metadata are the metadata to be used for the generated service account

ServiceTemplateSpec

Appears in:

- PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec}

ServiceTemplateSpec is a structure allowing the user to set a template for Service generation.

Field	Description
metadata *Metadata*	Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata
spec *core/v1.ServiceSpec*	Specification of the desired behavior of the service. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

SnapshotOwnerReference

(Alias of `string`)

Appears in:

- VolumeSnapshotConfiguration {postgresql-cnpg-io-v1-VolumeSnapshotConfiguration}

SnapshotOwnerReference defines the reference type for the owner of the snapshot. This specifies which owner the processed resources should relate to.

SnapshotType

(Alias of `string`)

Appears in:

- Import {postgresql-cnpg-io-v1-Import}

SnapshotType is a type of allowed import

StorageConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}
- TablespaceConfiguration {postgresql-cnpg-io-v1-TablespaceConfiguration}

StorageConfiguration is the configuration used to create and reconcile PVCs, usable for WAL volumes, PGDATA volumes, or tablespaces

Field	Description
storageClass *string*	StorageClass to use for PVCs. Applied after evaluating the PVC template, if available. If not specified, the generated PVCs will use the default storage class
size *string*	Size of the storage. Required if not already specified in the PVC template. Changes to this field are automatically reapplied to the created PVCs. Size cannot be decreased.
resizeInUseVolumes *bool*	Resize existent PVCs, defaults to true
pvcTemplate *core/v1.PersistentVolumeClaimSpec*	Template to be used to generate the Persistent Volume Claim

SwitchReplicaClusterStatus

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

SwitchReplicaClusterStatus contains all the statuses regarding the switch of a cluster to a replica cluster

Field	Description
inProgress *bool*	InProgress indicates if there is an ongoing procedure of switching a cluster to a replica cluster.

SyncReplicaElectionConstraints

Appears in:

- PostgresConfiguration {postgresql-cnpg-io-v1-PostgresConfiguration}

SyncReplicaElectionConstraints contains the constraints for sync replicas election.

For anti-affinity parameters two instances are considered in the same location if all the labels values match.

In future synchronous replica election restriction by name will be supported.

Field	Description
nodeLabelsAntiAffinity *string*	A list of node labels values to extract and compare to evaluate if the pods reside in the same topology or not
enabled **[Required]** *bool*	This flag enables the constraints for sync replicas

SynchronizeReplicasConfiguration

Appears in:

- ReplicationSlotsConfiguration {postgresql-cnpg-io-v1-ReplicationSlotsConfiguration}

SynchronizeReplicasConfiguration contains the configuration for the synchronization of user defined physical replication slots

Field	Description
enabled **[Required]** *bool*	When set to true, every replication slot that is on the primary is synchronized on each standby
excludePatterns *string*	List of regular expression patterns to match the names of replication slots to be excluded (by default empty)
• ** Required]** [*synchronizeReplicasCache*	No description provided.

continues on next page

Table 86 – continued from previous page

Field	Description

TablespaceConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

TablespaceConfiguration is the configuration of a tablespace, and includes the storage specification for the tablespace

Field	Description
name **[Required]** *string*	The name of the tablespace
storage ** *archive_timeout*	The storage configuration for the tablespace
owner *DatabaseRoleRef*	Owner is the PostgreSQL user owning the tablespace
temporary *bool*	When set to true, the tablespace will be added as a temp_tablespaces entry in PostgreSQL, and will be available to automatically house temp database objects, or other temporary files. Please refer to PostgreSQL documentation for more information on the temp_tablespaces GUC.

TablespaceState

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

TablespaceState represents the state of a tablespace in a cluster

Field	Description
name **[Required]** *string*	Name is the name of the tablespace
owner *string*	Owner is the PostgreSQL user owning the tablespace
state ** *archive_timeout*	State is the latest reconciliation state
error *string*	Error is the reconciliation error, if any

TablespaceStatus

(Alias of `string`)

Appears in:

- TablespaceState {postgresql-cnpg-io-v1-TablespaceState}

TablespaceStatus represents the status of a tablespace in the cluster

Topology

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

Topology contains the cluster topology

Field	Description
instances *map[PodName]PodTopologyLabels*	Instances contains the pod topology of the instances
nodesUsed *int32*	NodesUsed represents the count of distinct nodes accommodating the instances. A value of '1' suggests that all instances are hosted on a single node, implying the absence of High Availability (HA). Ideally, this value should be the same as the number of instances in the Postgres HA cluster, implying shared nothing architecture on the compute side.
successfullyExtracted *bool*	SuccessfullyExtracted indicates if the topology data was extract. It is useful to enact fallback behaviors in synchronous replica election in case of failures

VolumeSnapshotConfiguration

Appears in:

- BackupConfiguration {postgresql-cnpg-io-v1-BackupConfiguration}

VolumeSnapshotConfiguration represents the configuration for the execution of snapshot backups.

Field	Description
labels *map[string]string*	Labels are key-value pairs that will be added to .metadata.labels snapshot resources.
annotations *map[string]string*	Annotations key-value pairs that will be added to .metadata.annotations snapshot resources.
className *string*	ClassName specifies the Snapshot Class to be used for PG_DATA PersistentVolumeClaim. It is the default class for the other types if no specific class is present
walClassName *string*	WalClassName specifies the Snapshot Class to be used for the PG_WAL PersistentVolumeClaim.
tablespaceClassName *map[string]string*	TablespaceClassName specifies the Snapshot Class to be used for the tablespaces. defaults to the PGDATA Snapshot Class, if set
snapshotOwnerReference *SnapshotOwnerReference*	SnapshotOwnerReference indicates the type of owner reference the snapshot should have
online *bool*	Whether the default type of backup with volume snapshots is online/hot (true, default) or offline/cold (false)
onlineConfiguration *OnlineConfiguration*	Configuration parameters to control the online/hot backup with volume snapshots

WalBackupConfiguration

Appears in:

- BarmanObjectStoreConfiguration {postgresql-cnpg-io-v1-BarmanObjectStoreConfiguration}

WalBackupConfiguration is the configuration of the backup of the WAL stream

Field	Description
compression *CompressionType*	Compress a WAL file before sending it to the object store. Available options are empty string (no compression, default), gzip, bzip2 or snappy.
encryption *EncryptionType*	Whenever to force the encryption of files (if the bucket is not already configured for that). Allowed options are empty string (use the bucket policy, default), AES256 and aws:kms
maxParallel *int*	Number of WAL files to be either archived in parallel (when the PostgreSQL instance is archiving to a backup object store) or restored in parallel (when a PostgreSQL standby is fetching WAL files from a recovery object store). If not specified, WAL files will be processed one at a time. It accepts a positive integer as a value- with 1 being the minimum accepted value.

1.4.55 Supported releases

- This page lists the status, timeline and policy for currently supported releases of CloudNativePG*.

We are committed to providing support for the latest minor release, with a dedication to launching a new minor release every two months. Each release remains fully supported until reaching its designated "End of Life" date, as outlined in the support status table for CloudNativePG releases . This includes an additional 3-month assistance window to facilitate seamless upgrade planning.

Supported releases of CloudNativePG include releases that are in the active maintenance window and are patched for security and bug fixes.

Subsequent patch releases on a minor release contain backward-compatible changes only.

- *Support policy*
- *Naming scheme*
- *Support status of CloudNativePG releases*
- *What we mean by support*

Support policy

We produce new builds of CloudNativePG for each commit.

Approximately every two months, we build a minor release and run through several additional tests as well as release qualification. We release patch versions for issues found in supported minor releases.

The various types of releases represent a different product quality level and level of assistance from the CloudNativePG community. For details on the support provided by the community, see *What we mean by support* .

Type	Support level	Quality and recommended Use
Development Build	No support	Dangerous, might not be fully reliable. Useful to experiment with.
Minor Release	Support provided until 3 months after the N+1 minor release (ex. 1.23 supported until 3 months after 1.24.0 is released)	
Patch	Same as the corresponding minor release	Users are encouraged to adopt patch releases as soon as they are available for a given release.
Security Patch	Same as a patch, however, it doesn't contain any additional code other than the security fix from the previous patch	Given the nature of security fixes, users are **strongly** encouraged to adopt security patches after release.

You can find available releases on the releases page .

You can find high-level more information for each minor and patch release in the *Release notes for 1.14.0 and earlier* .

Naming scheme

Our naming scheme is based on Semantic Versioning 2.0.0

as follows:

```
<major>.<minor>.<patch>
```

Where `<minor>` is increased for each release, and `<patch>` counts the number of patches for the current `<minor>` release. A patch is usually a small change relative to the `<minor>` release.

Git tags for versions are prepended with `v` .

Support status of CloudNativePG releases

Version	Currently supported	Release date	End of life	Supported Kubernetes versions	Tested, but not supported	Supported Postgres versions
1.23.x	Yes	April 24, 2024	~ October, 2024	1.27, 1.28, 1.29	1.23, 1.24, 1.25, 1.26	12 - 16
1.22.x	Yes	December 21, 2023	July 24, 2024	1.26, 1.27, 1.28	1.23, 1.24, 1.25, 1.29	12 - 16
1.21.x	Yes	October 12, 2023	May 24, 2024	1.25, 1.26, 1.27, 1.28	1.23, 1.24, 1.29	12 - 16
main	No, development only					12 - 16

The list of supported Kubernetes versions in the table depends on what the CloudNativePG maintainers think is reasonable to support and to test.

At the moment, the CloudNativePG community doesn't support or test any additional Kubernetes distribution, like Red Hat OpenShift. This might change in the future and, in that case, that would be reflected in an official policy written by the CloudNativePG maintainers.

Supported PostgreSQL versions

The list of supported Postgres versions in the previous table generally depends on what PostgreSQL versions were supported by the community at the time the minor version of CloudNativePG was released.

See the PostgreSQL Versioning Policy

page for more information about supported versions.

Note: Starting from November 9, 2023, Postgres 11 is no longer supported.

We also recommend that you regularly update your PostgreSQL operand images and use the latest minor release for the major version you have in use, as not upgrading is riskier than upgrading. As a result, when opening an issue with an older minor version of PostgreSQL, we might not be able to help you.

Upcoming releases

Version	Release date	End of life	Supported Kubernetes versions
1.23.0	April 23, 2024	•	•
1.24.0	June/July, 2024	•	•

Note: Feature freeze happens one week before the release

Important: Dates in the future are uncertain and might change. This applies to Kubernetes versions, too. Updates and changes on the release schedule will be communicated in the Release updates

discussion in the main GitHub repository.

Old releases

Version	Release date	End of life	Compatible Kubernetes versions
1.20.x	April 27, 2023	January 21, 2024	1.24, 1.25, 1.26, 1.27
1.19.x	February 14, 2023	November 3, 2023	1.23, 1.24, 1.25, 1.26
1.18.x	Nov 10, 2022	June 12, 2023	1.23, 1.24, 1.25, 1.26, 1.27
1.17.x	September 6, 2022	March 20, 2023	1.22, 1.23, 1.24
1.16.x	July 7, 2022	December 21, 2022	1.22, 1.23, 1.24
1.15.x	April 21, 2022	October 6, 2022	1.21, 1.22, 1.23

What we mean by support

Our support window is roughly five months for each release branch (latest minor release, plus 3 additional months), given that we produce a new final release every two months.

In the following diagram, `release-1.23` is an example of a release branch.

For example, if the latest release is `v1.23.0` , you can expect a supplementary 3-month support period for the preceding release, `v1.22.x` .

Only the last patch release of each branch is supported.

```
- -----+----------------------------------------------> main (trunk development)
       \             \
        \             \
         \             \             v1.23.0
          \             \            Apr 24, 2024                  ^
           \             \----------+---------------> release-1.23  |
            \                                                       | SUPPORTED
             \                                                      | RELEASES
              \   v1.22.0                                           | = last minor
               \  Dec 21, 2023                                      |   release +
                +-------------------+---------------> release-1.22  |   3 months
                                                                    v
```

We offer two types of support:

Technical support : Technical assistance is offered on a best-effort basis for supported releases only. You can request support from the community on the CloudNativePG Slack (in the `#general` channel), or using GitHub Discussions .

Security and bug fixes : We backport important bug fixes — including security fixes - to all currently supported releases. Before backporting a patch, we ask ourselves: *"Does this backport improve ``CloudNativePG``, bearing in mind that we really value stability for already-released versions?"*

If you' re looking for professional support, see the Support page in the website . The vendors listed there might provide service level agreements that included extended support timeframes.

1.4.56 Release notes

History of user-visible changes for CloudNativePG, classified for each minor release.

- *Release notes for CloudNativePG 1.23*
- *Release notes for CloudNativePG 1.22*
- *Release notes for CloudNativePG 1.21*

For information on the community support policy for CloudNativePG, please refer to *Supported releases* .

Older releases:

- *Release notes for CloudNativePG 1.20*
- *Release notes for CloudNativePG 1.19*
- *Release notes for CloudNativePG 1.18*
- *Release notes for CloudNativePG 1.17*
- *Release notes for CloudNativePG 1.16*
- *Release notes for CloudNativePG 1.15*

We also keep record of all the *release notes from 1.0.0 to 1.14.0 of Cloud Native PostgreSQL by EDB* , the predecessor of CloudNativePG.

Release notes for CloudNativePG 1.23

History of user-visible changes in the 1.23 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

Version 1.23.1

Release date: Apr 29, 2024

Fixes:

- Corrects the reconciliation of `PodMonitor` resources, which was failing due to a regression (#4286)

Version 1.23.0

Release date: Apr 24, 2024

Features:

- **PostgreSQL Image Catalogs:** Introduced `ClusterImageCatalog` and `ImageCatalog` CRDs to manage operand container images based on PostgreSQL major version. This is facilitated through the `Cluster` ' s `.spec.imageCatalogRef stanza` . This feature provides an alternative to the imageName option and will eventually replace it as the default method to define operand container images.
- **User-Defined Replication Slots:** Enhanced the synchronization of physical replication slots to cover user-defined replication slots on the primary, via the newly introduced stanza `replicationSlots.synchronizeReplicas` .

- **Configuration of Pod Disruption Budgets (PDB)**: Introduced the `.spec.enablePDB` field to disable PDBs on the primary instance, allowing proper eviction of the pod during maintenance operations. This is particularly useful for single-instance deployments. This feature is intended to replace the node maintenance window feature.

Enhancements:

- Users now have the capability to transition an existing cluster into replica mode, simplifying cross-datacenter switchover operations (#4261)
- Users can now customize the connection pooler service, including its type, labels, and annotations (#3384)
- Users can now configure the `wal_log_hints` PostgreSQL parameter (#4218) (#4218)
- Fully Qualified Domain Names (FQDN) in URIs for automatically generated secrets (#4095)
- Cleanup of instance Pods not owned by the Cluster during Cluster restore (#4141)
- Command output of the plugin' s `status` command to show the status of PDBs (#4319)
- Error detection when invoking `barman-cloud-wal-restore` in `recovery` bootstrap (#4101)

Fixes:

- Ensured that before a switchover, the elected replica is in streaming replication (#4288)
- Correctly handle parsing errors of instances' LSN when sorting them (#4283)
- Recreate the primary Pod if there are no healthy standbys available to promote (#4132)
- Cleanup `PGDATA` in case of failure of the restore job (#4151)
- Reload certificates on configuration update (#3705)
- `cnpg` plugin for `kubectl` :
  - Improve the arguments handling of `destroy` , `fencing` , and `promote` plugin commands (#4280)
  - Correctly handle the percentage of the backup progress in `cnpg status` (#4131)
  - Gracefully handle databases with no sequences in `sync-sequences` command (#4346)

Changes:

- Operator images are now based on `gcr.io/distroless/static-debian12:nonroot` (#4201)
- The Grafana dashboard now resides at https://github.com/cloudnative-pg/grafana-dashboards (#4154)

Release notes for CloudNativePG 1.22

History of user-visible changes in the 1.22 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

Version 1.22.3

Release date: Apr 24, 2024

Enhancements:

- Users can now configure the `wal_log_hints` PostgreSQL parameter (#4218) (#4218)
- Fully Qualified Domain Names (FQDN) in URIs for automatically generated secrets (#4095)
- Cleanup of instance Pods not owned by the Cluster during Cluster restore (#4141)
- Error detection when invoking `barman-cloud-wal-restore` in `recovery` bootstrap (#4101)

Fixes:

- Ensured that before a switchover, the elected replica is in streaming replication (#4288)
- Correctly handle parsing errors of instances' LSN when sorting them (#4283)
- Recreate the primary Pod if there are no healthy standbys available to promote (#4132)
- Cleanup `PGDATA` in case of failure of the restore job (#4151)
- Reload certificates on configuration update (#3705)
- `cnpg` plugin for `kubectl` :
  - Improve the arguments handling of `destroy` , `fencing` , and `promote` plugin commands (#4280)
  - Correctly handle the percentage of the backup progress in `cnpg status` (#4131)
  - Gracefully handle databases with no sequences in `sync-sequences` command (#4346)

Changes:

- The Grafana dashboard now resides at https://github.com/cloudnative-pg/grafana-dashboards (#4154)

Version 1.22.2

Release date: Mar 14, 2024

Enhancements

- Allow customization of the `wal_level` GUC in PostgreSQL (#4020)
- Add the `cnpg.io/skipWalArchiving` annotation to disable WAL archiving when set to `enabled` (#4055)
- Enrich the `cnpg` plugin for `kubectl` with the `publication` and `subscription` command groups to imperatively set up PostgreSQL native logical replication (#4052)
- Allow customization of `CERTIFICATE_DURATION` and `EXPIRING_CHECK_THRESHOLD` for automated management of TLS certificates handled by the operator (#3686)
- Retrieve the correct architecture' s binary from the corresponding catalog in the running operator image during in-place updates, enabling the operator to inject the correct binary into any Pod with a supported architecture (#3840)
- Introduce initial support for tab-completion with the `cnpg` plugin for `kubectl` (#3875)

Fixes

- Properly synchronize PVC group labels with those on the pods, a critical aspect when all pods are deleted and the operator needs to decide which Pod to recreate first (#3930)
- Disable `wal_sender_timeout` when cloning a replica to prevent timeout errors due to slow connections (#4080)
- Ensure that volume snapshots are ready before initiating recovery bootstrap procedures, preventing an error condition where recovery with incomplete backups could enter an error loop (#3663)
- Prevent an error loop when unsetting connection limits in managed roles (#3832)
- Resolve a corner case in hibernation where the instance pod has been deleted, but the cluster status still has the hibernation condition set to false (#3970)
- Correctly detect Google Cloud capabilities for Barman Cloud (#3931)

Security

- Use `Role` instead of `ClusterRole` for operator permissions in OLM, requiring fewer privileges when installed on a per-namespace basis (#3855, #3990)
- Enforce fully-qualified object names in SQL queries for the PgBouncer pooler (#4080)

Changes

- Follow Kubernetes recommendations to switch from client-side to server-side application of manifests, requiring the `--server-side` option by default when installing the operator (#3729).
- Set the default operand image to PostgreSQL 16.2 (#3823).

