CloudNativePG

Release 1.19.1

The CloudNativePG Contributors.

Mar 25, 2023

CONTENTS

1 Supported Kubernetes distributions **3**

2 Container images **5**

3 Main features **7**

4 About this guide **9**

4.1 Before You Start . . . 9
4.1.1 Kubernetes terminology . . . 9
4.1.2 PostgreSQL terminology . . . 10
4.1.3 Cloud terminology . . . 11
4.1.4 What to do next . . . 11
4.2 Use cases . . . 11
4.2.1 Case 1: Applications inside Kubernetes . . . 11
4.2.2 Case 2: Applications outside Kubernetes . . . 14
4.3 Architecture . . . 14
4.3.1 Synchronizing the state . . . 14
4.3.2 Kubernetes architecture . . . 15
4.3.3 PostgreSQL architecture . . . 17
4.3.4 Deployments across Kubernetes clusters . . . 19
4.4 Installation and upgrades . . . 20
4.4.1 Installation on Kubernetes . . . 20
4.4.2 Details about the deployment . . . 21
4.4.3 Upgrades . . . 22
4.5 Quickstart . . . 23
4.5.1 Part 1 - Setup the local Kubernetes playground . . . 24
4.5.2 Part 2 - Install CloudNativePG . . . 24
4.5.3 Part 3 - Deploy a PostgreSQL cluster . . . 24
4.5.4 Part 4 - Monitor clusters with Prometheus and Grafana . . . 26
4.6 Bootstrap . . . 29
4.6.1 The `bootstrap` section . . . 30
4.6.2 The `externalClusters` section . . . 30
4.6.3 Bootstrap an empty cluster (`initdb`) . . . 31
4.6.4 Bootstrap from another cluster . . . 34
4.7 Importing Postgres databases . . . 45
4.7.1 How it works . . . 46
4.7.2 The `microservice` type . . . 46
4.7.3 The `monolith` type . . . 48
4.8 Security . . . 50
4.8.1 Code . . . 50

4.8.2 Container . . . 50
4.8.3 Cluster . . . 51
4.9 Postgres instance manager . . . 55
4.9.1 Liveness and readiness probes . . . 55
4.9.2 Shutdown control . . . 55
4.9.3 Failover . . . 56
4.10 Scheduling . . . 56
4.10.1 Pod affinity and anti-affinity . . . 57
4.10.2 Node selection through `nodeSelector` . . . 58
4.10.3 Tolerations . . . 58
4.11 Resource management . . . 58
4.12 Failure Modes . . . 60
4.12.1 Storage space usage . . . 60
4.12.2 Failure modes . . . 61
4.12.3 Self-healing . . . 62
4.12.4 Manual intervention . . . 63
4.13 Rolling Updates . . . 63
4.13.1 Automated updates (`unsupervised`) . . . 64
4.13.2 Manual updates (`supervised`) . . . 64
4.14 Replication . . . 65
4.14.1 Application-level replication . . . 65
4.14.2 A very mature technology . . . 65
4.14.3 Streaming replication support . . . 65
4.14.4 Synchronous replication . . . 66
4.14.5 Replication slots for High Availability . . . 67
4.15 Backup and Recovery . . . 69
4.15.1 Cloud provider support . . . 70
4.15.2 On-demand backups . . . 77
4.15.3 Scheduled backups . . . 78
4.15.4 WAL archiving . . . 79
4.15.5 Backup from a standby . . . 80
4.15.6 Recovery . . . 81
4.15.7 Retention policies . . . 82
4.15.8 Compression algorithms . . . 83
4.15.9 Tagging of backup objects . . . 83
4.16 PostgreSQL Configuration . . . 84
4.16.1 The `postgresql` section . . . 84
4.16.2 The `pg_hba` section . . . 88
4.16.3 Changing configuration . . . 90
4.16.4 Dynamic Shared Memory settings . . . 90
4.16.5 Fixed parameters . . . 91
4.17 Operator configuration . . . 93
4.17.1 Available options . . . 94
4.17.2 Defining an operator config map . . . 94
4.17.3 Defining an operator secret . . . 95
4.17.4 Restarting the operator to reload configs . . . 95
4.17.5 PPROF HTTP SERVER . . . 95
4.18 Instance Pod Configuration . . . 96
4.18.1 Projected Volumes . . . 96
4.18.2 Environment variables . . . 97
4.19 Storage . . . 98
4.19.1 Benchmarking CloudNativePG . . . 99
4.19.2 Persistent Volume Claim . . . 99
4.19.3 Configuration via a storage class . . . 99

4.19.4 Configuration via a PVC template . . . 100
4.19.5 Volume for WAL . . . 100
4.19.6 Volume expansion . . . 101
4.20 Labels and annotations . . . 104
4.20.1 Pre-requisites . . . 105
4.20.2 Defining cluster's metadata . . . 105
4.20.3 Current limitations . . . 105
4.21 Monitoring . . . 106
4.21.1 Monitoring Instances . . . 106
4.21.2 Monitoring the operator . . . 118
4.21.3 How to inspect the exported metrics . . . 119
4.21.4 Auxiliary resources . . . 120
4.22 Logging . . . 120
4.22.1 Operator log . . . 121
4.22.2 PostgreSQL log . . . 121
4.22.3 PGAudit logs . . . 122
4.22.4 Other logs . . . 123
4.23 Certificates . . . 124
4.23.1 Operator managed mode . . . 124
4.23.2 User-provided certificates mode . . . 125
4.24 Client TLS/SSL Connections . . . 129
4.24.1 Issuing a new certificate . . . 130
4.24.2 Testing the connection via a TLS certificate . . . 130
4.25 Connecting from an application . . . 132
4.25.1 DNS resolution . . . 133
4.25.2 Environment variables . . . 133
4.25.3 Secrets . . . 133
4.26 Connection Pooling . . . 133
4.26.1 Architecture . . . 134
4.26.2 Quickstart . . . 134
4.26.3 Pooler resource lifecycle . . . 135
4.26.4 Security . . . 135
4.26.5 Authentication . . . 136
4.26.6 PodTemplates . . . 137
4.26.7 High Availability (HA) . . . 138
4.26.8 PgBouncer configuration options . . . 139
4.26.9 Monitoring . . . 140
4.26.10 Logging . . . 146
4.26.11 Pausing connections . . . 146
4.26.12 Limitations . . . 147
4.27 Replica clusters . . . 147
4.27.1 Basic concepts . . . 147
4.27.2 Setting up a replica cluster . . . 148
4.27.3 Promoting the designated primary in the replica cluster . . . 150
4.28 Kubernetes Upgrade . . . 150
4.28.1 Single instance clusters with `reusePVC` set to `false` . . . 151
4.29 Exposing Postgres Services . . . 152
4.29.1 Testing on Minikube . . . 154
4.30 CloudNativePG Plugin . . . 154
4.30.1 Install . . . 154
4.30.2 Use . . . 156
4.31 Automated failover . . . 168
4.31.1 RTO and RPO impact . . . 169
4.31.2 Delayed failover . . . 169

4.32 Troubleshooting . . . 169
4.32.1 Before you start . . . 170
4.32.2 Emergency backup . . . 170
4.32.3 Operator information . . . 172
4.32.4 Cluster information . . . 173
4.32.5 Pod information . . . 174
4.32.6 Backup information . . . 176
4.32.7 Storage information . . . 176
4.32.8 Node information . . . 177
4.32.9 Conditions . . . 177
4.32.10 Some common issues . . . 178
4.33 Fencing . . . 179
4.33.1 How to fence instances . . . 179
4.33.2 How to lift fencing . . . 180
4.33.3 How fencing works . . . 180
4.34 PostGIS . . . 181
4.34.1 Basic concepts about a PostGIS cluster . . . 181
4.34.2 Create a new PostgreSQL cluster with PostGIS . . . 182
4.35 End-to-End Tests . . . 184
4.36 Container Image Requirements . . . 185
4.36.1 Image tag requirements . . . 185
4.37 Operator Capability Levels . . . 186
4.37.1 Level 1 - Basic Install . . . 187
4.37.2 Level 2 - Seamless Upgrades . . . 190
4.37.3 Level 3 - Full Lifecycle . . . 191
4.37.4 Level 4 - Deep Insights . . . 194
4.37.5 Level 5 - Auto Pilot . . . 195
4.38 Custom Pod Controller . . . 195
4.38.1 PVC resizing . . . 196
4.38.2 Primary Instances versus Replicas . . . 196
4.38.3 Coherence of PVCs . . . 196
4.38.4 Local storage, remote storage, and database size . . . 197
4.39 Examples . . . 197
4.40 Benchmarking . . . 198
4.40.1 pgbench . . . 199
4.40.2 fio . . . 199
4.41 Commercial support . . . 202
4.42 Frequently Asked Questions (FAQ) . . . 202
4.42.1 Running PostgreSQL in Kubernetes . . . 202
4.42.2 High availability . . . 204
4.43 API Reference . . . 204
4.43.1 AffinityConfiguration . . . 206
4.43.2 AzureCredentials . . . 207
4.43.3 Backup . . . 208
4.43.4 BackupConfiguration . . . 208
4.43.5 BackupList . . . 208
4.43.6 BackupSource . . . 209
4.43.7 BackupSpec . . . 209
4.43.8 BackupStatus . . . 209
4.43.9 BarmanCredentials . . . 210
4.43.10 BarmanObjectStoreConfiguration . . . 210
4.43.11 BootstrapConfiguration . . . 211
4.43.12 BootstrapInitDB . . . 211
4.43.13 BootstrapPgBaseBackup . . . 212

4.43.14 BootstrapRecovery . . . 213
4.43.15 CertificatesConfiguration . . . 213
4.43.16 CertificatesStatus . . . 214
4.43.17 Cluster . . . 214
4.43.18 ClusterList . . . 215
4.43.19 ClusterSpec . . . 215
4.43.20 ClusterStatus . . . 217
4.43.21 ConfigMapKeySelector . . . 219
4.43.22 ConfigMapResourceVersion . . . 219
4.43.23 DataBackupConfiguration . . . 219
4.43.24 EmbeddedObjectMetadata . . . 220
4.43.25 ExternalCluster . . . 220
4.43.26 GoogleCredentials . . . 220
4.43.27 Import . . . 221
4.43.28 ImportSource . . . 221
4.43.29 InstanceID . . . 221
4.43.30 InstanceReportedState . . . 221
4.43.31 LDAPBindAsAuth . . . 222
4.43.32 LDAPBindSearchAuth . . . 222
4.43.33 LDAPConfig . . . 222
4.43.34 LocalObjectReference . . . 222
4.43.35 Metadata . . . 223
4.43.36 MonitoringConfiguration . . . 223
4.43.37 NodeMaintenanceWindow . . . 223
4.43.38 PgBouncerIntegrationStatus . . . 224
4.43.39 PgBouncerSecrets . . . 224
4.43.40 PgBouncerSpec . . . 224
4.43.41 PodTemplateSpec . . . 225
4.43.42 Pooler . . . 225
4.43.43 PoolerIntegrations . . . 225
4.43.44 PoolerList . . . 225
4.43.45 PoolerSecrets . . . 226
4.43.46 PoolerSpec . . . 226
4.43.47 PoolerStatus . . . 226
4.43.48 PostInitApplicationSQLRefs . . . 226
4.43.49 PostgresConfiguration . . . 227
4.43.50 RecoveryTarget . . . 227
4.43.51 ReplicaClusterConfiguration . . . 228
4.43.52 ReplicationSlotsConfiguration . . . 228
4.43.53 ReplicationSlotsHAConfiguration . . . 228
4.43.54 RollingUpdateStatus . . . 229
4.43.55 S3Credentials . . . 229
4.43.56 ScheduledBackup . . . 229
4.43.57 ScheduledBackupList . . . 230
4.43.58 ScheduledBackupSpec . . . 230
4.43.59 ScheduledBackupStatus . . . 231
4.43.60 SecretKeySelector . . . 231
4.43.61 SecretVersion . . . 231
4.43.62 SecretsResourceVersion . . . 231
4.43.63 ServiceAccountTemplate . . . 232
4.43.64 StorageConfiguration . . . 232
4.43.65 SyncReplicaElectionConstraints . . . 232
4.43.66 Topology . . . 233
4.43.67 WalBackupConfiguration . . . 233