Version 1.22.1

Release date: Feb 2, 2024

Enhancements:

- Tailor ephemeral volume storage in a Postgres cluster using a claim template through the `ephemeralVolumeSource` option (#3678)
- Introduce the `pgadmin4` command in the `cnpg` plugin for `kubectl` , providing a straightforward method to demonstrate connecting to a given database cluster and navigate its content in a local environment such as kind - for evaluation purposes only (#3701)
- Allow customization of PostgreSQL' s ident map file via the `.spec.postgresql.pg_ident` stanza, through a list of user name maps (#3534)

Fixes:

- Prevent an unrecoverable issue with `pg_rewind` failing due to `postgresql.auto.conf` being read-only on clusters where the `ALTER SYSTEM` SQL command is disabled - the default (#3728)
- Proper recovery of tablespaces from volume snapshots (#3682)
- Reduce the risk of disk space shortage when using the import facility of the `initdb` bootstrap method, by disabling the durability settings in the PostgreSQL instance for the duration of the import process (#3743)
- Avoid pod restart due to erroneous resource quantity comparisons, e.g. "1 != 1000m" (#3706)

- Properly escape reserved characters in `pgpass` connection fields (#3713)
- Prevent systematic rollout of pods due to considering zero and nil different values in `.spec.projectedVolumeTemplate.sources` (#3647)
- Ensure configuration coherence by pruning from `postgresql.auto.conf` any options now incorporated into `override.conf` (#3773)

Version 1.22.0

Release date: Dec 21, 2023

Features:

- **Declarative Tablespaces**: Introducing the `tablespaces` stanza in the `Cluster` spec, enabling comprehensive lifecycle management of PostgreSQL tablespaces for enhanced vertical scalability (#3410).
- **Temporary Tablespaces**: Adding the `.spec.tablespaces[*].temporary` option to facilitate the utilization of a tablespace for temporary database operations, by incorporating the name into the `temp_tablespaces` PostgreSQL parameter (#3464).

Security:

- By default, TLSv1.3 is now enforced on all PostgreSQL 12 or higher installations. Additionally, users can configure the `ssl_ciphers` , `ssl_min_protocol_version` , and `ssl_max_protocol_version` GUCs (#3408).
- Integration of Docker image scanning with Dockle and Snyk to enhance security measures (#3300).

Enhancements:

- Improved reconciliation of external clusters (#3533).
- Introduction of the ability to enable/disable the `ALTER SYSTEM` command (#3535).
- Support for Prometheus' dynamic relabeling through the `podMonitorMetricRelabelings` and `podMonitorRelabelings` options in the `.spec.monitoring` stanza of the `Cluster` and `Pooler` resources (#3075).
- Enhanced computation of the first recoverability point and last successful backup by considering volume snapshots alongside object-store backups (#2940).
- Elimination of the use of the `PGPASSFILE` environment variable when establishing a network connection to PostgreSQL (#3522).
- Improved `cnpg report` plugin command by collecting a cluster' s PVCs (#3357).
- Enhancement of the `cnpg status` plugin command, providing information about managed roles, including alerts (#3310).
- Introduction of Red Hat UBI 8 container images for the operator, suitable for OLM deployments.
- Connection pooler:
  - Scaling down instances of a `Pooler` resource to 0 is now possible (#3517).
  - Addition of the `cnpg.io/podRole` label with a value of 'pooler' to every pooler deployment, differentiating them from instance pods (#3396).

Fixes:

- Reconciliation of metadata, annotations, and labels of `PodDisruptionBudget` resources (#3312 and #3434).
- Reconciliation of the metadata of the managed credential secrets (#3316).

- Resolution of a bug in the backup snapshot code where an error reading the body would be handled as an overall error, leaving the backup process indefinitely stuck (#3321).
- Implicit setting of online backup with the `cnpg backup` plugin command when either `immediate-checkpoint` or `wait-for-archive` options are requested (#3449).
- Disabling of wal_sender_timeout when joining through pg_basebackup (#3586)
- Reloading of secrets used by external clusters (#3565)
- Connection pooler:
  - Ensuring the controller watches all secrets owned by a `Pooler` resource (#3428).
  - Reconciliation of `RoleBinding` for `Pooler` resources (#3391).
  - Reconciliation of `imagePullSecret` for `Pooler` resources (#3389).
  - Reconciliation of the service of a `Pooler` and addition of the required labels (#3349).
  - Extension of `Pooler` labels to the deployment as well, not just the pods (#3350).

Changes:

- Default operand image set to PostgreSQL 16.1 (#3270).
- The `ALTER SYSTEM` command is now disabled by default (#3545).

Release notes for CloudNativePG 1.21

History of user-visible changes in the 1.21 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

Version 1.21.5

Release date: Apr 24, 2024

> **Warning:** Version 1.21 is approaching its End-of-Life (EOL) on May 24, 2024. If you haven't already, please begin planning for an upgrade promptly to ensure continued support and security.

Enhancements:

- Users can now configure the `wal_log_hints` PostgreSQL parameter (#4218) (#4218)
- Fully Qualified Domain Names (FQDN) in URIs for automatically generated secrets (#4095)
- Cleanup of instance Pods not owned by the Cluster during Cluster restore (#4141)
- Error detection when invoking `barman-cloud-wal-restore` in `recovery` bootstrap (#4101)

Fixes:

- Ensured that before a switchover, the elected replica is in streaming replication (#4288)
- Correctly handle parsing errors of instances' LSN when sorting them (#4283)
- Recreate the primary Pod if there are no healthy standbys available to promote (#4132)
- Cleanup `PGDATA` in case of failure of the restore job (#4151)

- Reload certificates on configuration update (#3705)
- `cnpg` plugin for `kubectl` :
  - Improve the arguments handling of `destroy` , `fencing` , and `promote` plugin commands (#4280)
  - Correctly handle the percentage of the backup progress in `cnpg status` (#4131)
  - Gracefully handle databases with no sequences in `sync-sequences` command (#4346)

Changes:

- The Grafana dashboard now resides at https://github.com/cloudnative-pg/grafana-dashboards (#4154)

Version 1.21.4

Release date: Mar 14, 2024

Enhancements

- Allow customization of the `wal_level` GUC in PostgreSQL (#4020)
- Add the `cnpg.io/skipWalArchiving` annotation to disable WAL archiving when set to `enabled` (#4055)
- Enrich the `cnpg` plugin for `kubectl` with the `publication` and `subscription` command groups to imperatively set up PostgreSQL native logical replication (#4052)
- Allow customization of `CERTIFICATE_DURATION` and `EXPIRING_CHECK_THRESHOLD` for automated management of TLS certificates handled by the operator (#3686)
- Introduce initial support for tab-completion with the `cnpg` plugin for `kubectl` (#3875)
- Retrieve the correct architecture' s binary from the corresponding catalog in the running operator image during in-place updates, enabling the operator to inject the correct binary into any Pod with a supported architecture (#3840)

Fixes

- Properly synchronize PVC group labels with those on the pods, a critical aspect when all pods are deleted and the operator needs to decide which Pod to recreate first (#3930)
- Disable `wal_sender_timeout` when cloning a replica to prevent timeout errors due to slow connections (#4080)
- Ensure that volume snapshots are ready before initiating recovery bootstrap procedures, preventing an error condition where recovery with incomplete backups could enter an error loop (#3663)
- Prevent an error loop when unsetting connection limits in managed roles (#3832)
- Resolve a corner case in hibernation where the instance pod has been deleted, but the cluster status still has the hibernation condition set to false (#3970)
- Correctly detect Google Cloud capabilities for Barman Cloud (#3931)

Security

- Use `Role` instead of `ClusterRole` for operator permissions in OLM, requiring fewer privileges when installed on a per-namespace basis (#3855, #3990)
- Enforce fully-qualified object names in SQL queries for the PgBouncer pooler (#4080)

Changes

- Follow Kubernetes recommendations to switch from client-side to server-side application of manifests, requiring the `--server-side` option by default when installing the operator (#3729).
- Set the default operand image to PostgreSQL 16.2 (#3823).

Version 1.21.3

Release date: Feb 2, 2024

Enhancements:

- Tailor ephemeral volume storage in a Postgres cluster using a claim template through the `ephemeralVolumeSource` option (#3678)
- Introduce the `pgadmin4` command in the `cnpg` plugin for `kubectl` , providing a straightforward method to demonstrate connecting to a given database cluster and navigate its content in a local environment such as kind - for evaluation purposes only (#3701)
- Allow customization of PostgreSQL' s ident map file via the `.spec.postgresql.pg_ident` stanza, through a list of user name maps (#3534)

Fixes:

- Prevent an unrecoverable issue with `pg_rewind` failing due to `postgresql.auto.conf` being read-only on clusters where the `ALTER SYSTEM` SQL command is disabled - the default (#3728)
- Reduce the risk of disk space shortage when using the import facility of the `initdb` bootstrap method, by disabling the durability settings in the PostgreSQL instance for the duration of the import process (#3743)
- Avoid pod restart due to erroneous resource quantity comparisons, e.g. "1 != 1000m" (#3706)
- Properly escape reserved characters in `pgpass` connection fields (#3713)
- Prevent systematic rollout of pods due to considering zero and nil different values in `.spec.projectedVolumeTemplate.sources` (#3647)
- Ensure configuration coherence by pruning from `postgresql.auto.conf` any options now incorporated into `override.conf` (#3773)

Version 1.21.2

Release date: Dec 21, 2023

Security:

- By default, TLSv1.3 is now enforced on all PostgreSQL 12 or higher installations. Additionally, users can configure the `ssl_ciphers` , `ssl_min_protocol_version` , and `ssl_max_protocol_version` GUCs (#3408).
- Integration of Docker image scanning with Dockle and Snyk to enhance security measures (#3300).

Enhancements:

- Improved reconciliation of external clusters (#3533).
- Introduction of the ability to enable/disable the `ALTER SYSTEM` command (#3535).
- Support for Prometheus' dynamic relabeling through the `podMonitorMetricRelabelings` and `podMonitorRelabelings` options in the `.spec.monitoring` stanza of the `Cluster` and `Pooler` resources (#3075).
- Enhanced computation of the first recoverability point and last successful backup by considering volume snapshots alongside object-store backups (#2940).
- Elimination of the use of the `PGPASSFILE` environment variable when establishing a network connection to PostgreSQL (#3522).
- Improved `cnpg report` plugin command by collecting a cluster' s PVCs (#3357).
- Enhancement of the `cnpg status` plugin command, providing information about managed roles, including alerts (#3310).
- Introduction of Red Hat UBI 8 container images for the operator, suitable for OLM deployments.
- Connection pooler:
  - Scaling down instances of a `Pooler` resource to 0 is now possible (#3517).
  - Addition of the `cnpg.io/podRole` label with a value of 'pooler' to every pooler deployment, differentiating them from instance pods (#3396).

Fixes:

- Reconciliation of metadata, annotations, and labels of `PodDisruptionBudget` resources (#3312 and #3434).
- Reconciliation of the metadata of the managed credential secrets (#3316).
- Resolution of a bug in the backup snapshot code where an error reading the body would be handled as an overall error, leaving the backup process indefinitely stuck (#3321).
- Implicit setting of online backup with the `cnpg backup` plugin command when either `immediate-checkpoint` or `wait-for-archive` options are requested (#3449).
- Disabling of wal_sender_timeout when joining through pg_basebackup (#3586)
- Reloading of secrets used by external clusters (#3565)
- Connection pooler:
  - Ensuring the controller watches all secrets owned by a `Pooler` resource (#3428).
  - Reconciliation of `RoleBinding` for `Pooler` resources (#3391).
  - Reconciliation of `imagePullSecret` for `Pooler` resources (#3389).
  - Reconciliation of the service of a `Pooler` and addition of the required labels (#3349).

  - Extension of `Pooler` labels to the deployment as well, not just the pods (#3350).

Changes:

- Default operand image set to PostgreSQL 16.1 (#3270).

Version 1.21.1

Release date: Nov 3, 2023

Enhancements:

- Introduce support for online/hot backups with volume snapshots by using the PostgreSQL API for physical online base backups. Default configuration for hot/cold backups on a given Postgres cluster can be controlled through the `online` option and the `onlineConfiguration` stanza in `.spec.backup.volumeSnapshot` . Unless explicitly set, backups on volume snapshots are now taken online by default (#3102)
- Introduce the possibility to override the above default settings on volume snapshot backup using the `ScheduledBackup` and `Backup` resources (#3208, #3226)
- Enhance cold backup on volume snapshots by reducing the time window in which the target instance (standby or primary) is fenced, by lifting it as soon as the volume snapshot have been cut and provisioned (#3210)
- During a recovery from volume snapshots, ensure that the provided volume snapshots are coherent by validating the existing labels and annotations
- The `backup` command of the `cnpg` plugin for `kubectl` improves the volume snapshot backup experience through the `--online` , `--immediate-checkpoint` , and `--wait-for-archive` runtime options
- Enhance the `status` command of the `cnpg` plugin for `kubectl` with progress information on active streaming base backups (#3101)
- Allow the configuration of `max_prepared_statements` with the pgBouncer `Pooler` resource (#3174)

Fixes:

- Suspend WAL archiving during a switchover and resume it when it is completed (#3227)
- Ensure that the instance manager always uses `synchronous_commit = local` when managing the PostgreSQL cluster (#3143)
- Custom certificates for streaming replication user through `.spec.certificates.replicationTLSSecret` are now working (#3209)
- Set the `cnpg.io/cluster` label to the `Pooler` pods (#3153)
- Reduce the number of labels in `VolumeSnapshots` resources and render them into more appropriate annotations (#3151)

Changes:

- Volume snapshot backups, introduced in 1.21.0, are now online/hot by default; in order to restore offline/cold backups set `.spec.backup.volumeSnapshot` to `false`
- Stop using the `postgresql.auto.conf` file inside PGDATA to control Postgres replication settings, and replace it with a file named `override.conf` (#2812)

Technical enhancements:

- Use extended query protocol for PostgreSQL in the instance manager (#3152)

Version 1.21.0

Release date: Oct 12, 2023

Features:

- **Volume Snapshot support for backup and recovery:** leverage the standard Kubernetes API on Volume Snapshots to take advantage of capabilities like incremental and differential copy for both backup and recovery operations. This first step, covering cold backups from a standby, will continue in 1.22 with support for hot backups using the PostgreSQL API and tablespaces.
- **OLM installation method**: introduce support for Operator Lifecycle Manager via OperatorHub.io for the latest patch version of the latest minor release through the *stable* channel. Many thanks to EDB for donating the bundle of their "EDB Postgres for Kubernetes" operator and adapting it for CloudNativePG.