4.44 Supported releases . . . 233
4.44.1 Support policy . . . 234
4.44.2 Naming scheme . . . 234
4.44.3 Support status of CloudNativePG releases . . . 235
4.44.4 Upcoming releases . . . 235
4.44.5 Old releases . . . 235
4.44.6 What we mean by support . . . 236
4.45 Release notes . . . 236
4.46 Release notes for CloudNativePG 1.19 . . . 237
4.46.1 Version 1.19.1 . . . 237
4.46.2 Version 1.19.0 . . . 238
4.47 Release notes for CloudNativePG 1.18 . . . 239
4.47.1 Version 1.18.3 . . . 239
4.47.2 Version 1.18.2 . . . 240
4.47.3 Version 1.18.1 . . . 240
4.47.4 Version 1.18.0 . . . 242
4.48 Release notes for 1.14.0 and earlier . . . 242
4.48.1 Version 1.14.0 . . . 243
4.48.2 Version 1.13.0 . . . 244
4.48.3 Version 1.12.0 . . . 245
4.48.4 Version 1.11.0 . . . 246
4.48.5 Version 1.10.0 . . . 247
4.48.6 Version 1.9.2 . . . 248
4.48.7 Version 1.9.1 . . . 248
4.48.8 Version 1.9.0 . . . 249
4.48.9 Version 1.8.0 . . . 249
4.48.10 Version 1.7.1 . . . 250
4.48.11 Version 1.7.0 . . . 250
4.48.12 Version 1.6.0 . . . 251
4.48.13 Version 1.5.1 . . . 252
4.48.14 Version 1.5.0 . . . 252
4.48.15 Version 1.4.0 . . . 253
4.48.16 Version 1.3.0 . . . 253
4.48.17 Version 1.2.1 . . . 254
4.48.18 Version 1.2.0 . . . 254
4.48.19 Version 1.1.0 . . . 254
4.48.20 Version 1.0.0 . . . 255
4.49 Release notes for CloudNativePG 1.17 . . . 255
4.49.1 Version 1.17.5 . . . 255
4.49.2 Version 1.17.4 . . . 256
4.49.3 Version 1.17.3 . . . 257
4.49.4 Version 1.17.2 . . . 258
4.49.5 Version 1.17.1 . . . 258
4.49.6 Version 1.17.0 . . . 259
4.50 Release notes for CloudNativePG 1.16 . . . 259
4.50.1 Version 1.16.5 . . . 259
4.50.2 Version 1.16.4 . . . 260
4.50.3 Version 1.16.3 . . . 261
4.50.4 Version 1.16.2 . . . 261
4.50.5 Version 1.16.1 . . . 262
4.50.6 Version 1.16.0 . . . 262
4.51 Release notes for CloudNativePG 1.15 . . . 264
4.51.1 Version 1.15.5 . . . 264
4.51.2 Version 1.15.4 . . . 264

4.51.3 Version 1.15.3 265
4.51.4 Version 1.15.2 265
4.51.5 Version 1.15.1 266
4.51.6 Version 1.15.0 267

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.

CHAPTER

ONE

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.

CHAPTER

TWO

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 .

> **Warning:** CloudNativePG requires that all nodes in a Kubernetes cluster have the same CPU architecture, thus a hybrid CPU architecture Kubernetes cluster is not supported.

CHAPTER

THREE

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` : `pg_audit` , `auto_explain` , and `pg_stat_statements`
- Support for Local Persistent Volumes with PVC templates
- Reuse of Persistent Volumes storage in Pods
- Separate volume for WAL files
- 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 backup to an object store (AWS S3 and S3-compatible, Azure Blob Storage, and Google Cloud Storage)
- Backup from a standby
- Backup retention policies (based on recovery window)
- Full recovery and Point-In-Time recovery from an existing backup in an object store
- Offline import of existing PostgreSQL databases, including major upgrades of PostgreSQL
- 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
- Support for Synchronous Replicas

- Support for HA physical replication slots at cluster level
- 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`
- Fencing of an entire PostgreSQL cluster, or a subset of the instances
- Simple bind and search+bind LDAP client authentication
- Multi-arch format container images
- Postgres cluster hibernation

CHAPTER

FOUR

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.*

4.1 Before You Start

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

4.1.1 Kubernetes terminology

Resource	Description
Node selection through `nodeSelector`<Node selection through `nodeSelector>`	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 Security Policies	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.
The `microservice type<The *microservice* type>`	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.
Defining an operator 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.
Configuration via a 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 Claim	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.

continues on next page

Table 1 – continued from previous page

Resource	Description
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.
Role Based Access Control (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.
Override of operand images through the 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*.
Directly using the operator manifest	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.
Using the `cnpg plugin for *kubectl*`<*Using the `cnpg* plugin for *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.

4.1.2 PostgreSQL terminology

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

continues on next page

Table 2 – continued from previous page

Resource	Description
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*).

4.1.3 Cloud terminology

Resource	Description
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.

4.1.4 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.

4.2 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.

4.2.1 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.

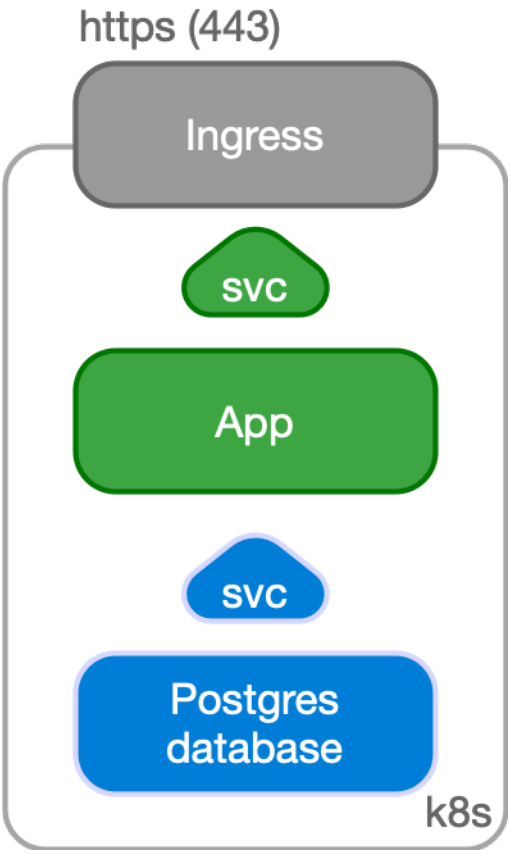

Fig. 1: Application and Database inside Kubernetes

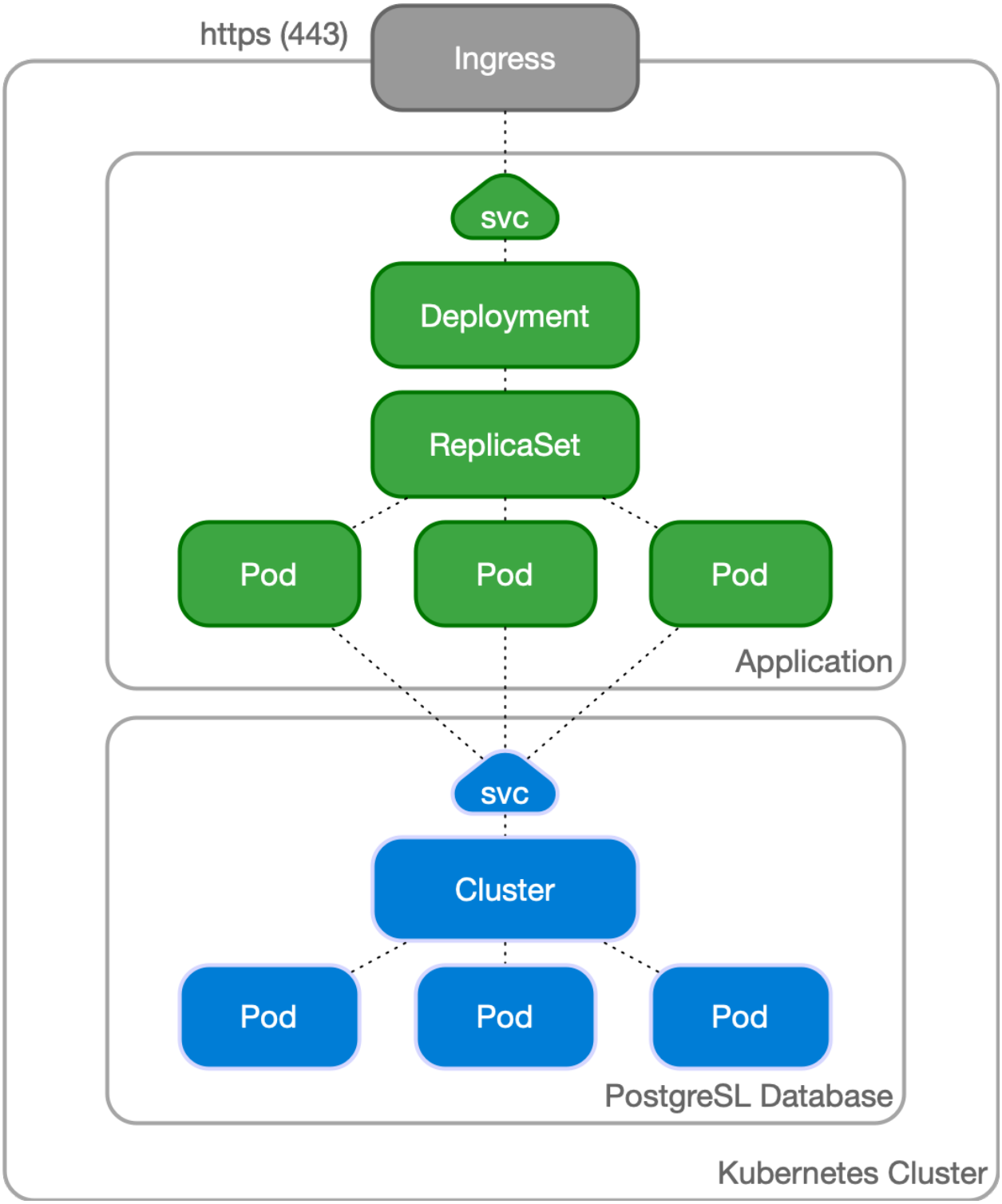

Fig. 2: Close-up view of application and database inside Kubernetes

4.2.2 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.

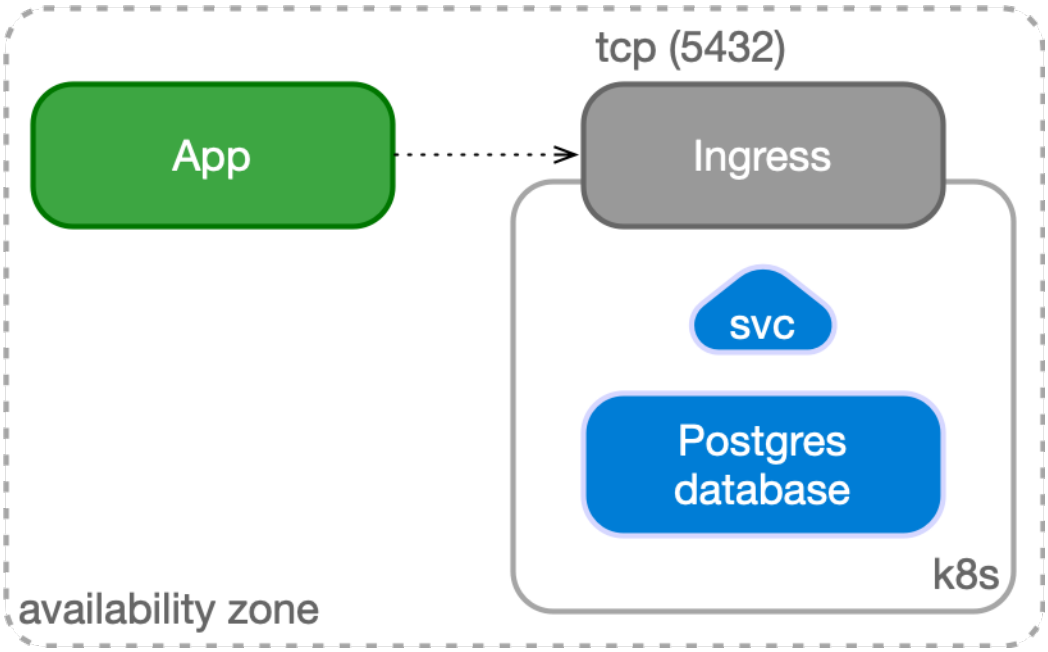

Fig. 3: Application outside Kubernetes

4.3 Architecture

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

4.3.1 Synchronizing the state

PostgreSQL is a database management system and, as such, it needs to be treated as a **statefulworkload** 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 **physicalreplication** capabilities based on **WriteAheadLog(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.

4.3.2 Kubernetes architecture

Kubernetes natively provides the possibility to span separate physical locations - also known as data centers, failure zones, or more frequently **availabilityzones** - 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 **eachdatacenterisactiveatanytime** 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.

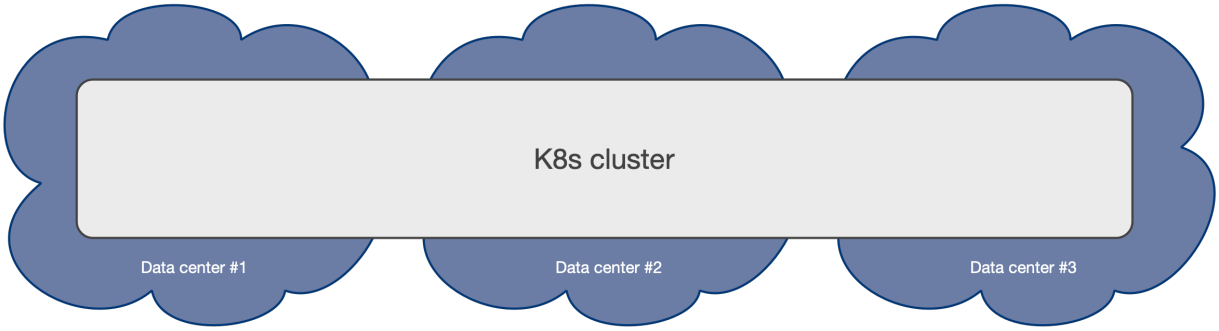

Fig. 4: Kubernetes cluster spanning over 3 independent data centers

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.