Important Changes:

- Change the default value of `stopDelay` to 1800 seconds instead of 30 seconds (#2848)
- Introduce a new parameter, called `smartShutdownTimeout` , to control the window of time reserved for the smart shutdown of Postgres to complete; the general formula to compute the overall timeout to stop Postgres is `max(stopDelay - smartShutdownTimeout, 30)` (#2848)
- Change the default value of `startDelay` to 3600, instead of 30 seconds (#2847)
- Replace the livenessProbe initial delay with a more proper Kubernetes startup probe to deal with the start of a Postgres server (#2847)
- Change the default value of `switchoverDelay` to 3600 seconds instead of 40000000 seconds (#2846)
- Disable superuser access by default for security (#2905)
- Enable replication slots for HA by default (#2903)
- Stop supporting the `postgresql` label - replaced by `cnpg.io/cluster` in 1.18 (#2744)

Security:

- Add a default `seccompProfile` to the operator deployment (#2926)

Enhancements:

- Enable bootstrap of a replica cluster from a consistent set of volume snapshots (#2647)
- Enable full and Point In Time recovery from a consistent set of volume snapshots (#2390)
- Introduce the `cnpg.io/coredumpFilter` annotation to control the content of a core dump generated in the unlikely event of a PostgreSQL crash, by default set to exclude shared memory segments from the dump (#2733)
- Allow to configure ephemeral-storage limits for the shared memory and temporary data ephemeral volumes (#2830)
- Validate resource limits and requests through the webhook (#2663)
- Ensure that PostgreSQL's `shared_buffers` are coherent with the pods' allocated memory resources (#2840)
- Add `uri` and `jdbc-uri` fields in the credential secrets to facilitate developers when connecting their applications to the database (#2186)
- Add a new phase `Waiting for the instances to become active` for finer control of a cluster's state waiting for the replicas to be ready (#2612)
- Improve detection of Pod rollout conditions through the `podSpec` annotation (#2243)
- Add primary timestamp and uptime to the kubectl plugin's `status` command (#2953)

Fixes:

- Ensure that the primary instance is always recreated first by prioritizing ready PVCs with a primary role (#2544)
- Honor the `cnpg.io/skipEmptyWalArchiveCheck` annotation during recovery to bypass the check for an empty WAL archive (#2731)
- Prevent a cluster from being stuck when the PostgreSQL server is down but the pod is up on the primary (#2966)
- Avoid treating the designated primary in a replica cluster as a regular HA replica when replication slots are enabled (#2960)
- Reconcile services every time the selectors change or when labels/annotations need to be changed (#2918)
- Defaults to `app` both the owner and database during recovery bootstrap (#2957)
- Avoid write-read concurrency on cached cluster (#2884)
- Remove empty items, make them unique and sort in the `ResourceName` sections of the generated roles (#2875)
- Ensure that the `ContinuousArchiving` condition is properly set to 'failed' in case of errors (#2625)
- Make the `Backup` resource reconciliation cycle more resilient on interruptions by stopping only if the backup is completed or failed (#2591)
- Reconcile PodMonitor `labels` and `annotations` (#2583)
- Fix backup failure due to missing RBAC `resourceNames` on the `Role` object (#2956)
- Observability:
  - Add TCP port label to default `pg_stat_replication` metric (#2961)
  - Fix the `pg_wal_stat` default metric for Prometheus (#2569)
  - Improve the `pg_replication` default metric for Prometheus (#2744 and #2750)
  - Use `alertInstanceLabelFilter` instead of `alertName` in the provided Grafana dashboard
  - Enforce `standard_conforming_strings` in metric collection (#2888)

Changes:

- Set the default operand image to PostgreSQL 16.0
- Fencing now uses PostgreSQL's fast shutdown instead of smart shutdown to halt an instance (#3051)
- Rename webhooks from kb.io to cnpg.io group (#2851)
- Replace the `cnpg snapshot` command with `cnpg backup -m volumeSnapshot` for the `kubectl` plugin
- Let the `cnpg hibernate` plugin command use the `ClusterManifestAnnotationName` and `PgControldataAnnotationName` annotations on PVCs (#2657)
- Add the `cnpg.io/instanceRole` label while deprecating the existing `role` label (#2915)

Technical enhancements:

- Replace `k8s-api-docgen` with `gen-crd-api-reference-docs` to automatically build the API reference documentation (#2606)

old

Release notes for CloudNativePG 1.20

History of user-visible changes in the 1.20 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

Version 1.20.6

Release date: Feb 2, 2024

> **Warning:** This is expected to be the last release in the 1.20.X series. Users are encouraged to update to a newer minor version soon.

Enhancements:

- Tailor ephemeral volume storage in a Postgres cluster using a claim template through the `ephemeralVolumeSource` option (#3678)
- Introduce the `pgadmin4` command in the `cnpg` plugin for `kubectl` , providing a straightforward method to demonstrate connecting to a given database cluster and navigate its content in a local environment such as kind - for evaluation purposes only (#3701)
- Allow customization of PostgreSQL' s ident map file via the `.spec.postgresql.pg_ident` stanza, through a list of user name maps (#3534)

Fixes:

- Prevent an unrecoverable issue with `pg_rewind` failing due to `postgresql.auto.conf` being read-only on clusters where the `ALTER SYSTEM` SQL command is disabled - the default (#3728)
- Reduce the risk of disk space shortage when using the import facility of the `initdb` bootstrap method, by disabling the durability settings in the PostgreSQL instance for the duration of the import process (#3743)
- Avoid pod restart due to erroneous resource quantity comparisons, e.g. "1 != 1000m" (#3706)
- Properly escape reserved characters in `pgpass` connection fields (#3713)
- Prevent systematic rollout of pods due to considering zero and nil different values in `.spec.projectedVolumeTemplate.sources` (#3647)

Version 1.20.5

Release date: Dec 21, 2023

Security:

- By default, TLSv1.3 is now enforced on all PostgreSQL 12 or higher installations. Additionally, users can configure the `ssl_ciphers` , `ssl_min_protocol_version` , and `ssl_max_protocol_version` GUCs (#3408).
- Integration of Docker image scanning with Dockle and Snyk to enhance security measures (#3300).

Enhancements:

- Improved reconciliation of external clusters (#3533).

- Introduction of the ability to enable/disable the `ALTER SYSTEM` command (#3535).
- Support for Prometheus' dynamic relabeling through the `podMonitorMetricRelabelings` and `podMonitorRelabelings` options in the `.spec.monitoring` stanza of the `Cluster` and `Pooler` resources (#3075).
- Elimination of the use of the `PGPASSFILE` environment variable when establishing a network connection to PostgreSQL (#3522).
- Improved `cnpg report` plugin command by collecting a cluster's PVCs (#3357).
- Enhancement of the `cnpg status` plugin command, providing information about managed roles, including alerts (#3310).
- Connection pooler:
  - Scaling down instances of a `Pooler` resource to 0 is now possible (#3517).
  - Addition of the `cnpg.io/podRole` label with a value of 'pooler' to every pooler deployment, differentiating them from instance pods (#3396).

Fixes:

- Reconciliation of metadata, annotations, and labels of `PodDisruptionBudget` resources (#3312 and #3434).
- Reconciliation of the metadata of the managed credential secrets (#3316).
- Resolution of a bug in the backup snapshot code where an error reading the body would be handled as an overall error, leaving the backup process indefinitely stuck (#3321).
- Implicit setting of online backup with the `cnpg backup` plugin command when either `immediate-checkpoint` or `wait-for-archive` options are requested (#3449).
- Disabling of wal_sender_timeout when joining through pg_basebackup (#3586)
- Reloading of secrets used by external clusters (#3565)
- Connection pooler:
  - Ensuring the controller watches all secrets owned by a `Pooler` resource (#3428).
  - Reconciliation of `RoleBinding` for `Pooler` resources (#3391).
  - Reconciliation of `imagePullSecret` for `Pooler` resources (#3389).
  - Reconciliation of the service of a `Pooler` and addition of the required labels (#3349).
  - Extension of `Pooler` labels to the deployment as well, not just the pods (#3350).

Changes:

- Default operand image set to PostgreSQL 16.1 (#3270).

Version 1.20.4

Release date: Nov 3, 2023

Enhancements:

- Enhance the `status` command of the `cnpg` plugin for `kubectl` with progress information on active streaming base backups (#3101)
- Allow the configuration of `max_prepared_statements` with the pgBouncer `Pooler` resource (#3174)

Fixes:

- Suspend WAL archiving during a switchover and resume it when it is completed (#3227)
- Ensure that the instance manager always uses `synchronous_commit = local` when managing the PostgreSQL cluster (#3143)
- Custom certificates for streaming replication user through `.spec.certificates.replicationTLSSecret` are now working (#3209)
- Set the `cnpg.io/cluster` label to the `Pooler` pods (#3153)

Changes:

- Stop using the `postgresql.auto.conf` file inside PGDATA to control Postgres replication settings, and replace it with a file named `override.conf` (#2812)

Technical enhancements:

- Use extended query protocol for PostgreSQL in the instance manager (#3152)

Version 1.20.3

Release date: Oct 11, 2023

Important Changes:

- Change the default value of `stopDelay` to 1800 seconds instead of 30 seconds (#2848)
- Introduce a new parameter, called `smartShutdownTimeout` , to control the window of time reserved for the smart shutdown of Postgres to complete; the general formula to compute the overall timeout to stop Postgres is `max(stopDelay - smartShutdownTimeout, 30)` (#2848)
- Change the default value of `startDelay` to 3600, instead of 30 seconds (#2847)
- Replace the livenessProbe initial delay with a more proper Kubernetes startup probe to deal with the start of a Postgres server (#2847)
- Change the default value of `switchoverDelay` to 3600 seconds instead of 40000000 seconds (#2846)
- Stop supporting the `postgresql` label - replaced by `cnpg.io/cluster` in 1.18 (#2744)

Security:

- Add a default `seccompProfile` to the operator deployment (#2926)

Enhancements:

- Introduce the `cnpg.io/coredumpFilter` annotation to control the content of a core dump generated in the unlikely event of a PostgreSQL crash, by default set to exclude shared memory segments from the dump (#2733)
- Allow to configure ephemeral-storage limits for the shared memory and temporary data ephemeral volumes (#2830)
- Validate resource limits and requests through the webhook (#2663)
- Ensure that PostgreSQL' s `shared_buffers` are coherent with the pods' allocated memory resources (#2840)
- Add `uri` and `jdbc-uri` fields in the credential secrets to facilitate developers when connecting their applications to the database (#2186)
- Add a new phase `Waiting for the instances to become active` for finer control of a cluster' s state waiting for the replicas to be ready (#2612)
- Improve detection of Pod rollout conditions through the `podSpec` annotation (#2243)
- Add primary timestamp and uptime to the kubectl plugin' s `status` command (#2953)

Fixes:

- Ensure that the primary instance is always recreated first by prioritizing ready PVCs with a primary role (#2544)
- Honor the `cnpg.io/skipEmptyWalArchiveCheck` annotation during recovery to bypass the check for an empty WAL archive (#2731)
- Prevent a cluster from being stuck when the PostgreSQL server is down but the pod is up on the primary (#2966)
- Avoid treating the designated primary in a replica cluster as a regular HA replica when replication slots are enabled (#2960)
- Reconcile services every time the selectors change or when labels/annotations need to be changed (#2918)
- Defaults to `app` both the owner and database during recovery bootstrap (#2957)
- Avoid write-read concurrency on cached cluster (#2884)
- Remove empty items, make them unique and sort in the `ResourceName` sections of the generated roles (#2875)
- Ensure that the `ContinuousArchiving` condition is properly set to 'failed' in case of errors (#2625)
- Make the `Backup` resource reconciliation cycle more resilient on interruptions by stopping only if the backup is completed or failed (#2591)
- Reconcile PodMonitor `labels` and `annotations` (#2583)
- Fix backup failure due to missing RBAC `resourceNames` on the `Role` object (#2956)
- Observability:
  - Add TCP port label to default `pg_stat_replication` metric (#2961)
  - Fix the `pg_wal_stat` default metric for Prometheus (#2569)
  - Improve the `pg_replication` default metric for Prometheus (#2744 and #2750)
  - Use `alertInstanceLabelFilter` instead of `alertName` in the provided Grafana dashboard
  - Enforce `standard_conforming_strings` in metric collection (#2888)

Changes:

- Set the default operand image to PostgreSQL 16.0
- Fencing now uses PostgreSQL's fast shutdown instead of smart shutdown to halt an instance (#3051)
- Rename webhooks from kb.io to cnpg.io group (#2851)
- Replace the `cnpg snapshot` command with `cnpg backup -m volumeSnapshot` for the `kubectl` plugin
- Let the `cnpg hibernate` plugin command use the `ClusterManifestAnnotationName` and `PgControldataAnnotationName` annotations on PVCs (#2657)
- Add the `cnpg.io/instanceRole` label while deprecating the existing `role` label (#2915)

Technical enhancements:

- Replace `k8s-api-docgen` with `gen-crd-api-reference-docs` to automatically build the API reference documentation (#2606)

Version 1.20.2

Release date: July 27, 2023

Enhancements:

- New `logs` command in the kubectl plugin, to retrieve or follow the logs of all pods in a cluster (#2375)
- Add support for specifying priorityClassName in pods, helping Kubernetes make scheduling decisions (#2043)
- Add a metric and status field to monitor node usage by a CloudNativePG cluster (#2257)
- Various enhancements to the documentation:
  - Add troubleshooting instructions relating to hugepages (#1390)
  - Extend the FAQs page (#2344)

Technical enhancements:

- Add a check at the start of the restore process to ensure it can proceed; give improved error diagnostics if it cannot (#2419)
- Improve handling of non-expiring passwords in managed roles (#2334)

Fixes:

- Ensure the logic of setting the recovery target matches that of Postgres (#2460)
- Prevent taking over service accounts not owned by the cluster, by setting ownerMetadata only during service account creation (#2462)
- Ensure correct permissions of the PGDATA directory for initdb and restore (#2384)
- Prevent a possible crash of the instance manager during the configuration reload (#2393)
- Prevent the LastFailedArchiveTime alert from triggering if a new backup has been successful after the failed ones (#1751)
- Prevent services from targeting non-instance pods (#2336)

Security:

- Updated all project dependencies to the latest versions

Version 1.20.1

Release date: June 12, 2023

Enhancements:

- Add the `snapshot` command to the `cnpg` plugin to create a consistent cold backup of the cluster from a standby using the Kubernetes `VolumeSnapshot` standard resource (#1960)
- First implementation of recovery from a set of CSI VolumeSnapshot resources via the `.spec.bootstrap.recovery.volumeSnapshot` stanza (#1960)
- Add `pg_failover_slots` to managed extensions (#2057)
- Improved Grafana dashboard with updated instructions in the documentation and the quickstart guide (#1916)
- Introduce the `schemaOnly` option in the `import` stanza, to avoid exporting and importing data when you bootstrap a new Postgres Cluster from one or more existing databases (#2234)
- Add support for TopologySpreadConstraints to manage scheduling of instance pods (#2202)

- Add `PodMonitor` support to the `Pooler` for PgBouncer (#2034)
- Add option to override the default Kubernetes scheduler (#2013)
- Allow configuration of deployment strategy of a `Pooler` resource (#1983)
- Update default PostgreSQL version to 15.3 (#2022)
- Use PgBouncer 1.19 by default (#2018)

Technical enhancements:

- Updated k8s kind tested versions (#2054)
- Declarative roles should ignore passwords if not set, easing management of previously existing roles (#2029)
- Use separate transactions to reconcile role credentials. Before this patch, the operator would revert the synchronization of all roles if one failed (#2004)
- Ensure fencing is removed during cluster restore (#1987)
- Improve logging when deleting Pods (#2136)

Fixes:

- Fix unbound variable with k3s engine which could prevent setup on K3' s (#2157)
- Report the correct PG version in the metrics (#2126)
- Use the correct walStorage key in the documentation (#2140)
- Halt reconciliation when the operator cannot connect with the instances, and provide a clear diagnostic on such occasions. This will help clarify cases where network issues obstruct normal operation of CloudNativePG (#2145), (#2233), and (#2242)

Version 1.20.0

Release date: April 27, 2023

Features:

- **Declarative role management:** introduce the `managed.roles` stanza in the `Cluster` spec to provide full lifecycle management of database roles, by providing an abstraction to the related DDL commands in PostgreSQL, such as `CREATE ROLE` and `ALTER ROLE` (#1524, #1793 and #1815)
- **Declarative hibernation of a PostgreSQL cluster:** introduce a new annotation called `cnpg.io/hibernation` to declaratively hibernate a PostgreSQL cluster by deleting all pods and keeping the PVCs only; the feature also implements the inverse procedure (#1657)

Enhancements:

- Improve the `--logs` option of the `report` command of the `cnpg` plugin for `kubectl` to also include the previous logs where available (#1811)
- The `-any` service is now disabled by default (#1755)

Security:

- Enable customization of `SeccompProfile` through override via a local file (#1827)

Fixes:

- Apply the PostgreSQL configuration provided by the user during the `initdb` bootstrap phase, before the server is started the first time (#1858)

Release notes for CloudNativePG 1.19

History of user-visible changes in the 1.19 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

Version 1.19.6

Release date: Nov 3, 2023

> **Warning:** This is expected to be the last release in the 1.19.X series. Users are encouraged to update to a newer minor version soon.