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)datacenters** 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.

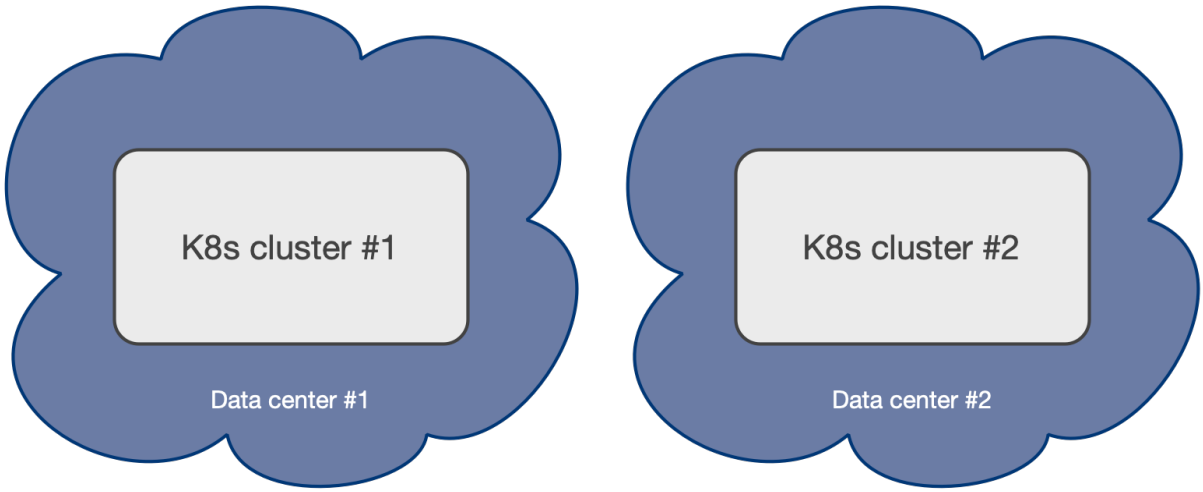

Fig. 5: Example of a Kubernetes architecture with only 2 data centers

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.

4.3.3 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.

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.

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:

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.

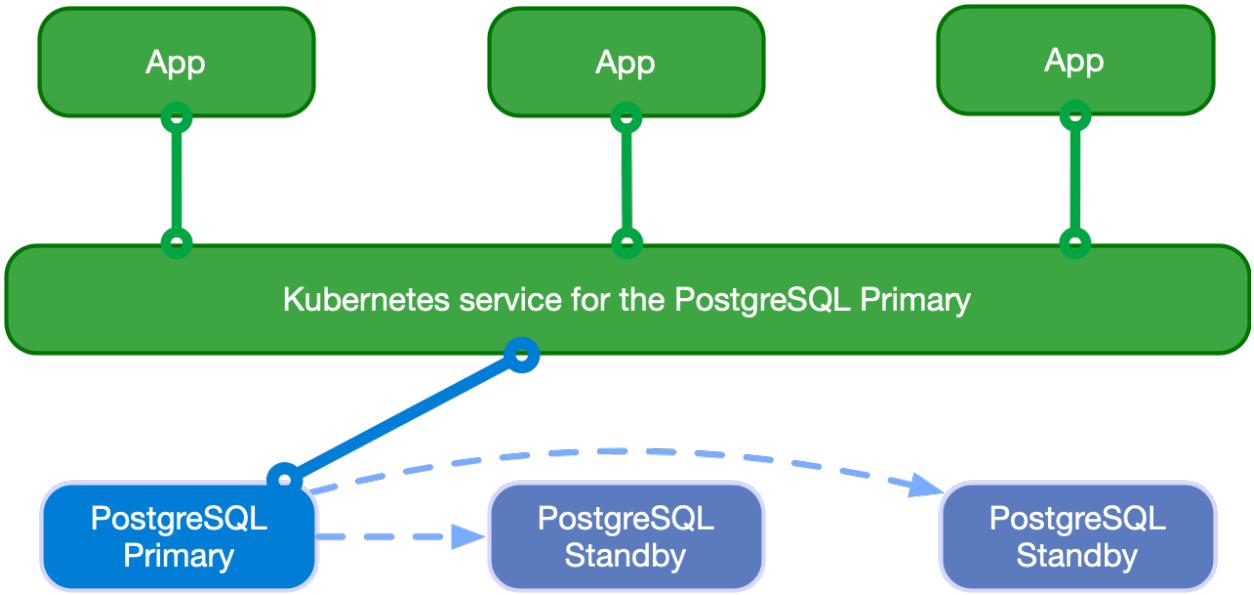

Fig. 6: Applications writing to the single primary

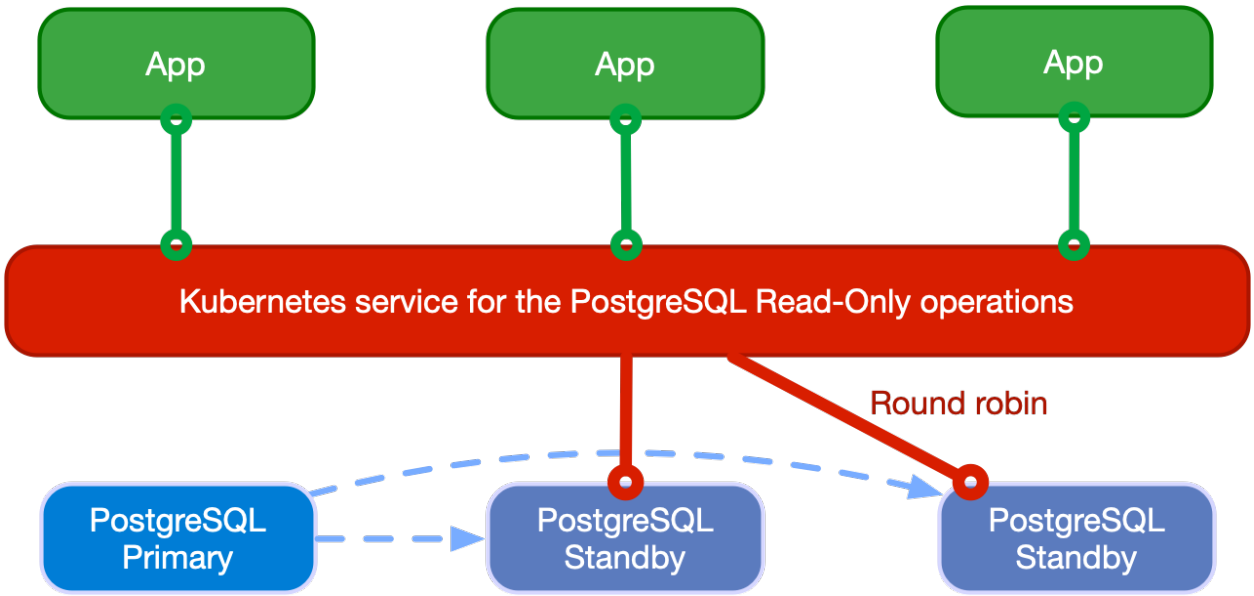

Fig. 7: Applications reading from hot standby replicas in round robin

4.3.4 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 **recoverypointobjectives** (RPO) by storing PostgreSQL backup data in multiple locations, regions and possibly using different providers (Disaster Recovery)
- reduce global **recoverytimeobjectives** (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.