Enhancements:

- Enhance the `status` command of the `cnpg` plugin for `kubectl` with progress information on active streaming base backups (#3101)
- Allow the configuration of `max_prepared_statements` with the pgBouncer `Pooler` resource (#3174)

Fixes:

- Suspend WAL archiving during a switchover and resume it when it is completed (#3227)
- Ensure that the instance manager always uses `synchronous_commit = local` when managing the PostgreSQL cluster (#3143)
- Custom certificates for streaming replication user through `.spec.certificates.replicationTLSSecret` are now working (#3209)
- Set the `cnpg.io/cluster` label to the `Pooler` pods (#3153)

Changes:

- Stop using the `postgresql.auto.conf` file inside PGDATA to control Postgres replication settings, and replace it with a file named `override.conf` (#2812)

Technical enhancements:

- Use extended query protocol for PostgreSQL in the instance manager (#3152)

Version 1.19.5

Release date: Oct 11, 2023

> **Warning:** Version 1.19 will reach its End-of-Life (EOL) on November 9, 2023. If you haven't done it yet, please start planning an upgrade as soon as possible.

Important Changes:

- Change the default value of `stopDelay` to 1800 seconds instead of 30 seconds (#2848)

- Introduce a new parameter, called `smartShutdownTimeout` , to control the window of time reserved for the smart shutdown of Postgres to complete; the general formula to compute the overall timeout to stop Postgres is `max(stopDelay - smartShutdownTimeout, 30)` (#2848)
- Change the default value of `startDelay` to 3600, instead of 30 seconds (#2847)
- Replace the livenessProbe initial delay with a more proper Kubernetes startup probe to deal with the start of a Postgres server (#2847)
- Change the default value of `switchoverDelay` to 3600 seconds instead of 40000000 seconds (#2846)
- Stop supporting the `postgresql` label - replaced by `cnpg.io/cluster` in 1.18 (#2744)

Security:

- Add a default `seccompProfile` to the operator deployment (#2926)

Enhancements:

- Introduce the `cnpg.io/coredumpFilter` annotation to control the content of a core dump generated in the unlikely event of a PostgreSQL crash, by default set to exclude shared memory segments from the dump (#2733)
- Allow to configure ephemeral-storage limits for the shared memory and temporary data ephemeral volumes (#2830)
- Validate resource limits and requests through the webhook (#2663)
- Ensure that PostgreSQL' s `shared_buffers` are coherent with the pods' allocated memory resources (#2840)
- Add `uri` and `jdbc-uri` fields in the credential secrets to facilitate developers when connecting their applications to the database (#2186)
- Add a new phase `Waiting for the instances to become active` for finer control of a cluster' s state waiting for the replicas to be ready (#2612)
- Improve detection of Pod rollout conditions through the `podSpec` annotation (#2243)
- Add primary timestamp and uptime to the kubectl plugin' s `status` command (#2953)

Fixes:

- Ensure that the primary instance is always recreated first by prioritizing ready PVCs with a primary role (#2544)
- Honor the `cnpg.io/skipEmptyWalArchiveCheck` annotation during recovery to bypass the check for an empty WAL archive (#2731)
- Prevent a cluster from being stuck when the PostgreSQL server is down but the pod is up on the primary (#2966)
- Avoid treating the designated primary in a replica cluster as a regular HA replica when replication slots are enabled (#2960)
- Reconcile services every time the selectors change or when labels/annotations need to be changed (#2918)
- Defaults to `app` both the owner and database during recovery bootstrap (#2957)
- Avoid write-read concurrency on cached cluster (#2884)
- Remove empty items, make them unique and sort in the `ResourceName` sections of the generated roles (#2875)
- Ensure that the `ContinuousArchiving` condition is properly set to 'failed' in case of errors (#2625)
- Make the `Backup` resource reconciliation cycle more resilient on interruptions by stopping only if the backup is completed or failed (#2591)
- Reconcile PodMonitor `labels` and `annotations` (#2583)
- Fix backup failure due to missing RBAC `resourceNames` on the `Role` object (#2956)

- Observability:
  - Add TCP port label to default `pg_stat_replication` metric (#2961)
  - Fix the `pg_wal_stat` default metric for Prometheus (#2569)
  - Improve the `pg_replication` default metric for Prometheus (#2744 and #2750)
  - Use `alertInstanceLabelFilter` instead of `alertName` in the provided Grafana dashboard
  - Enforce `standard_conforming_strings` in metric collection (#2888)

Changes:

- Set the default operand image to PostgreSQL 16.0
- Fencing now uses PostgreSQL' s fast shutdown instead of smart shutdown to halt an instance (#3051)
- Rename webhooks from kb.io to cnpg.io group (#2851)
- Let the `cnpg hibernate` plugin command use the `ClusterManifestAnnotationName` and `PgControldataAnnotationName` annotations on PVCs (#2657)
- Add the `cnpg.io/instanceRole` label while deprecating the existing `role` label (#2915)

Technical enhancements:

- Replace `k8s-api-docgen` with `gen-crd-api-reference-docs` to automatically build the API reference documentation (#2606)

Version 1.19.4

Release date: July 27, 2023

Enhancements:

- New `logs` command in the kubectl plugin, to retrieve or follow the logs of all pods in a cluster (#2375)
- Add support for specifying priorityClassName in pods, helping Kubernetes make scheduling decisions (#2043)
- Add a metric and status field to monitor node usage by a CloudNativePG cluster (#2257)
- Various enhancements to the documentation:
  - Add troubleshooting instructions relating to hugepages (#1390)
  - Extend the FAQs page (#2344)

Technical enhancements:

- Add a check at the start of the restore process to ensure it can proceed; give improved error diagnostics if it cannot (#2419)

Fixes:

- Ensure the logic of setting the recovery target matches that of Postgres (#2460)
- Prevent taking over service accounts not owned by the cluster, by setting ownerMetadata only during service account creation (#2462)
- Ensure correct permissions of the PGDATA directory for initdb and restore (#2384)
- Prevent a possible crash of the instance manager during the configuration reload (#2393)
- Prevent the LastFailedArchiveTime alert from triggering if a new backup has been successful after the failed ones (#1751)

- Prevent services from targeting non-instance pods (#2336)

Security:

- Updated all project dependencies to the latest versions

Version 1.19.3

Release date: June 12, 2023

Enhancements:

- Add the `snapshot` command to the `cnpg` plugin to create a consistent cold backup of the cluster from a standby using the Kubernetes `VolumeSnapshot` standard resource (#1960)
- First implementation of recovery from a set of CSI VolumeSnapshot resources via the `.spec.bootstrap.recovery.volumeSnapshot` stanza (#1960)
- Add `pg_failover_slots` to managed extensions (#2057)
- Improved Grafana dashboard with updated instructions in the documentation and the quickstart guide (#1916)
- Introduce the `schemaOnly` option in the `import` stanza, to avoid exporting and importing data when you bootstrap a new Postgres Cluster from one or more existing databases (#2234)
- Add support for TopologySpreadConstraints to manage scheduling of instance pods (#2202)
- Add `PodMonitor` support to the `Pooler` for PgBouncer (#2034)
- Add option to override the default Kubernetes scheduler (#2013)
- Allow configuration of deployment strategy of a `Pooler` resource (#1983)
- Update default PostgreSQL version to 15.3 (#2022)
- Use PgBouncer 1.19 by default (#2018)

Technical enhancements:

- Updated k8s kind tested versions (#2054)
- Use separate transactions to reconcile role credentials. Before this patch, the operator would revert the synchronization of all roles if one failed (#2004)
- Ensure fencing is removed during cluster restore (#1987)
- Improve logging when deleting Pods (#2136)

Fixes:

- Fix unbound variable with k3d engine which could prevent setup on k3d (#2157)
- Report the correct PG version in the metrics (#2126)
- Use the correct walStorage key in the documentation (#2140)
- Halt reconciliation when the operator cannot connect with the instances, and provide a clear diagnostic on such occasions. This will help clarify cases where network issues obstruct normal operation of CloudNativePG (#2145), (#2233), and (#2242)

Version 1.19.2

Release date: April 27, 2023

Enhancements:

- Improve the `--logs` option of the `report` command of the `cnpg` plugin for `kubectl` to also include the previous logs where available (#1811)
- The `-any` service is now disabled by default (#1755)

Security:

- Enable customization of `SeccompProfile` through override via a local file (#1827)

Fixes:

- Apply the PostgreSQL configuration provided by the user during the `initdb` bootstrap phase, before the server is started the first time (#1858)

Version 1.19.1

Release date: March 20, 2023

Enhancements:

- Allow overriding the default backup target policy (#1602): previously, all backups and scheduled backups would use the cluster-level target policy
- Extend the `debug` cluster' s log level to the `initdb` job (#1503)
- Support IPv6 and custom `pg_hba` for the PgBouncer pooler (#1395)
- Enhance observability of backups with two new metrics and additional information in the status (#1428)
- Document API calls from the instance manager (#1641)
- Clarify deployment name via Helm (#1505)
- Add the `psql` command to the `cnpg` plugin for `kubectl` (#1668) allowing the user to start a `psql` session with a pod (the primary by default)

Technical enhancements:

- Adopt Renovate for dependency tracking/updating (#1367, #1473)
- Inject binaries for all supported architectures in the operator image (#1513)
- Use the backup name to match resources in the backup object store (#1650) Leverages the `--name` option introduced with Barman 3.3 to make the association between backups and the object store more robust.

Fixes:

- Prevent panic with error handling in the probes (#1716)
- Ensure that the HTTP package and controller runtime logs are in JSON format (#1442)
- Adds WAL storage to a cluster in a single instance Cluster (#1570)
- Various improvements to make backup code more robust (#1536, #1564, #1588, #1466, #1647)
- Properly show WAL archiving information with `status` command of the `cnpg` plugin (#1666)
- Ensure `nodeAffinity` is applied even if `AdditionalPodAffinity` and `AdditionalPodAntiAffinity` are not set (#1663)

- Introduce failover delay during OnlineUpgrading phase (#1728) Previously, the online upgrade process could trigger failover logic unnecessarily.

Version 1.19.0

Release date: Feb 14, 2023

Important announcements:

- PostgreSQL version 10 is no longer supported as it has reached its EOL. Versions 11 and newer are supported. Please plan your migration to PostgreSQL 15 as soon as possible. Refer to Importing Postgres databases

for more information on PostgreSQL major offline upgrades.

Features:

- Backup from a standby: introduce the `.spec.backup.target` option accepting that when set to `prefer-standby` will run take the physical base backup from the most aligned replica (#1162)
- Delayed failover: introduce the `failoverDelay` parameter to delay the failover process once the primary has been detected unhealthy (#1366)

Enhancements:

- Introduce support for Kubernetes' projected volumes (#1269)
- Introduce support custom environment variables for finer control of the PostgreSQL server process (#1275)
- Introduce the `backup` command in the `cnpg` plugin for `kubectl` to issue a new base backup of the cluster (#1348)
- Improve support for the separate WAL volume feature by enabling users to move WAL files to a dedicated volume on an existing Postgres cluster (#1066)
- Enhance WAL observability with additional metrics for the Prometheus exporter, including values equivalent to the `min_wal_size` , `max_wal_size` , `keep_wal_size` , `wal_keep_segments` , as well as the maximum number of WALs that can be stored in the dedicated volume (#1382)
- Add a database comment on the `streaming_replica` user (#1349)
- Document the firewall issues with webhooks on GKE (#1364)
- Add note about postgresql.conf in `recovery` (#1211)
- Add instructions on installing plugin using packages (#1357)
- Specify Postgres versions supported by each minor release (#1355)
- Clarify the meaning of PVC group in CloudNativePG (#1344)
- Add an example of the DigitalOcean S3-compatible Spaces (#1289)
- Update default PostgreSQL version for new cluster definitions to 15.2 (#1430)
- Cover the Kubernetes layer in greater detail in the Architecture documentation (#1432)

Technical enhancements:

- Added daily end-to-end smoke test for release branches (#1235)

Fixes:

- Skip executing a `CHECKPOINT` as the `streaming_replica` user (#1408)
- Make `waitForWalArchiveWorking` resilient to connection errors (#1399)
- Ensure that the PVC roles are always consistent (#1380)

- Permit `walStorage` resize when using `pvcTemplate` (#1315)
- Ensure `ExecCommand` obeys timeout (#1242)
- Avoid `PodMonitor` reconcile if Prometheus is not installed (#1238)
- Avoid looking for `PodMonitor` when not needed (#1213)

Release notes for CloudNativePG 1.18

History of user-visible changes in the 1.18 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

Version 1.18.5

Release date: June 12, 2023

> **Warning:** This is expected to be the last release in the 1.18.X series. Users are encouraged to update to a newer minor version soon.

Enhancements:

- Add the `snapshot` command to the `cnpg` plugin to create a consistent cold backup of the cluster from a standby using the Kubernetes `VolumeSnapshot` standard resource (#1960)
- First implementation of recovery from a set of CSI VolumeSnapshot resources via the `.spec.bootstrap.recovery.volumeSnapshot` stanza (#1960)
- Add `pg_failover_slots` to managed extensions (#2057)
- Improved Grafana dashboard with updated instructions in the documentation and the quickstart guide (#1916)
- Introduce the `schemaOnly` option in the `import` stanza, to avoid exporting and importing data when you bootstrap a new Postgres Cluster from one or more existing databases (#2234)
- Add support for TopologySpreadConstraints to manage scheduling of instance pods (#2202)
- Add `PodMonitor` support to the `Pooler` for PgBouncer (#2034)
- Add option to override the default Kubernetes scheduler (#2013)
- Allow configuration of deployment strategy of a `Pooler` resource (#1983)
- Update default PostgreSQL version to 15.3 (#2022)
- Use PgBouncer 1.19 by default (#2018)

Technical enhancements:

- Updated k8s kind tested versions (#2054)
- Use separate transactions to reconcile role credentials. Before this patch, the operator would revert the synchronization of all roles if one failed (#2004)
- Ensure fencing is removed during cluster restore (#1987)
- Improve logging when deleting Pods (#2136)

Fixes:

- Fix unbound variable with k3s engine which could prevent setup on K3' s (#2157)
- Report the correct PG version in the metrics (#2126)
- Use the correct walStorage key in the documentation (#2140)
- Halt reconciliation when the operator cannot connect with the instances, and provide a clear diagnostic on such occasions. This will help clarify cases where network issues obstruct normal operation of CloudNativePG (#2145), (#2233), and (#2242)

Version 1.18.4

Release date: April 27, 2023

Important: CloudNativePG is dropping support for PostgreSQL 10, as PostgreSQL 10 reached End-of-Life (EOL) in November 2022. Versions 11 and newer are supported. Please plan your migration to PostgreSQL 15 as soon as possible. Refer to Importing Postgres databases

---

for more information on PostgreSQL major offline upgrades.

Enhancements:

- Improve the `--logs` option of the `report` command of the `cnpg` plugin for `kubectl` to also include the previous logs where available (#1811)
- The `-any` service is now disabled by default (#1755)

Security:

- Enable customization of `SeccompProfile` through override via a local file (#1827)

Fixes:

- Apply the PostgreSQL configuration provided by the user during the `initdb` bootstrap phase, before the server is started the first time (#1858)

Version 1.18.3

Release date: March 20, 2023

Enhancements:

- Extend the `debug` cluster' s log level to the `initdb` job (#1503)
- Support IPv6 and custom `pg_hba` for the PgBouncer pooler (#1395)
- Enhance observability of backups with two new metrics and additional information in the status (#1428)
- Document API calls from the instance manager (#1641)
- Clarify deployment name via Helm (#1505)
- Add the `psql` command to the `cnpg` plugin for `kubectl` (#1668) allowing the user to start a `psql` session with a pod (the primary by default)

Technical enhancements:

- Adopt Renovate for dependency tracking/updating (#1367, #1473)

- Inject binaries for all supported architectures in the operator image (#1513)
- Use the backup name to match resources in the backup object store (#1650) Leverages the `--name` option introduced with Barman 3.3 to make the association between backups and the object store more robust.

Fixes:

- Prevent panic with error handling in the probes (#1716)
- Ensure that the HTTP package and controller runtime logs are in JSON format (#1442)
- Adds WAL storage to a cluster in a single instance Cluster (#1570)
- Various improvements to make backup code more robust (#1536, #1564, #1588, #1466, #1647)
- Properly show WAL archiving information with `status` command of the `cnpg` plugin (#1666)
- Ensure `nodeAffinity` is applied even if `AdditionalPodAffinity` and `AdditionalPodAntiAffinity` are not set (#1663)

Version 1.18.2

Release date: Feb 14, 2023

Enhancements:

- Introduce support for Kubernetes' projected volumes (#1269)
- Introduce support custom environment variables for finer control of the PostgreSQL server process (#1275)
- Introduce the `backup` command in the `cnpg` plugin for `kubectl` to issue a new base backup of the cluster (#1348)
- Improve support for the separate WAL volume feature by enabling users to move WAL files to a dedicated volume on an existing Postgres cluster (#1066)
- Enhance WAL observability with additional metrics for the Prometheus exporter, including values equivalent to the `min_wal_size` , `max_wal_size` , `keep_wal_size` , `wal_keep_segments` , as well as the maximum number of WALs that can be stored in the dedicated volume (#1382)
- Add a database comment on the `streaming_replica` user (#1349)
- Document the firewall issues with webhooks on GKE (#1364)
- Add note about postgresql.conf in `recovery` (#1211)
- Add instructions on installing plugin using packages (#1357)
- Specify Postgres versions supported by each minor release (#1355)
- Clarify the meaning of PVC group in CloudNativePG (#1344)
- Add an example of the DigitalOcean S3-compatible Spaces (#1289)

Technical enhancements:

- Added daily end-to-end smoke test for release branches (#1235)

Fixes:

- Skip executing a `CHECKPOINT` as the `streaming_replica` user (#1408)
- Make `waitForWalArchiveWorking` resilient to connection errors (#1399)
- Ensure that the PVC roles are always consistent (#1380)
- Permit `walStorage` resize when using `pvcTemplate` (#1315)

- Ensure `ExecCommand` obeys timeout (#1242)
- Avoid `PodMonitor` reconcile if Prometheus is not installed (#1238)
- Avoid looking for `PodMonitor` when not needed (#1213)

Version 1.18.1

Release date: Dec 21, 2022

Important announcements:

- Alert on the impending deprecation of `postgresql` as a label to identify the CNPG cluster. In the remote case you have used this label, please start using the `cnpg.io/cluster` label instead (#1130)
- Recognizing Armando Ruocco (@armru) as a new CloudNativePG maintainer for his consistent and impactful contributions (#1167)
- Remove ARMv7 support (#1092)
- FINAL patch release for 1.16: 1.16.5. Release 1.16 reaches end of life.

Enhancements:

- Customize labels and annotations for the service account: add a service account template that can be used, for example, to make authentication easier via identity management on GKE or EKS via IRSA (#1105)
- Add `nodeAffinity` support (#1182) - allows for richer scheduling options
- Improve compatibility with Istio: add support for Istio's quit endpoint so that jobs with Istio sidecars do not run indefinitely (#967)
- Allow fields remapping in JSON logs: helpful for use cases where the `level` and `ts` fields might interfere with the existing logging (#843)
- Add `fio` command to the kubectl-cnpg plugin (#1097)
- Add rpm/deb package for kubectl-cnpg plugin (#1008)
- Update default PostgreSQL version for new cluster definitions to 15.1 (#908)
- Documentation
- Remove references to CNPG sandbox (#1120) - the CNPG sandbox has been deprecated, in favor of instructions on monitoring in the Quickstart documentation
- Link to the "Release updates" discussion (#1148) - the release updates discussion will become the default channel for release announcements and discussions
- Document emeritus status for maintainers in GOVERNANCE.md (#1033) - explains how maintainers should proceed if they are not ready to continue contributing
- Improve instructions on creating pull requests (#1132)
- Troubleshooting emergency backup instructions (#1184)
- Cover the Kubernetes layer in greater detail in the Architecture documentation (#1432)

Fixes:

- Ensure PGDATA permissions on bootstrap are properly set to 750 (#1164)
- Ensure the PVC containing WALs is deleted when scaling down the cluster (#1135)
- Fix missing ApiVersion and Kind in the pgbench manifest when using `--dry-run` (#1088)

- Ensure that we create secrets and services only when not found (#1145)
- Respect configured pg-wal when restoring (#1216)
- Filter out replicas from nodeToClusters map (#1194)

Technical enhancements:

- Use `ciclops` for test summary (#1064): rely on the ciclops GitHub action to provide summaries of the E2E suite, inheriting improvements from that project
- Add backport pull request workflow (#965) - automatically backport patches to release branches if they are so annotated
- Make the operator log level configurable in e2e test suite (#1094)
- Enable test execution based on labels (#951)
- Update Go version from 1.18 to 1.19 (#1166)

Version 1.18.0

Release date: Nov 10, 2022

Features:

- **Cluster-managed physical replication slots for High Availability**: automatically manages physical replication slots for each hot standby replica in the High Availability cluster, both in the primary and the standby (#740)
- **Postgres cluster hibernation**: introduces cluster hibernation via the plugin, with a new subcommand `kubectl cnpg hibernate on/off/status <cluster-name>`. Hibernation destroys all the resources generated by the cluster, except the PVCs that belong to the PostgreSQL primary instance (#782)

Security:

- Add `SeccomProfile` to Pods and Containers (#888)

Enhancements:

- Allow omitting the storage size in the cluster spec if there is a size request in the `pvcTemplate` (#914)
- `status` command for the `cnpg` plugin:
- Add replication slots information (#873)
- Clarify display for fenced clusters (#886)
- Improve display for replica clusters (#871)
- Documentation:
- Improve monitoring page, providing instructions on how to evaluate the observability capabilities of CloudNativePG on a local system using Prometheus and Grafana (#968)
- Add page on design reasons for custom controller (#918)
- Updates to the End-to-End Test Suite page (#945)
- New subcommands in the `cnpg` plugin:
  - `pgbench` generates a job definition executing `pgbench` against a cluster (#958)
  - `install` generates an installation manifest for the operator (#944)
- Set PostgreSQL 15.0 as the new default version (#821)

Fixes:

- Import a database with `plpgsql` functions (#974)
- Properly find the closest backup when doing Point-in-time recovery (#949)
- Clarify that the `ScheduledBackup` format does not follow Kubernetes `CronJob` format (#883)
- Bases the failover logic on the Postgres information from the instance manager, rather than Kubernetes pod readiness, which could be stale (#890)
- Ensure we have a WAL to archive for every newly created cluster. The lack could prevent backups from working (#897)
- Correct YAML key names for `barmanObjectStore` in documentation (#877)
- Fix `krew` release (#866)

Release notes for CloudNativePG 1.17

History of user-visible changes in the 1.17 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

Version 1.17.5

Release date: March 20, 2023

> **Warning:** This is expected to be the last release in the 1.17.X series. Users are encouraged to update to a newer minor version soon.

Enhancements:

- Extend the `debug` cluster' s log level to the `initdb` job (#1503)
- Support IPv6 and custom `pg_hba` for the PgBouncer pooler (#1395)
- Document API calls from the instance manager (#1641)
- Clarify deployment name via Helm (#1505)
- Add the `psql` command to the `cnpg` plugin for `kubectl` (#1668) allowing the user to start a `psql` session with a pod (the primary by default)

Technical enhancements:

- Adopt Renovate for dependency tracking/updating (#1367, #1473)

Fixes:

- Prevent panic with error handling in the probes (#1716)
- Properly show WAL archiving information with `status` command of the `cnpg` plugin (#1666)

Version 1.17.4

Release date: Feb 14, 2023

Features:

- Support for Kubernetes' projected volumes (#1269)
- Support custom environment variables for finer control of the PostgreSQL server process (#1275)

Enhancements:

- Introduce the `backup` command in the `cnpg` plugin for `kubectl` to issue a new base backup of the cluster (#1348)
- Add a database comment on the `streaming_replica` user (#1349)
- Document the firewall issues with webhooks on GKE (#1364)
- Add note about postgresql.conf in `recovery` (#1211)
- Add instructions on installing plugin using packages (#1357)
- Specify Postgres versions supported by each minor release (#1355)
- Clarify the meaning of PVC group in CloudNativePG (#1344)
- Add an example of the DigitalOcean S3-compatible Spaces (#1289)

Technical enhancements:

- Added daily end-to-end smoke test for release branches (#1235)

Fixes:

- Ensure that the PVC roles are always consistent (#1380)
- Permit `walStorage` resize when using `pvcTemplate` (#1315)
- Avoid `PodMonitor` reconcile if Prometheus is not installed (#1238)
- Avoid looking for `PodMonitor` when not needed (#1213)

Version 1.17.3

Release date: Dec 21, 2022

Important announcements:

- Recognizing Armando Ruocco (@armru) as a new CloudNativePG maintainer for his consistent and impactful contributions (#1167)
- Remove ARMv7 support (#1092)
- FINAL patch release for 1.16: 1.16.5. Release 1.16 reaches end of life.

Enhancements:

- Improve compatibility with Istio: add support for Istio's quit endpoint so that jobs with Istio sidecars do not run indefinitely (#967)
- Add rpm/deb package for kubectl-cnpg plugin (#1008)
- Update default PostgreSQL version for new cluster definitions to 15.1 (#908)
- Documentation

- Remove references to CNPG sandbox (#1120) - the CNPG sandbox has been deprecated, in favor of instructions on monitoring in the Quickstart documentation
- Link to the "Release updates" discussion (#1148) - the release updates discussion will become the default channel for release announcements and discussions
- Document emeritus status for maintainers in GOVERNANCE.md (#1033) - explains how maintainers should proceed if they are not ready to continue contributing
- Improve instructions on creating pull requests (#1132)
- Troubleshooting emergency backup instructions (#1184)
- Cover the Kubernetes layer in greater detail in the Architecture documentation (#1432)

Fixes:

- Ensure PGDATA permissions on bootstrap are properly set to 750 (#1164)
- Ensure the PVC containing WALs is deleted when scaling down the cluster (#1135)
- Ensure that we create secrets and services only when not found (#1145)
- Respect configured pg-wal when restoring (#1216)
- Filter out replicas from nodeToClusters map (#1194)

Technical enhancements:

- Use `ciclops` for test summary (#1064): rely on the ciclops GitHub action to provide summaries of the E2E suite, inheriting improvements from that project
- Add backport pull request workflow (#965) - automatically backport patches to release branches if they are so annotated
- Make the operator log level configurable in e2e test suite (#1094)
- Enable test execution based on labels (#951)
- Update Go version from 1.18 to 1.19 (#1166)

Version 1.17.2

Release date: Nov 10, 2022

Security:

- Add `SeccomProfile` to Pods and Containers (#888)

Enhancements:

- `status` command for the `cnpg` plugin:
- Clarify display for fenced clusters (#886)
- Improve display for replica clusters (#871)
- Documentation:
- Improve monitoring page, providing instructions on how to evaluate the observability capabilities of CloudNativePG on a local system using Prometheus and Grafana (#968)
- Add page on design reasons for custom controller (#918)
- Set PostgreSQL 15.0 as the new default version (#821)

Fixes:

- Import a database with `plpgsql` functions (#974)
- Properly find the closest backup when doing Point-in-time recovery (#949)
- Clarify that the `ScheduledBackup` format does not follow Kubernetes `CronJob` format (#883)
- Bases the failover logic on the Postgres information from the instance manager, rather than Kubernetes pod readiness, which could be stale (#890)
- Ensure we have a WAL to archive for every newly created cluster. The lack could prevent backups from working (#897)
- Correct YAML key names for `barmanObjectStore` in documentation (#877)
- Fix `krew` release (#866)

Version 1.17.1

Release date: Oct 6, 2022

Enhancements:

- Introduce `leaseDuration` and `renewDeadline` parameters in the controller manager to enhance configuration of the leader election in operator deployments (#759)
- Improve the mechanism that checks that the backup object store is empty before archiving a WAL file for the first time: a new file called `.check-empty-wal-archive` is placed in the `PGDATA` immediately after the cluster is bootstrapped and it is then removed after the first WAL file is successfully archived

Security:

- Explicitly set permissions of the instance manager binary that is copied in the `distroless/static:nonroot` container image, by using the `nonroot:nonroot` user (#754)

Fixes:

- Drop any active connection on a standby after it is promoted to primary (#737)
- Honor `MAPPEDMETRIC` and `DURATION` metric types conversion in the native Prometheus exporter (#765)
- Ensure that timestamps that are specified with microsecond precision using the PostgreSQL format are correctly parsed (#741)

Version 1.17.0

Release date: Sep 6, 2022 (minor release)

Features:

- **Separate volume for WAL files:** Support for separating Write Ahead Log (WAL) and database data files onto different disks, potentially leading to better performance on high write systems by easing I/O load on the data directory. This option is controlled with the introduction of the optional `walStorage` section to separate WAL files (`pg_wal`) in a dedicated volume, separate from the `PGDATA` defined in the main and mandatory `storage` section (#513). Current limitations: `walStorage` can only be set at cluster creation and cannot be added or removed when the cluster is up and running.

Enhancements:

- Enable configuration of low-level network TCP settings in the PgBouncer connection pooler implementation (#584)
- Make sure that the `cnpg.io/instanceName` and the `cnpg.io/podRole` labels are always present on pods and PVCs (#632 and #680)
- Propagate the `role` label of an instance to the underlying PVC (#634)
- Introduce the `kubectl cnpg destroy` command to help remove an instance and all the associated PVCs (#643)

Fixes:

- Use `shared_preload_libraries` when bootstrapping the new cluster' s primary (#642)
- Prevent multiple in-place upgrade processes of the operator from running simultaneously by atomically checking whether another one is in progress (#655)
- Avoid using a hardcoded file name to store the newly uploaded instance manager, preventing a possible race condition during online upgrades of the operator (#660)
- Prevent a panic from happening when invoking `GetAllAccessibleDatabases` (#641)

Release notes for CloudNativePG 1.16

History of user-visible changes in the 1.16 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

Version 1.16.5

Release date: Dec 21, 2022

> **Warning:** This is expected to be the last release in the 1.16.X series. Users are encouraged to update to a newer minor version soon.

Important announcements:

- Recognizing Armando Ruocco (@armru) as a new CloudNativePG maintainer for his consistent and impactful contributions (#1167)
- Remove ARMv7 support (#1092)
- FINAL patch release for 1.16: 1.16.5. Release 1.16 reaches end of life.

Enhancements:

- Add rpm/deb package for kubectl-cnpg plugin (#1008)
- Update default PostgreSQL version for new cluster definitions to 15.1 (#908)
- Documentation
- Remove references to CNPG sandbox (#1120) - the CNPG sandbox has been deprecated, in favor of instructions on monitoring in the Quickstart documentation
- Link to the "Release updates" discussion (#1148) - the release updates discussion will become the default channel for release announcements and discussions

- Document emeritus status for maintainers in GOVERNANCE.md (#1033) - explains how maintainers should proceed if they are not ready to continue contributing
- Improve instructions on creating pull requests (#1132)
- Troubleshooting emergency backup instructions (#1184)

Fixes:

- Ensure PGDATA permissions on bootstrap are properly set to 750 (#1164)
- Ensure that we create secrets and services only when not found (#1145)
- Respect configured pg-wal when restoring (#1216)
- Filter out replicas from nodeToClusters map (#1194)

Technical enhancements:

- Use `ciclops` for test summary (#1064): rely on the ciclops GitHub action to provide summaries of the E2E suite, inheriting improvements from that project
- Add backport pull request workflow (#965) - automatically backport patches to release branches if they are so annotated
- Make the operator log level configurable in e2e test suite (#1094)
- Enable test execution based on labels (#951)
- Update Go version from 1.18 to 1.19 (#1166)

Version 1.16.4

Release date: Nov 10, 2022

Security:

- Add `SeccomProfile` to Pods and Containers (#888)

Enhancements:

- `status` command for the `cnpg` plugin:
- Clarify display for fenced clusters (#886)
- Improve display for replica clusters (#871)
- Documentation:
- Improve monitoring page, providing instructions on how to evaluate the observability capabilities of CloudNativePG on a local system using Prometheus and Grafana (#968)
- Add page on design reasons for custom controller (#918)

Fixes:

- Import a database with `plpgsql` functions (#974)
- Properly find the closest backup when doing Point-in-time recovery (#949)
- Clarify that the `ScheduledBackup` format does not follow Kubernetes `CronJob` format (#883)
- Bases the failover logic on the Postgres information from the instance manager, rather than Kubernetes pod readiness, which could be stale (#890)

- Ensure we have a WAL to archive for every newly created cluster. The lack could prevent backups from working (#897)
- Correct YAML key names for `barmanObjectStore` in documentation (#877)
- Fix `krew` release (#866)

Version 1.16.3

Release date: Oct 6, 2022

Enhancements:

- Introduce `leaseDuration` and `renewDeadline` parameters in the controller manager to enhance configuration of the leader election in operator deployments (#759)
- Improve the mechanism that checks that the backup object store is empty before archiving a WAL file for the first time: a new file called `.check-empty-wal-archive` is placed in the `PGDATA` immediately after the cluster is bootstrapped and it is then removed after the first WAL file is successfully archived

Security:

- Explicitly set permissions of the instance manager binary that is copied in the `distroless/static:nonroot` container image, by using the `nonroot:nonroot` user (#754)

Fixes:

- Drop any active connection on a standby after it is promoted to primary (#737)
- Honor MAPPEDMETRIC and DURATION metric types conversion in the native Prometheus exporter (#765)

Version 1.16.2

Release date: Sep 6, 2022

Enhancements:

- Enable configuration of low-level network TCP settings in the PgBouncer connection pooler implementation (#584)
- Make sure that the `cnpg.io/instanceName` and the `cnpg.io/podRole` labels are always present on pods and PVCs (#632 and #680)
- Propagate the `role` label of an instance to the underlying PVC (#634)

Fixes:

- Use `shared_preload_libraries` when bootstrapping the new cluster' s primary (#642)
- Prevent multiple in-place upgrade processes of the operator from running simultaneously by atomically checking whether another one is in progress (#655)
- Avoid using a hardcoded file name to store the newly uploaded instance manager, preventing a possible race condition during online upgrades of the operator (#660)
- Prevent a panic from happening when invoking `GetAllAccessibleDatabases` (#641)

Version 1.16.1

Release date: Aug 12, 2022

Enhancements:

- Enable the configuration of the `huge_pages` option for PostgreSQL (#456)
- Enhance log during promotion and demotion, after a failover or a switchover, by printing the time elapsed between the request of promotion and the actual availability for writes (#371)
- Introduce the PostgreSQL cluster' s timeline in the cluster status (#462)
- Add the `instanceName` and `clusterName` labels on jobs, pods, and PVCs to improve interaction with these resources (#534)
- Add instructions on how to create PostGIS clusters (#570)

Security:

- Explicitly assign `securityContext` to the `Pooler` deployment (#485)
- Add read timeout values to the internal web servers to prevent Slowloris DDoS (#437)

Fixes:

- Use the correct delays for restarts (`stopDelay`) and for switchover (`switchoverDelay`), as they were erroneously swapped before. This is an important fix, as it might block indefinitely restarts if `switchoverDelay` is not set and uses the default value of 40000000 seconds (#531)
- Prevent the metrics collector from causing panic when the query returns an error (#396)
- Removing an unsafe debug message that was referencing an unchecked pointer, leading in some cases to segmentation faults regardless of the log level (#491)
- Prevent panic when fencing in case the cluster had no annotation (#512)
- Avoid updating the CRD if a TLS certificate is not changed (#501)
- Handle conflicts while injecting a certificate in the CRD (#547)
- Database import:
  - Use the `postgres` user while running `pg_restore` in database import (#411)
  - Document the requirement to explicitly set `sslmode` in the monolith import case to control SSL connections with the origin external server (#572)
  - Fix bug that prevented import from working when `dbname` was specified in `connectionParameters` (#569)
- Backup and recovery: - Correctly pass object store credentials in Google Cloud (#454)

Minor changes:

- Set the default operand image to PostgreSQL 15.0

Version 1.16.0

Release date: Jul 7, 2022 (minor release)

Features:

- **Offline data import and major upgrades for PostgreSQL:** introduce the `bootstrap.initdb.import` section to provide a way to import objects via the network from an existing PostgreSQL instance (even outside Kubernetes) inside a brand new CloudNativePG cluster using the PostgreSQL logical backup concept (`pg_dump` /`pg_restore`). The same method can be used to perform major PostgreSQL upgrades on a new cluster. The feature introduces two types of import: `microservice` (import one database only in the new cluster) and `monolith` (import the selected databases and roles from the existing instance).
- Anti-affinity rules for synchronous replication based on labels: make sure that synchronous replicas are running on nodes with different characteristics than the node where the primary is running, for example, availability zone

Enhancements:

- Improve fencing by removing the existing limitation that disables failover when one or more instances are fenced
- Enhance the automated extension management framework by checking whether an extension exists in the catalog instead of running `DROP EXTENSION IF EXISTS` unnecessarily
- Improve logging of the instance manager during switchover and failover
- Enable redefining the name of the database of the application, its owner, and the related secret when recovering from an object store or cloning an instance via `pg_basebackup` (this was only possible in the `initdb` bootstrap so far)
- Backup and recovery:
  - Require Barman >= 3.0.0 for future support of PostgreSQL 15
  - Enable Azure AD Workload Identity for Barman Cloud backups through the `inheritFromAzureAD` option
  - Introduce `barmanObjectStore.s3Credentials.region` to define the region in AWS (`AWS_DEFAULT_REGION`) for both backup and recovery object stores
- Support for Kubernetes 1.24

Changes:

- Set the default operand image to PostgreSQL 15.0
- Use conditions from the Kubernetes API instead of relying on our own implementation for backup and WAL archiving

Fixes:

- Fix the initialization order inside the `WithActiveInstance` function that starts the CSV log pipe for the PostgreSQL server, ensuring proper logging in the cluster initialization phase - this is especially useful in bootstrap operations like recovery from a backup are failing (before this patch, such logs were not sent to the standard output channel and were permanently lost)
- Avoid an unnecessary switchover when a hot standby sensitive parameter is decreased, and the primary has already restarted
- Properly quote role names in `ALTER ROLE` statements
- Backup and recovery:
  - Fix the algorithm detecting the closest Barman backup for PITR, which was comparing the requested recovery timestamp with the backup start instead of the end

  - Fix Point in Time Recovery based on a transaction ID, a named restore point, or the "immediate" target by providing a new field called `backupID` in the `recoveryTarget` section
  - Fix encryption parameters invoking `barman-cloud-wal-archive` and `barman-cloud-backup` commands
  - Stop ignoring `barmanObjectStore.serverName` option when recovering from a backup object store using a server name that doesn't match the current cluster name
- `cnpg` plug-in: - Make sure that the plug-in complies with the `-n` parameter when specified by the user - Fix the `status` command to sort results and remove variability in the output

Release notes for CloudNativePG 1.15

History of user-visible changes in the 1.15 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

> **Warning:** This is expected to be the last release in the 1.15.X series. Users are encouraged to update to a newer minor version soon.

Version 1.15.5

Release date: Oct 6, 2022

Enhancements:

- Introduce `leaseDuration` and `renewDeadline` parameters in the controller manager to enhance configuration of the leader election in operator deployments (#759)
- Improve the mechanism that checks that the backup object store is empty before archiving a WAL file for the first time: a new file called `.check-empty-wal-archive` is placed in the `PGDATA` immediately after the cluster is bootstrapped and it is then removed after the first WAL file is successfully archived

Security:

- Explicitly set permissions of the instance manager binary that is copied in the `distroless/static:nonroot` container image, by using the `nonroot:nonroot` user (#754)

Fixes:

- Make the cluster's conditions compatible with `metav1.Conditions` struct (#720)
- Drop any active connection on a standby after it is promoted to primary (#737)
- Honor `MAPPEDMETRIC` and `DURATION` metric types conversion in the native Prometheus exporter (#765)

Version 1.15.4

Release date: Sep 6, 2022

Enhancements:

- Enable configuration of low-level network TCP settings in the PgBouncer connection pooler implementation (#584)
- Make sure that the `cnpg.io/instanceName` and the `cnpg.io/podRole` labels are always present on pods and PVCs (#632 and #680)
- Propagate the `role` label of an instance to the underlying PVC (#634)

Fixes:

- Prevent multiple in-place upgrade processes of the operator from running simultaneously by atomically checking whether another one is in progress (#655)
- Avoid using a hardcoded file name to store the newly uploaded instance manager, preventing a possible race condition during online upgrades of the operator (#660)
- Prevent a panic from happening when invoking `GetAllAccessibleDatabases` (#641)

Version 1.15.3

Release date: Aug 12, 2022

Enhancements:

- Enable the configuration of the `huge_pages` option for PostgreSQL (#456)
- Enhance log during promotion and demotion, after a failover or a switchover, by printing the time elapsed between the request of promotion and the actual availability for writes (#371)
- Add the `instanceName` and `clusterName` labels on jobs, pods, and PVCs to improve interaction with these resources (#534)
- Add instructions on how to create PostGIS clusters (#570)

Security:

- Explicitly assign `securityContext` to the `Pooler` deployment (#485)
- Add read timeout values to the internal web servers to prevent Slowloris DDoS (#437)

Fixes:

- Use the correct delays for restarts (`stopDelay`) and for switchover (`switchoverDelay`), as they were erroneously swapped before. This is an important fix, as it might block indefinitely restarts if `switchoverDelay` is not set and uses the default value of 40000000 seconds (#531)
- Prevent the metrics collector from causing panic when the query returns an error (#396)
- Removing an unsafe debug message that was referencing an unchecked pointer, leading in some cases to segmentation faults regardless of the log level (#491)
- Prevent panic when fencing in case the cluster had no annotation (#512)
- Avoid updating the CRD if a TLS certificate is not changed (#501)
- Handle conflicts while injecting a certificate in the CRD (#547)
- Backup and recovery: - Correctly pass object store credentials in Google Cloud (#454)

Minor changes:

- Set the default operand image to PostgreSQL 15.0

Version 1.15.2

Release date: Jul 7, 2022 (patch release)

Enhancements:

- Improve logging of the instance manager during switchover and failover
- Require Barman >= 3.0.0 for future support of PostgreSQL 15 in backup and recovery

Changes:

- Set the default operand image to PostgreSQL 15.0

Fixes:

- Fix the initialization order inside the `WithActiveInstance` function that starts the CSV log pipe for the PostgreSQL server, ensuring proper logging in the cluster initialization phase - this is especially useful in bootstrap operations like recovery from a backup are failing (before this patch, such logs were not sent to the standard output channel and were permanently lost)
- Avoid an unnecessary switchover when a hot standby sensitive parameter is decreased, and the primary has already restarted
- Properly quote role names in `ALTER ROLE` statements
- Backup and recovery:
  - Fix the algorithm detecting the closest Barman backup for PITR, which was comparing the requested recovery timestamp with the backup start instead of the end
  - Fix Point in Time Recovery based on a transaction ID, a named restore point, or the "immediate" target by providing a new field called `backupID` in the `recoveryTarget` section
  - Fix encryption parameters invoking `barman-cloud-wal-archive` and `barman-cloud-backup` commands
  - Stop ignoring `barmanObjectStore.serverName` option when recovering from a backup object store using a server name that doesn't match the current cluster name
- `cnpg` plug-in:
  - Make sure that the plug-in complies with the `-n` parameter when specified by the user
  - Fix the `status` command to sort results and remove variability in the output

Version 1.15.1

Release date: May 27, 2022 (patch release)

Minor changes:

- Enable configuration of the `archive_timeout` setting for PostgreSQL, which was previously a fixed parameter (by default set to 5 minutes)
- Introduce a new field called `backupOwnerReference` in the `scheduledBackup` resource to set the ownership reference on the created backup resources, with possible values being `none` (default), `self` (objects owned by the scheduled backup object), and `cluster` (owned by the Postgres cluster object)

- Introduce automated collection of `pg_stat_wal` metrics for PostgreSQL 14 or higher in the native Prometheus exporter
- Set the default operand image to PostgreSQL 15.0

Fixes:

- Fix fencing by killing orphaned processes related to `postgres`
- Enable the CSV log pipe inside the `WithActiveInstance` function to collect logs from recovery bootstrap jobs and help in the troubleshooting phase
- Prevent bootstrapping a new cluster with a non-empty backup object store, removing the risk of overwriting existing backups
- With the `recovery` bootstrap method, make sure that the recovery object store and the backup object store are different to avoid overwriting existing backups
- Re-queue the reconciliation loop if the RBAC for backups is not yet created
- Fix an issue with backups and the wrong specification of the cluster name property
- Ensures that operator pods always have the latest certificates in the case of a deployment of the operator in high availability, with more than one replica
- Fix the `cnpg report operator` command to correctly handle the case of a deployment of the operator in high availability, with more than one replica
- Properly propagate changes in the cluster' s `inheritedMetadata` set of labels and annotations to the related resources of the cluster without requiring a restart
- Fix the `cnpg` plugin to correctly parse any custom configmap and secret name defined in the operator deployment, instead of relying just on the default values
- Fix the local building of the documentation by using the `minidocks/mkdocs` image for `mkdocs`

Version 1.15.0

Release date: 21 April 2022

Features:

- **Fencing:** Introduction of the fencing capability for a cluster or a given set of PostgreSQL instances through the `cnpg.io/fencedInstances` annotation, which, if not empty, disables switchover/failovers in the cluster; fenced instances are shut down and the pod is kept running (while considered not ready) for inspection and emergencies
- **LDAP authentication:** Allow LDAP Simple Bind and Search+Bind configuration options in the `pg_hba.conf` to be defined in the Postgres cluster spec declaratively, enabling the optional use of Kubernetes secrets for sensitive options such as `ldapbindpasswd`
- Introduction of the `primaryUpdateMethod` option, accepting the values of `switchover` (default) and `restart` , to be used in case of unsupervised `primaryUpdateStrategy` ; this method controls what happens to the primary instance during the rolling update procedure
- New `report` command in the `kubectl cnp` plugin for better diagnosis and more effective troubleshooting of both the operator and a specific Postgres cluster
- Prune those `Backup` objects that are no longer in the backup object store
- Specification of target timeline and `LSN` in Point-In-Time Recovery bootstrap method
- Support for the `AWS_SESSION_TOKEN` authentication token in AWS S3 through the `sessionToken` option

- Default image name for PgBouncer in `Pooler` pods set to `quay.io/enterprisedb/pgbouncer:1.17.0`

Fixes:

- Base backup detection for Point-In-Time Recovery via `targetTime` correctly works now, as previously a target prior to the latest available backup was not possible (the detection algorithm was always wrong by selecting the last backup as a starting point)
- Improved resilience of hot standby sensitive parameters by relying on the values the operator collects from `pg_controldata`
- Intermediate certificates handling has been improved by properly discarding invalid entries, instead of throwing an invalid certificate error
- Prometheus exporter metric collection queries in the databases are now committed instead of rolled back (this might result in a change in the number of rolled back transactions that are visible from downstream dashboards, where applicable)

Version 1.15.0 is the first release of CloudNativePG. Previously, this software was called EDB Cloud Native PostgreSQL (now EDB Postgres for Kubernetes). If you are looking for information about a previous release, please refer to the EDB documentation .

Release notes for 1.14.0 and earlier

The first public release of CloudNativePG is version 1.15.0. Before that, the product was entirely owned by EDB and distributed under the name of Cloud Native PostgreSQL .

The list of changes in this page is only for informative purposes, to demonstrate the history of the product on top of commits. **None of the versions listed here exists for CloudNativePG.**

Version 1.14.0

Release date: 25 March 2022

Features:

- Natively support Google Cloud Storage for backup and recovery, by taking advantage of the features introduced in Barman Cloud 2.19
- Improved observability of backups through the introduction of the `LastBackupSucceeded` condition for the `Cluster` object
- Support update of Hot Standby sensitive parameters: `max_connections` , `max_prepared_transactions` , `max_locks_per_transaction` , `max_wal_senders` , `max_worker_processes`
- Add the `Online upgrade in progress` phase in the `Cluster` object to show when an online upgrade of the operator is in progress
- Ability to inherit an AWS IAM Role as an alternative way to provide credentials for the S3 object storage
- Support for Opaque secrets for Pooler' s authQuerySecret and certificates
- Updated default PostgreSQL version to 14.2
- Add a new command to `kubectl cnp` plugin named `maintenance` to set maintenance window to cluster(s) in one or all namespaces across the Kubernetes cluster

Container Images:

- Latest PostgreSQL containers include Barman Cloud 2.19

Security Enhancements:

- Stronger RBAC enforcement for namespaced operator installations with Operator Lifecycle Manager, including OpenShift. OpenShift users are recommended to update to this version.