A replica cluster can have the same architecture of the primary cluster. In place of the primary instance, a replica cluster has a **designatedprimary** 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.

4.4 Installation and upgrades

4.4.1 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 -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-1.19/releases/
↪cnpg-1.19.1.yaml
```

You can verify that with:

```
kubectl get deploy -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-namespaces "specific-namespace" \
  > cnpg_for_specific_namespace.yaml
```

Please refer to Using the `cnpg plugin for *kubectl`<Using the `cnpg* plugin for *kubectl>*` 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 *Some common issues* . 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 for this minor release with:

```
curl -sSfL \
  https://raw.githubusercontent.com/cloudnative-pg/artifacts/release-1.19/manifests/
↪operator-manifest.yaml | \
  kubectl apply -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 *Using the Helm Chart* .

4.4.2 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).

4.4.3 Upgrades

Important: Please carefully read the *Release notes for CloudNativePG 1.19*

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` .

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.

Important: This feature requires that all pods (operators and operands) run on the same platform/architecture (for example, all *linux/amd64*).

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, **westronglyencourageuserstoupgradetothelatestversionoftheoperator** , 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.

Important: In 2022, EDB plans an LTS release for CloudNativePG in environments where frequent online updates are not possible.

The *Release notes for CloudNativePG 1.19* 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.

4.5 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/>`__ .

4.5.1 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
```

4.5.2 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.

4.5.3 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 cluster-example.yaml sample file defines a simple `Cluster` using the default storage class to allocate disk space:

```
#  Example of PostgreSQL cluster

apiVersion: postgresql.cnpg.io/v1
```

(continues on next page)

(continued from previous page)

```yaml
kind: Cluster
metadata:
  name: cluster-example
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

  # Require 1Gi of space
  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:

```yaml
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.

4.5.4 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 example* , including the Alert Manager , and a *Part 4 - Monitor clusters with Prometheus and 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 *Using the Helm Chart* 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.

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

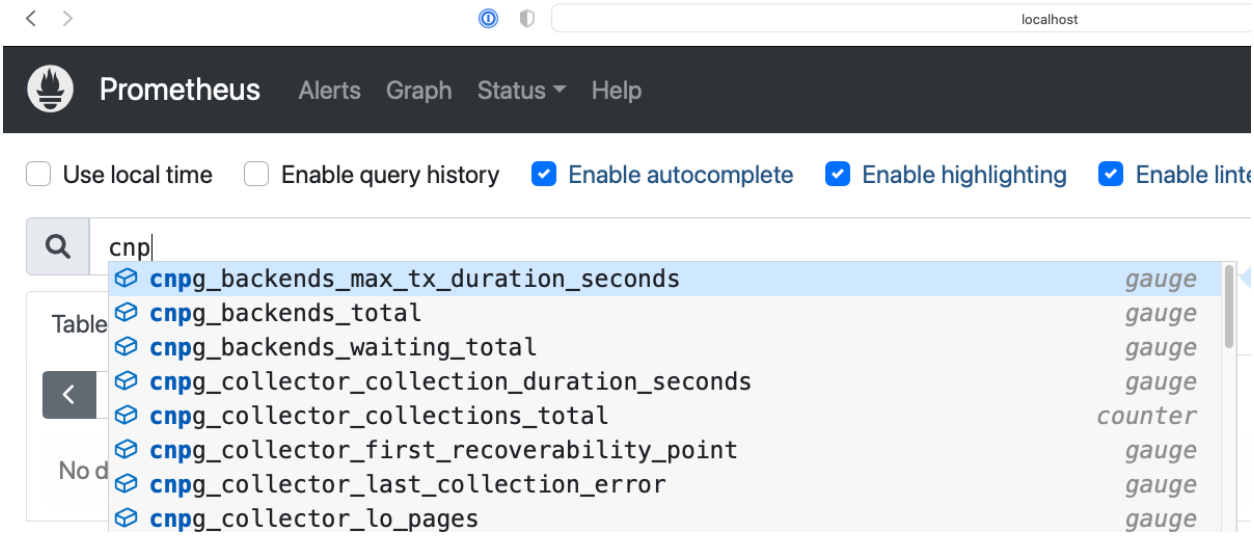

Fig. 8: local prometheus

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

We can now install our sample Grafana dashboard:

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

Which will be picked up by the Grafana page in a few seconds. You should now see the `CloudNativePG` dashboard.

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

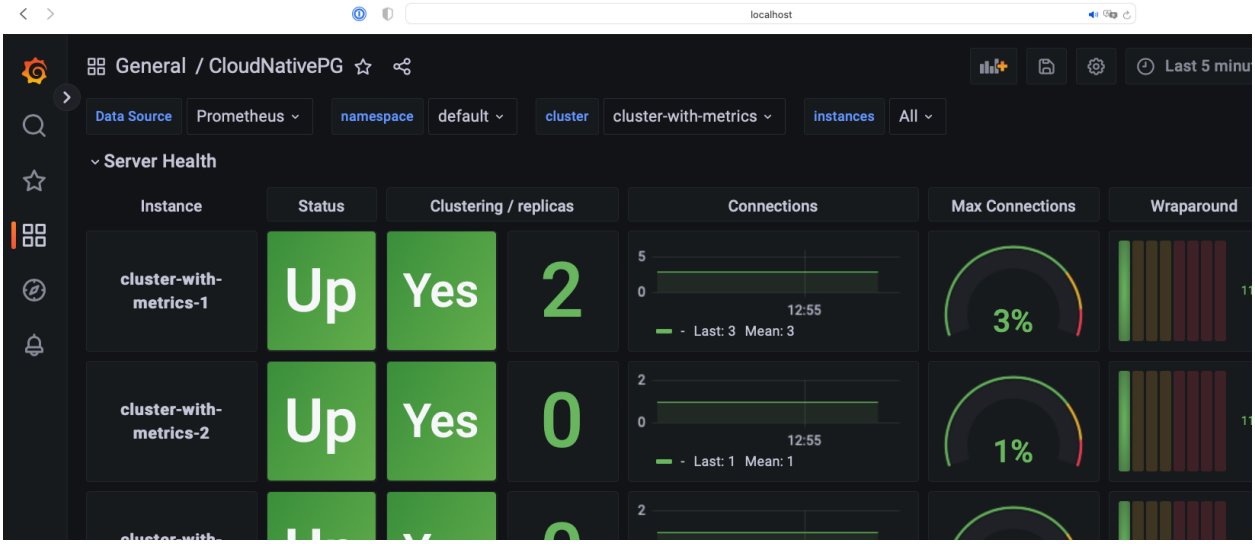

Fig. 9: local grafana

4.6 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 (`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!!!

4.6.1 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 an existing cluster via a backup object store, and 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.

for more information.

4.6.2 The `externalClusters` section

The `externalClusters` section allows you to define one or more PostgreSQL clusters that are somehow related to the current one. While in the future this section will enable more complex scenarios, it is currently intended to define a cross-region PostgreSQL cluster based on physical replication, and spanning over different Kubernetes clusters or even traditional VM/bare-metal environments.

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 **recoveryobjectstore** , which is a Barman Cloud compatible object store that contains the backup files of the source cluster - that is, base backups and WAL archives.

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.

for more information.

4.6.3 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

  superuserSecret:
    name: superuser-secret

  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. set a password for the postgres *superuser* from the secret named superuser-secret
3. create an *unprivileged* user named app
4. 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)
5. 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 passwords, store them as secrets, and use them in the PostgreSQL cluster - as described in the above example.

The supplied secrets 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
```