Fixes:

- Allow the instance manager to retry an interrupted `pg_rewind` by preserving a copy of the original `pg_control` file
- Clean up stale PID files before running `pg_rewind`
- Force sorting by key in `primary_conninfo` to avoid random restarts with PostgreSQL versions prior to 13
- Preserve `ServiceAccount` changes (e.g., labels, annotations) upon reconciliation
- Disable enforcement of the imagePullPolicy default value
- Improve initDB validation for WAL segment size
- Properly handle the `targetLSN` option when recovering a cluster with the LSN specified
- Fix custom TLS certificates validation by allowing a certificates chain both in the server and CA certificates

Version 1.13.0

Release date: 17 February 2022

Features:

- Support for Snappy compression. Snappy is a fast compression option for backups that increase the speed of uploads to the object store using a lower compression ratio
- Support for tagging files uploaded to the Barman object store. This feature requires Barman 2.18 in the operand image. of backups after Cluster deletion
- Extension of the status of a Cluster with `status.conditions` . The condition `ContinuousArchiving` indicates that the Cluster has started to archive WAL files
- Improve the status command of the `cnp` plugin for `kubectl` with additional information: add a `Cluster Summary` section showing the status of the Cluster and a `Certificates Status` section including the status of the certificates used in the Cluster along with the time left to expire
- Support the new `barman-cloud-check-wal-archive` command to detect a non-empty backup destination when creating a new cluster
- Add support for using a `Secret` to add default monitoring queries through `MONITORING_QUERIES_SECRET` configuration variable.
- Allow the user to restrict container' s permissions using AppArmor (on Kubernetes clusters deployed with AppArmor support)
- Add Windows platform support to `cnp` plugin for `kubectl` , now the plugin is available on Windows x86 and ARM
- Drop support for Kubernetes 1.18 and deprecated API versions

Container Images:

- PostgreSQL containers include Barman 2.18

Security Fix:

- Add coherence check of username field inside owner and superuser secrets; previously, a malicious user could have used the secrets to change the password of any PostgreSQL user

Fixes:

- Fix a memory leak in code fetching status from Postgres pods
- Disable PostgreSQL self-restart after a crash. The instance controller handles the lifecycle of the PostgreSQL instance
- Prevent modification of `.spec.postgresUID` and `.spec.postgresGID` fields in validation webhook. Changing these fields after Cluster creation makes PostgreSQL unable to start
- Reduce the log verbosity from the backup and WAL archiving handling code
- Correct a bug resulting in a Cluster being marked as `Healthy` when not initialized yet
- Allows standby servers in clusters with a very high WAL production rate to switch to streaming once they are aligned
- Fix a race condition during the startup of a PostgreSQL pod that could seldom lead to a crash
- Fix a race condition that could lead to a failure initializing the first PVC in a Cluster
- Remove an extra restart of a just demoted primary Pod before joining the Cluster as a replica
- Correctly handle replication-sensitive PostgreSQL configuration parameters when recovering from a backup
- Fix missing validation of PostgreSQL configurations during Cluster creation

Version 1.12.0

Release date: 11 January 2022

Features:

- Add Kubernetes 1.23 to the list of supported Kubernetes distributions and remove end-to-end tests for 1.17, which ended support by the Kubernetes project in Dec 2020
- Improve the responsiveness of pod status checks in case of network issues by adding a connection timeout of 2 seconds and a communication timeout of 30 seconds. This change sets a limit on the time the operator waits for a pod to report its status before declaring it as failed, enhancing the robustness and predictability of a failover operation
- Introduce the `.spec.inheritedMetadata` field to the Cluster allowing the user to specify labels and annotations that will apply to all objects generated by the Cluster
- Reduce the number of queries executed when calculating the status of an instance
- Add a readiness probe for PgBouncer
- Add support for custom Certification Authority of the endpoint of Barman' s backup object store when using Azure protocol

Fixes:

- During a failover, wait to select a new primary until all the WAL streaming connections are closed. The operator now sets by default `wal_sender_timeout` and `wal_receiver_timeout` to 5 seconds to make sure standby nodes will quickly notice if the primary has network issues
- Change WAL archiving strategy in replica clusters to fix rolling updates by setting "archive_mode" to "always" for any PostgreSQL instance in a replica cluster. We then restrict the upload of the WAL only from the current and target designated primary. A WAL may be uploaded twice during switchovers, which is not an issue

- Fix support for custom Certification Authority of the endpoint of Barman's backup object store in replica clusters source
- Use a fixed name for default monitoring config map in the cluster namespace
- If the defaulting webhook is not working for any reason, the operator now updates the Cluster with the defaults also during the reconciliation cycle
- Fix the comparison of resource requests and limits to fix a rare issue leading to an update of all the pods on every reconciliation cycle
- Improve log messages from webhooks to also include the object namespace
- Stop logging a "default" message at the start of every reconciliation loop
- Stop logging a PodMonitor deletion on every reconciliation cycle if `enablePodMonitor` is false
- Do not complain about possible architecture mismatch if a pod is not reachable

Version 1.11.0

Release date: 15 December 2021

Features:

- **Parallel WAL archiving and restore:** allow the database to keep up with WAL generation on high write systems by introducing the `backupObjectStore.maxParallel` option to set the maximum number of parallel jobs to be executed during both WAL archiving (by PostgreSQL's `archive_command`) and WAL restore (by `restore_command`). Using parallel restore option can allow newly promoted Standbys to get to a ready state faster by fetching needed WAL files to replay in parallel rather than sequentially
- **Default set of metrics for monitoring:** a new `ConfigMap` called `default-monitoring` is automatically deployed in the same namespace of the operator and, by default, added to any existing Postgres cluster. Such behavior can be changed globally by setting the `MONITORING_QUERIES_CONFIGMAP` parameter in the operator's configuration, or at cluster level through the `.spec.monitoring.disableDefaultQueries` option (by default set to `false`)
- Introduce the `enablePodMonitor` option in the monitoring section of a cluster to automatically manage a `PodMonitor` resource and seamlessly integrate with Prometheus
- Improve the PostgreSQL shutdown procedure by trying to execute a smart shutdown for the first half of the desired `stopDelay` time, and a fast shutdown for the remaining half, before the pod is killed by Kubernetes
- Add the `switchoverDelay` option to control the time given to the former primary to shut down gracefully and archive all the WAL files before promoting the new primary (by default, CloudNativePG waits indefinitely to privilege data durability)
- Handle changes to resource requests and limits for a PostgreSQL `Cluster` by issuing a rolling update
- Improve the `status` command of the `cnp` plugin for `kubectl` with additional information: streaming replication status, total size of the database, role of an instance in the cluster
- Enhance support of workloads with many parallel workers by enabling configuration of the `dynamic_shared_memory_type` and `shared_memory_type` parameters for PostgreSQL's management of shared memory
- Propagate labels and annotations defined at cluster level to the associated resources, including pods (deletions are not supported)
- Automatically remove pods that have been evicted by the Kubelet

- Manage automated resizing of persistent volumes in Azure through the `ENABLE_AZURE_PVC_UPDATES` operator configuration option, by issuing a rolling update of the cluster if needed (disabled by default)
- Introduce the`cnpg.io/reconciliationLoop` annotation that, when set to `disabled` on a given Postgres cluster, prevents the reconciliation loop from running
- Introduce the `postInitApplicationSQL` option as part of the `initdb` bootstrap method to specify a list of SQL queries to be executed on the main application database as a superuser immediately after the cluster has been created

Fixes:

- Liveness probe now correctly handles the startup process of a PostgreSQL server. This fixes an issue reported by a few customers and affects a restarted standby server that needs to recover WAL files to reach a consistent state, but it was not able to do it before the timeout of liveness probe would kick in, leaving the pods in `CrashLoopBackOff` status.
- Liveness probe now correctly handles the case of a former primary that needs to use `pg_rewind` to re-align with the current primary after a timeline diversion. This fixes the pod of the new standby from repeatedly being killed by Kubernetes.
- Reduce client-side throttling from Postgres pods (e.g. `Waited for 1.182388649s due to client-side throttling, not priority and fairness, request: GET`)
- Disable Public Key Infrastructure (PKI) initialization on OpenShift and OLM installations, by using the provided one
- When changing configuration parameters that require a restart, always leave the primary as last
- Mark a PVC to be ready only after a job has been completed successfully, preventing a race condition in PVC initialization
- Use the correct public key when renewing the expired webhook TLS secret.
- Fix an overflow when parsing an LSN
- Remove stale PID files at startup
- Let the `Pooler` resource inherit the `imagePullSecret` defined in the operator, if exists

Version 1.10.0

Release date: 11 November 2021

Features:

- **Connection Pooling with PgBouncer**: introduce the `Pooler` resource and controller to automatically manage a PgBouncer deployment to be used as a connection pooler for a local PostgreSQL `Cluster`. The feature includes TLS client/server connections, password authentication, High Availability, pod templates support, configuration of key PgBouncer parameters, `PAUSE` /`RESUME`, logging in JSON format, Prometheus exporter for stats, pools, and lists
- **Backup Retention Policies**: support definition of recovery window retention policies for backups (e.g. '30d' to ensure a recovery window of 30 days)
- **In-Place updates of the operator**: introduce an in-place online update of the instance manager, which removes the need to perform a rolling update of the entire cluster following an update of the operator. By default this option is disabled (please refer to the *documentation for more detailed information*)
- Limit the list of options that can be customized in the `initdb` bootstrap method to `dataChecksums`, `encoding`, `localeCollate`, `localeCType`, `walSegmentSize`. This makes the `options` array obsolete and planned to be removed in the v2 API

- Introduce the `postInitTemplateSQL` option as part of the `initdb` bootstrap method to specify a list of SQL queries to be executed on the `template1` database as a superuser immediately after the cluster has been created. This feature allows you to include default objects in all application databases created in the cluster
- New default metrics added to the instance Prometheus exporter: Postgres version, cluster name, and first point of recoverability according to the backup catalog
- Retry taking a backup after a failure
- Build awareness about Barman Cloud capabilities in order to prevent the operator from invoking recently introduced features (such as retention policies, or Azure Blob Container storage) that are not present in operand images that are not frequently updated
- Integrate the output of the `status` command of the `cnp` plugin with information about the backup
- Introduce a new annotation that reports the status of a PVC (being initialized or ready)
- Set the cluster name in the `k8s.enterprisedb.io/cluster` label for every object generated in a `Cluster` , including `Backup` objects
- Drop support for deprecated API version `postgresql.cnpg.io/v1alpha1` on the `Cluster` , `Backup` , and `ScheduledBackup` kinds
- Set default operand image to PostgreSQL 14.2

Security:

- Set allowPrivilegeEscalation to `false` for the operator containers securityContext

Fixes:

- Disable primary PodDisruptionBudget during maintenance in single-instance clusters
- Use the correct certificate certification authority (CA) during recovery operations
- Prevent Postgres connection leaking when checking WAL archiving status before taking a backup
- Let WAL archive/restore sleep for 100ms following transient errors that would flood logs otherwise

Version 1.9.2

Release date: 15 October 2021

Features:

- Enhance JSON log with two new loggers: `wal-archive` for PostgreSQL' s `archive_command` , and `wal-restore` for `restore_command` in a standby

Fixes:

- Enable WAL archiving during the standby promotion (prevented `.history` files from being archived)
- Pass the `--cloud-provider` option to Barman Cloud tools only when using Barman 2.13 or higher to avoid errors with older operands
- Wait for the pod of the primary to be ready before triggering a backup

Version 1.9.1

Release date: 30 September 2021

- This release is to celebrate the launch of PostgreSQL 14

by making it the default major version when a new `Cluster` is created without defining a specific image name.*

Fixes:

- Fix issue causing `Error while getting barman endpoint CA secret` message to appear in the logs of the primary pod, which prevented the backup to work correctly
- Properly retry requesting a new backup in case of temporary communication issues with the instance manager

Version 1.9.0

Release date: 28 September 2021

- Version 1.9.0 is not available on OpenShift due to delays with the release process and the subsequent release of version 1.9.1.*

Features:

- Add Kubernetes 1.22 to the list of supported Kubernetes distributions, and remove 1.16
- Introduce support for the `--restore-target-wal` option in `pg_rewind` , in order to fetch WAL files from the backup archive, if necessary (available only with PostgreSQL 13+)
- Expose a default metric for the Prometheus exporter that estimates the number of pages in the `pg_catalog.pg_largeobject` table in each database
- Enhance the performance of WAL archiving and fetching, through local in-memory cache

Fixes:

- Explicitly set the `postgres` user when invoking `pg_isready` - required by restricted SCC in OpenShift
- Properly update the `FirstRecoverabilityPoint` in the status
- Set `archive_mode = always` on the designated primary if backup is requested
- Minor bug fixes

Version 1.8.0

Release date: 13 September 2021

Features:

- Bootstrap a new cluster via full or Point-In-Time Recovery directly from an object store defined in the external cluster section, eliminating the previous requirement to have a Backup CR defined
- Introduce the `immediate` option in scheduled backups to request a backup immediately after the first Postgres instance running, adding the capability to rewind to the very beginning of a cluster when Point-In-Time Recovery is configured
- Add the `firstRecoverabilityPoint` in the cluster status to report the oldest consistent point in time to request a recovery based on the backup object store' s content
- Enhance the default Prometheus exporter for a PostgreSQL instance by exposing the following new metrics:

    1. number of WAL files and computed total size on disk
    2. number of `.ready` and `.done` files in the archive status folder
    3. flag for replica mode
    4. number of requested minimum/maximum synchronous replicas, as well as the expected and actually observed ones
- Add support for the `runonserver` option when defining custom metrics in the Prometheus exporter to limit the collection of a metric to a range of PostgreSQL versions
- Natively support Azure Blob Storage for backup and recovery, by taking advantage of the feature introduced in Barman 2.13 for Barman Cloud
- Rely on `pg_isready` for the liveness probe
- Support RFC3339 format for timestamp specification in recovery target times
- Introduce `.spec.imagePullPolicy` to control the pull policy of image containers for all pods and jobs created for a cluster
- Add support for OpenShift 4.8, which replaces OpenShift 4.5
- Support PostgreSQL 14 (beta)
- Enhance the replica cluster feature with cross-cluster replication from an object store defined in an external cluster section, without requiring a streaming connection (experimental)
- Introduce `logLevel` option to the cluster' s spec to specify one of the following levels: error, info, debug or trace

Security Enhancements:

- Introduce `.spec.enableSuperuserAccess` to enable/disable network access with the `postgres` user through password authentication

Fixes:

- Properly inform users when a cluster enters an unrecoverable state and requires human intervention

Version 1.7.1

Release date: 11 August 2021

Features:

- Prefer self-healing over configuration with regards to synchronous replication, empowering the operator to temporarily override `minSyncReplicas` and `maxSyncReplicas` settings in case the cluster is not able to meet the requirements during self-healing operations
- Introduce the `postInitSQL` option as part of the `initdb` bootstrap method to specify a list of SQL queries to be executed as a superuser immediately after the cluster has been created

Fixes:

- Allow the operator to failover when the primary is not ready (bug introduced in 1.7.0)
- Execute administrative queries using the `LOCAL` synchronous commit level
- Correctly parse multi-line log entries in PGAudit

Version 1.7.0

Release date: 28 July 2021

Features:

- Add native support to PGAudit with a new type of `logger` called `pgaudit` directly available in the JSON output
- Enhance monitoring and observability capabilities through:
  - Native support for the `pg_stat_statements` and `auto_explain` extensions
  - The `target_databases` option in the Prometheus exporter to run a user-defined metric query on one or more databases (including auto-discovery of databases through shell-like pattern matching)
  - Exposure of the `manual_switchover_required` metric to promptly report whether a cluster with `primaryUpdateStrategy` set to `supervised` requires a manual switchover
- Transparently handle `shared_preload_libraries` for `pg_audit` , `auto_explain` and `pg_stat_statements`
  - Automatic configuration of `shared_preload_libraries` for PostgreSQL when `pg_stat_statements` , `pgaudit` or `auto_explain` options are added to the `postgresql` parameters section
- Support the `cnpg.io/reload` label to finely control the automated reload of config maps and secrets, including those used for custom monitoring/alerting metrics in the Prometheus exporter or to store certificates
- Add the `reload` command to the `cnp` plugin for `kubectl` to trigger a reconciliation loop on the instances
- Improve control of pod affinity and anti-affinity configurations through `additionalPodAffinity` and `additionalPodAntiAffinity`
- Introduce a separate `PodDisruptionBudget` for primary instances, by requiring at least a primary instance to run at any time

Security Enhancements:

- Add the `.spec.certificates.clientCASecret` and `.spec.certificates.replicationTLSSecret` options to define custom client Certification Authority and certificate for the PostgreSQL server, to be used to authenticate client certificates and secure communication between PostgreSQL nodes
- Add the `.spec.backup.barmanObjectStore.endpointCA` option to define the custom Certification Authority bundle of the endpoint of Barman' s backup object store

Fixes:

- Correctly parse histograms in the Prometheus exporter
- Reconcile services created by the operator for a cluster

Version 1.6.0

Release date: 12 July 2021

Features:

- Replica mode (**EXPERIMENTAL**): allow a cluster to be created as a replica of a source cluster. A replica cluster has a *designated primary* and any number of standbys.
- Add the `.spec.postgresql.promotionTimeout` parameter to specify the maximum amount of seconds to wait when promoting an instance to primary, defaulting to 40000000 seconds.

- Add the `.spec.affinity.podAntiAffinityType` parameter. It can be set to `preferred` (default), resulting in `preferredDuringSchedulingIgnoredDuringExecution` being used, or to `required` , resulting in `requiredDuringSchedulingIgnoredDuringExecution` .

Changes:

- Fixed a race condition when deleting a PVC and a pod which prevented the operator from creating a new pod.
- Fixed a race condition preventing the manager from detecting the need for a PostgreSQL restart on a configuration change.
- Fixed a panic in `kubectl-cnp` on clusters without annotations.
- Lowered the level of some log messages to `debug` .
- E2E tests for server CA and TLS injection.

Version 1.5.1

Release date: 17 June 2021

Change:

- Fix a bug with CRD validation preventing auto-update with Operator Deployments on Red Hat OpenShift
- Allow passing operator' s configuration using a Secret.