(continues on next page)

(continued from previous page)

```
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: Future implementations of the operator might allow you to create additional users in a declarative configuration fashion.

The `postgres` superuser and the `postgres` database are supposed to be used only by the operator to configure the cluster.

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.

Important: For now, changes to the name of the superuser secret are not applied to the cluster.

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
```

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 ROLE angus
        - CREATE ROLE malcolm
  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.

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.

4.6.4 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 a *recovery object store* (`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`)

The `recovery` bootstrap mode lets you create a new cluster from an existing backup, namely a *recovery object store*.

There are two ways to achieve this result in CloudNativePG:

- using a recovery object store, that is a backup of another cluster created by Barman Cloud and defined via the `barmanObjectStore` option in the `externalClusters` section (*recommended*)
- using an existing `Backup` object in the same namespace (this was the only option available before version 1.8.0).

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, the cluster can also be started in replica mode. Also, make sure that the PostgreSQL configuration (`.spec.postgresql.parameters`) of the recovered cluster is compatible, from a physical replication standpoint, with the original one.

Note: You can find more information about backup and recovery of a running cluster in the *Backup and Recovery* .

Recovery from an object store

You can recover from a backup created by Barman Cloud and stored on a supported object storage. Once you have defined the external cluster, including all the required configuration in the `barmanObjectStore` section, you need to reference it in the `.spec.recovery.source` option. The following 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 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 clusters definition).

Note: In the above example we are taking 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 certainly make a difference **when** (not if) you'll need it.

Recovery from a `Backup` object

In case a Backup resource is already available in the namespace in which the cluster should be created, you can specify its name through `.spec.bootstrap.recovery.backup.name` , as in the following example:

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

  superuserSecret:
    name: superuser-secret

  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.

Additional considerations

Whether you recover from a recovery object store or an existing `Backup` resource, the following considerations apply:

- The application database name and the application database user are preserved from the backup that is being restored. The operator does not currently attempt to back up the underlying secrets, as this is part of the usual maintenance activity of the Kubernetes cluster itself.
- In case you don't supply any `superuserSecret` , a new one is automatically generated with a secure and random password. The secret is then used to reset the password for the `postgres` user of the cluster.
- By default, the recovery will continue up to the latest available WAL on the default target timeline (`current` for PostgreSQL up to 11, `latest` for version 12 and above). You can optionally specify a `recoveryTarget` to perform a point in time recovery (see the

  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, we can ask PostgreSQL to stop replaying WALs at any given point in time, after having extracted a base backup. PostgreSQL uses this technique to achieve *point-in-time* recovery (PITR).

Note: PITR is available from recovery object stores as well as *Backup* objects.

The operator will generate the configuration parameters required for this feature to work in case a recovery target is specified, like in the following example that uses a recovery object stored in Azure and a timestamp based goal:

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

  storage:
    size: 5Gi

  bootstrap:
    recovery:
      source: clusterBackup
      recoveryTarget:
        targetTime: "2020-11-26 15:22:00.00000+00"

  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
```

You might have noticed that in the above example you only had to specify the `targetTime` in the form of a timestamp, without having to worry about specifying 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 will use that backup as base for the recovery.

Important: You need to make sure that such a backup exists and is accessible.

If the backup ID is not specified, the operator will automatically detect 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.

Here are the recovery target criteria you can use:

targetTime
: [time stamp up to which recovery will proceed, expressed in] RFC 3339 format (the precise stopping point is also influenced by

the `exclusive` option)

targetXID : transaction ID up to which recovery will proceed (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 will be 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 will proceed

targetLSN : LSN of the write-ahead log location up to which recovery will proceed (the precise stopping point is also influenced by the `exclusive` option)

targetImmediate : recovery should end as soon as a consistent state is reached - i.e. as early as possible. When restoring from an online backup, this means the point where taking the backup ended

Important: While the operator is able to automatically retrieve the closest backup when either *targetTime* or *targetLSN* is specified, this is not possible for the remaining targets: *targetName* , *targetXID* , and *targetImmediate* . In such cases, it is important to specify *backupID* , unless you are OK with the last available backup in the catalog.

The example below uses a `targetName` based recovery target:

```yaml
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 above in each `recoveryTarget` configuration.

Additionally, you can specify `targetTLI` force recovery to a specific timeline.

By default, the previous parameters are considered to be exclusive, stopping just before the recovery target. You can request inclusive behavior, stopping right after the recovery target, setting the `exclusive` parameter to `false` like in the following example relying on a blob container in Azure:

```yaml
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: false

  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, we can configure the application database name and credentials with additional configuration. To update application database credentials, we can generate our own passwords, store them as secrets, and update the database use the secrets. Or we can also let the operator generate a secret with randomly secure password for use. Please reference the Bootstrap an empty cluster (`initdb )<Bootstrap an empty cluster (*initdb* )>`

section for more information about secrets.

The following example configure the application database `app` with owner `app` , and supplied secret `app-secret` .

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  bootstrap:
    recovery:
      database: app
      owner: app
      secret:
```

(continues on next page)

(continued from previous page)

```
      name: app-secret
    [...]
```

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.