Version 1.5.0

Release date: 11 June 2021

Features:

- Introduce the `pg_basebackup` bootstrap method to create a new PostgreSQL cluster as a copy of an existing PostgreSQL instance of the same major version, even outside Kubernetes
- Add support for Kubernetes' tolerations in the `Affinity` section of the `Cluster` resource, allowing users to distribute PostgreSQL instances on Kubernetes nodes with the required taint
- Enable specification of a digest to an image name, through the `<image>:<tag>@sha256:<digestValue>` format, for more deterministic and repeatable deployments

Security Enhancements:

- Customize TLS certificates to authenticate the PostgreSQL server by defining secrets for the server certificate and the related Certification Authority that signed it
- Raise the `sslmode` for the WAL receiver process of internal and automatically managed streaming replicas from `require` to `verify-ca`

Changes:

- Enhance the `promote` subcommand of the `cnp` plugin for `kubectl` to accept just the node number rather than the whole name of the pod
- Adopt DNS-1035 validation scheme for cluster names (from which service names are inherited)
- Enforce streaming replication connection when cloning a standby instance or when bootstrapping using the `pg_basebackup` method
- Integrate the `Backup` resource with `beginWal` , `endWal` , `beginLSN` , `endLSN` , `startedAt` and `stoppedAt` regarding the physical base backup

- Documentation improvements:
  - Provide a list of ports exposed by the operator and the operand container
  - Introduce the `cnp-bench` helm charts and guidelines for benchmarking the storage and PostgreSQL for database workloads
- E2E tests enhancements:
  - Test Kubernetes 1.21
  - Add test for High Availability of the operator
  - Add test for node draining
- Minor bug fixes, including:
  - Timeout to pg_ctl start during recovery operations too short
  - Operator not watching over direct events on PVCs
  - Fix handling of `immediateCheckpoint` and `jobs` parameter in `barmanObjectStore` backups
  - Empty logs when recovering from a backup

Version 1.4.0

Release date: 18 May 2021

Features:

- Standard output logging of PostgreSQL error messages in JSON format
- Provide a basic set of PostgreSQL metrics for the Prometheus exporter
- Add the `restart` command to the `cnp` plugin for `kubectl` to restart the pods of a given PostgreSQL cluster in a rollout fashion

Security Enhancements:

- Set `readOnlyRootFilesystem` security context for pods

Changes:

- **IMPORTANT:** If you have previously deployed the CloudNativePG operator using the YAML manifest, you must delete the existing operator deployment before installing the new version. This is required to avoid conflicts with other Kubernetes API' s due to a change in labels and label selectors being directly managed by the operator. Please refer to the CloudNativePG documentation for additional detail on upgrading to 1.4.0
- Fix the labels that are automatically defined by the operator, renaming them from `control-plane: controller-manager` to `app.kubernetes.io/name: cloudnative-pg`
- Assign the `metrics` name to the TCP port for the Prometheus exporter
- Set `cnp_metrics_exporter` as the `application_name` to the metrics exporter connection in PostgreSQL
- When available, use the application database for monitoring queries of the Prometheus exporter instead of the `postgres` database
- Documentation improvements:
  - Customization of monitoring queries
  - Operator upgrade instructions
- E2E tests enhancements

- Minor bug fixes, including:
  - Avoid using `-R` when calling `pg_basebackup`
  - Remove stack trace from error log when getting the status

Version 1.3.0

Release date: 23 Apr 2021

Features:

- Inheritance of labels and annotations
- Set resource limits for every container

Security Enhancements:

- Support for restricted security context constraint on Red Hat OpenShift to limit pod execution to a namespace allocated UID and SELinux context
- Pod security contexts explicitly defined by the operator to run as non-root, non-privileged and without privilege escalation

Changes:

- Prometheus exporter endpoint listening on port 9187 (port 8000 is now reserved to instance coordination with API server)
- Documentation improvements
- E2E tests enhancements, including GKE environment
- Minor bug fixes

Version 1.2.1

Release date: 6 Apr 2021

- ScheduledBackup are no longer owners of the Backups, meaning that backups are not removed when ScheduledBackup objects are deleted
- Update on ubi8-minimal image to solve RHSA-2021:1024 (Security Advisory: Important)

Version 1.2.0

Release date: 31 Mar 2021

- Introduce experimental support for custom monitoring queries as ConfigMap and Secret objects using a compatible syntax with `postgres_exporter` for Prometheus
- Support Operator Lifecycle Manager (OLM) deployments, with the subsequent presence on OperatorHub.io
- Enhance container security by applying guidelines from the US Department of Defense (DoD)' s Defense Information Systems Agency (DISA) and the Center for Internet Security (CIS) and verifying them directly in the pipeline with Dockle
- Improve E2E tests on AKS
- Minor bug fixes

Version 1.1.0

Release date: 3 Mar 2021

- Add `kubectl cnp status` to pretty-print the status of a cluster, including JSON and YAML output
- Add `kubectl cnp certificate` to enable TLS authentication for client applications
- Add the `-ro` service to route connections to the available hot standby replicas only, enabling offload of read-only queries from the cluster' s primary instance
- Rollback scaling down a cluster to a value lower than `maxSyncReplicas`
- Request a checkpoint before demoting a former primary
- Send `SIGINT` signal (fast shutdown) to PostgreSQL process on `SIGTERM`
- Minor bug fixes

Version 1.0.0

Release date: 4 Feb 2021

The first major stable release of CloudNativePG implements `Cluster` , `Backup` and `ScheduledBackup` in the API group `postgresql.cnpg.io/v1` . It uses these resources to create and manage PostgreSQL clusters inside Kubernetes with the following main capabilities:

- Direct integration with Kubernetes API server for High Availability, without requiring an external tool
- Self-Healing capability, through:
  - failover of the primary instance by promoting the most aligned replica
  - automated recreation of a replica
- Planned switchover of the primary instance by promoting a selected replica
- Scale up/down capabilities
- Definition of an arbitrary number of instances (minimum 1 - one primary server)
- Definition of the *read-write* service to connect your applications to the only primary server of the cluster
- Definition of the *read* service to connect your applications to any of the instances for reading workloads
- Support for Local Persistent Volumes with PVC templates
- Reuse of Persistent Volumes storage in Pods
- Rolling updates for PostgreSQL minor versions and operator upgrades
- TLS connections and client certificate authentication
- Continuous backup to an S3 compatible object store
- Full recovery and point-in-time recovery from an S3 compatible object store backup
- Support for synchronous replicas
- Support for node affinity via `nodeSelector` property
- Standard output logging of PostgreSQL error messages

1.4.57 Appendixes

1.4.58 Appendixes

Appendix A - Common object stores for backups

You can store the *Appendix A - Common object stores for backups* files in any service that is supported by the Barman Cloud infrastructure. That is:

- Amazon S3
- *Microsoft Azure Blob Storage*
- *Google Cloud Storage*

You can also use any compatible implementation of the supported services.

The required setup depends on the chosen storage provider and is discussed in the following sections.

AWS S3

AWS Simple Storage Service (S3) is a very popular object storage service offered by Amazon.

As far as CloudNativePG backup is concerned, you can define the permissions to store backups in S3 buckets in two ways:

- If CloudNativePG is running in EKS. you may want to use the IRSA authentication method
- Alternatively, you can use the `ACCESS_KEY_ID` and `ACCESS_SECRET_KEY` credentials

AWS Access key

You will need the following information about your environment:

- `ACCESS_KEY_ID` : the ID of the access key that will be used to upload files into S3
- `ACCESS_SECRET_KEY` : the secret part of the access key mentioned above
- `ACCESS_SESSION_TOKEN` : the optional session token, in case it is required

The access key used must have permission to upload files into the bucket. Given that, you must create a Kubernetes secret with the credentials, and you can do that with the following command:

```
kubectl create secret generic aws-creds \
  --from-literal=ACCESS_KEY_ID=<access key here> \
  --from-literal=ACCESS_SECRET_KEY=<secret key here>
#  --from-literal=ACCESS_SESSION_TOKEN=<session token here> # if required
```

The credentials will be stored inside Kubernetes and will be encrypted if encryption at rest is configured in your installation.

Once that secret has been created, you can configure your cluster like in the following example:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path here>"
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
```

The destination path can be any URL pointing to a folder where the instance can upload the WAL files, e.g. `s3://BUCKET_NAME/path/to/folder` .

IAM Role for Service Account (IRSA)

In order to use IRSA you need to set an `annotation` in the `ServiceAccount` of the Postgres cluster.

We can configure CloudNativePG to inject them using the `serviceAccountTemplate` stanza:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
[...]
spec:
  serviceAccountTemplate:
    metadata:
      annotations:
        eks.amazonaws.com/role-arn: arn:[...]
        [...]
```

S3 lifecycle policy

Barman Cloud writes objects to S3, then does not update them until they are deleted by the Barman Cloud retention policy. A recommended approach for an S3 lifecycle policy is to expire the current version of objects a few days longer than the Barman retention policy, enable object versioning, and expire non-current versions after a number of days. Such a policy protects against accidental deletion, and also allows for restricting permissions to the CloudNativePG workload so that it may delete objects from S3 without granting permissions to permanently delete objects.

Other S3-compatible Object Storages providers

In case you're using S3-compatible object storage, like **MinIO** or **Linode Object Storage**, you can specify an endpoint instead of using the default S3 one.

In this example, it will use the `bucket` of **Linode** in the region `us-east1` .

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "s3://bucket/"
      endpointURL: "https://us-east1.linodeobjects.com"
      s3Credentials:
        [...]
```

In case you're using **Digital Ocean Spaces**, you will have to use the Path-style syntax. In this example, it will use the `bucket` from **Digital Ocean Spaces** in the region `SFO3` .

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "s3://[your-bucket-name]/[your-backup-folder]/"
      endpointURL: "https://sfo3.digitaloceanspaces.com"
      s3Credentials:
        [...]
```

Important: Suppose you configure an Object Storage provider which uses a certificate signed with a private CA, like when using MinIO via HTTPS. In that case, you need to set the option *endpointCA* referring to a secret containing the CA bundle so that Barman can verify the certificate correctly.

Note: If you want ConfigMaps and Secrets to be **automatically** reloaded by instances, you can add a label with key *cnpg.io/reload* to the Secrets/ConfigMaps. Otherwise, you will have to reload the instances using the *kubectl cnpg reload* subcommand.

Azure Blob Storage

Azure Blob Storage is the obect storage service provided by Microsoft.

In order to access your storage account for backup and recovery of CloudNativePG managed databases, you will need one of the following combinations of credentials:

- *Connection String*
- Storage account name and *Storage account access key*
- Storage account name and *Storage account SAS Token*

- Storage account name and Azure AD Workload Identity

properly configured.

Using **Azure AD Workload Identity**, you can avoid saving the credentials into a Kubernetes Secret, and have a Cluster configuration adding the `inheritFromAzureAD` as follows:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path here>"
      azureCredentials:
        inheritFromAzureAD: true
```

On the other side, using both **Storage account access key** or **Storage account SAS Token**, the credentials need to be stored inside a Kubernetes Secret, adding data entries only when needed. The following command performs that:

```
kubectl create secret generic azure-creds \
  --from-literal=AZURE_STORAGE_ACCOUNT=<storage account name> \
  --from-literal=AZURE_STORAGE_KEY=<storage account key> \
  --from-literal=AZURE_STORAGE_SAS_TOKEN=<SAS token> \
  --from-literal=AZURE_STORAGE_CONNECTION_STRING=<connection string>
```

The credentials will be encrypted at rest, if this feature is enabled in the used Kubernetes cluster.

Given the previous secret, the provided credentials can be injected inside the cluster configuration:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path here>"
      azureCredentials:
        connectionString:
          name: azure-creds
          key: AZURE_CONNECTION_STRING
        storageAccount:
          name: azure-creds
          key: AZURE_STORAGE_ACCOUNT
        storageKey:
          name: azure-creds
          key: AZURE_STORAGE_KEY
        storageSasToken:
          name: azure-creds
          key: AZURE_STORAGE_SAS_TOKEN
```

When using the Azure Blob Storage, the `destinationPath` fulfills the following structure:

```
<http|https>://<account-name>.<service-name>.core.windows.net/<resource-path>
```

where <resource-path> is <container>/<blob> . The **account name**, which is also called **storage account name**, is included in the used host name.

Other Azure Blob Storage compatible providers

If you are using a different implementation of the Azure Blob Storage APIs, the `destinationPath` will have the following structure:

```
<http|https>://<local-machine-address>:<port>/<account-name>/<resource-path>
```

In that case, <account-name> is the first component of the path.

This is required if you are testing the Azure support via the Azure Storage Emulator or Azurite .

Google Cloud Storage

Currently, the CloudNativePG operator supports two authentication methods for Google Cloud Storage :

- the first one assumes that the pod is running inside a Google Kubernetes Engine cluster
- the second one leverages the environment variable `GOOGLE_APPLICATION_CREDENTIALS`

Running inside Google Kubernetes Engine

When running inside Google Kubernetes Engine you can configure your backups to simply rely on Workload Identity , without having to set any credentials. In particular, you need to:

- set `.spec.backup.barmanObjectStore.googleCredentials.gkeEnvironment` to `true`
- set the `iam.gke.io/gcp-service-account` annotation in the `serviceAccountTemplate` stanza

Please use the following example as a reference:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  [...]
  backup:
    barmanObjectStore:
      destinationPath: "gs://<destination path here>"
      googleCredentials:
        gkeEnvironment: true

  serviceAccountTemplate:
    metadata:
      annotations:
        iam.gke.io/gcp-service-account:  [...].iam.gserviceaccount.com
        [...]
```

Using authentication

Following the instruction from Google

you will get a JSON file that contains all the required information to authenticate.

The content of the JSON file must be provided using a `Secret` that can be created with the following command:

```
kubectl create secret generic backup-creds --from-file=gcsCredentials=gcs_credentials_
↪file.json
```

This will create the `Secret` with the name `backup-creds` to be used in the yaml file like this:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "gs://<destination path here>"
      googleCredentials:
        applicationCredentials:
          name: backup-creds
          key: gcsCredentials
```

Now the operator will use the credentials to authenticate against Google Cloud Storage.

Important: This way of authentication will create a JSON file inside the container with all the needed information to access your Google Cloud Storage bucket, meaning that if someone gets access to the pod will also have write permissions to the bucket.

MinIO Gateway

Optionally, you can use MinIO Gateway as a common interface which relays backup objects to other cloud storage solutions, like S3 or GCS. For more information, please refer to MinIO official documentation .

Specifically, the CloudNativePG cluster can directly point to a local MinIO Gateway as an endpoint, using previously created credentials and service.

MinIO secrets will be used by both the PostgreSQL cluster and the MinIO instance. Therefore, you must create them in the same namespace:

```
kubectl create secret generic minio-creds \
  --from-literal=MINIO_ACCESS_KEY=<minio access key here> \
  --from-literal=MINIO_SECRET_KEY=<minio secret key here>
```

Note: Cloud Object Storage credentials will be used only by MinIO Gateway in this case.

Important: In order to allow PostgreSQL to reach MinIO Gateway, it is necessary to create a *ClusterIP* service on port *9000* bound to the MinIO Gateway instance.

For example:

```
apiVersion: v1
kind: Service
metadata:
  name: minio-gateway-service
spec:
  type: ClusterIP
  ports:
    - port: 9000
      targetPort: 9000
      protocol: TCP
  selector:
    app: minio
```

Warning: At the time of writing this documentation, the official MinIO Operator

for Kubernetes does not support the gateway feature. As such, we will use a `deployment` instead.

The MinIO deployment will use cloud storage credentials to upload objects to the remote bucket and relay backup files to different locations.

Here is an example using AWS S3 as Cloud Object Storage:

```
apiVersion: apps/v1
kind: Deployment
[...]
spec:
  containers:
  - name: minio
    image: minio/minio:RELEASE.2020-06-03T22-13-49Z
    args:
    - gateway
    - s3
    env:
    # MinIO access key and secret key
    - name: MINIO_ACCESS_KEY
      valueFrom:
        secretKeyRef:
          name: minio-creds
          key: MINIO_ACCESS_KEY
    - name: MINIO_SECRET_KEY
      valueFrom:
        secretKeyRef:
          name: minio-creds
          key: MINIO_SECRET_KEY
    # AWS credentials
    - name: AWS_ACCESS_KEY_ID
      valueFrom:
        secretKeyRef:
          name: aws-creds
          key: ACCESS_KEY_ID
```

(continues on next page)

(continued from previous page)

```
    - name: AWS_SECRET_ACCESS_KEY
      valueFrom:
        secretKeyRef:
          name: aws-creds
          key: ACCESS_SECRET_KEY
#  Uncomment the below section if session token is required

#    - name: AWS_SESSION_TOKEN

#      valueFrom:

#        secretKeyRef:

#          name: aws-creds

#          key: ACCESS_SESSION_TOKEN

        ports:
        - containerPort: 9000
```

Proceed by configuring MinIO Gateway service as the `endpointURL` in the `Cluster` definition, then choose a bucket name to replace `BUCKET_NAME` :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/
      endpointURL: http://minio-gateway-service:9000
      s3Credentials:
        accessKeyId:
          name: minio-creds
          key: MINIO_ACCESS_KEY
        secretAccessKey:
          name: minio-creds
          key: MINIO_SECRET_KEY
    [...]
```

Verify on `s3://BUCKET_NAME/` the presence of archived WAL files before proceeding with a backup.