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 **binarycompatible** 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
- source must not have any tablespace defined (see *Current limitations* below)
- 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 Bootstrap from a live cluster (`pg_basebackup )<Bootstrap from a live cluster (*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 15.2 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:15.2

  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 15.2).

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 15.2 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:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:15.2

  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␣

(continues on next page)

(continued from previous page)

```
↪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

Missing tablespace support

CloudNativePG does not currently include full declarative management of PostgreSQL global objects, namely roles, databases, and tablespaces. While roles and databases are copied from the source instance to the target cluster, tablespaces require a capability that this version of CloudNativePG is missing: definition and management of additional persistent volumes. When dealing with base backup and tablespaces, PostgreSQL itself requires that the exact mount points in the source instance must also exist in the target instance, in our case, the pods in Kubernetes that CloudNativePG manages. For this reason, you cannot directly migrate in CloudNativePG a PostgreSQL instance that takes advantage of tablespaces (you first need to remove them from the source or, if your organization requires this feature, contact EDB to prioritize it).

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. Feel free to contact EDB for assistance.

4.7 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".

4.7.1 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:

- **microserviceapproach** : the destination cluster is designed to host a single application database owned by the specified application user, as recommended by the CloudNativePG project
- **monolithapproach** : 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.

4.7.2 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:
  name: cluster-microservice
spec:
  instances: 3

  bootstrap:
    initdb:
```

(continues on next page)

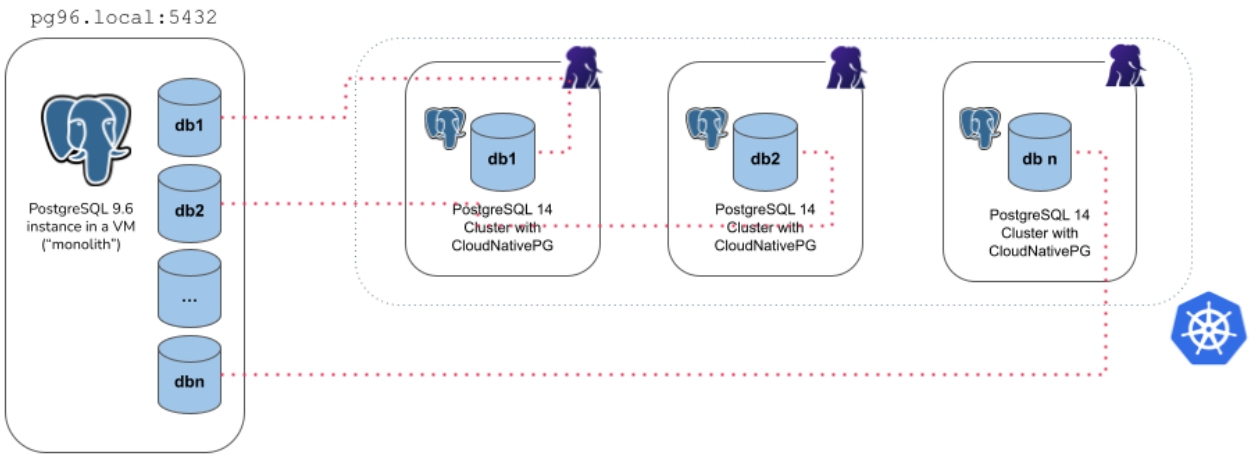

Fig. 10: Example of microservice import type

(continued from previous page)

```
    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`

4.7.3 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

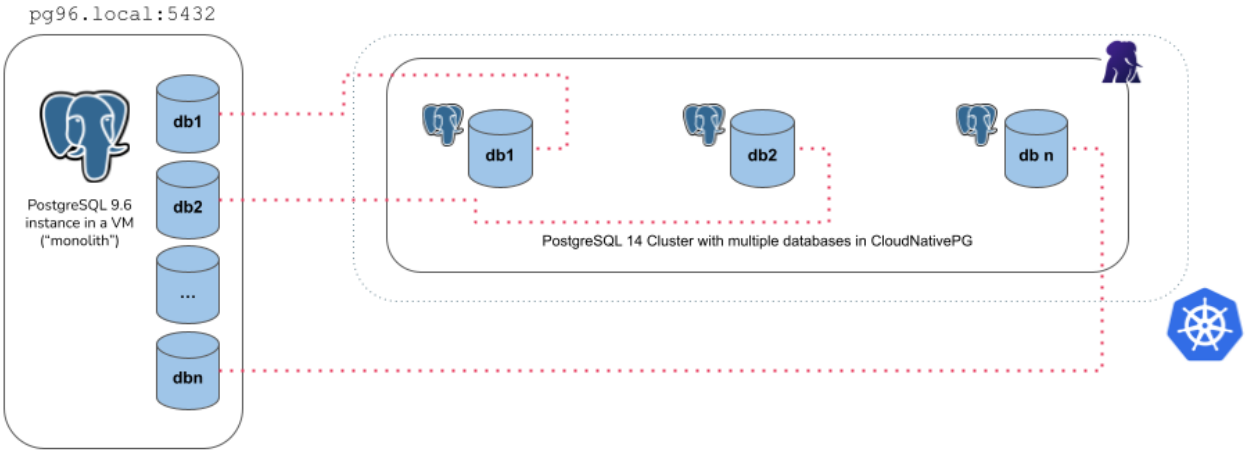

Fig. 11: Example of monolith import type

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.

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

4.8 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.

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

4.8.1 Code

Source code of CloudNativePG is *systematically scanned* for static analysis purposes, including **securityproblems** , 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.

4.8.2 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 EDB 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)

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

4.8.3 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* , and *ScheduledBackup* 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 .

`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

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 *Restricting Pod access using 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.

> **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.

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 `postgres` superuser and the database owner.

As far as encryption of password 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 *Username/Password authentication*

section in the PostgreSQL documentation for details.

You can disable management of the `postgres` user password via secrets by setting `enableSuperuserAccess` to `false` .

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 **physicalasyncstreamingreplication** with the current primary instance, with a special user called `streaming_replica` . The connection between nodes is **encrypted** and authentication is via **TLSclientcertificates** (please refer to the *Client TLS/SSL Connections* page for details).

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.

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

4.9 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 **Postgresinstancemanager** .

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 liveness and readiness probes.

4.9.1 Liveness and readiness probes

The liveness probe relies 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.

> The two probes will report a failure if the probe command fails 3 times with a 10 seconds interval between each check.

For now, the operator doesn't configure a `startupProbe` on the Pods, since startup probes have been introduced only in Kubernetes 1.17.

The liveness probe is used to detect 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, which is used to prevent an instance with a long startup time from being restarted.

The number of seconds after the Pod has started before the liveness probe starts working is expressed in the `.spec.startDelay` parameter, which defaults to 30 seconds. The correct value for your cluster is related to the time needed by PostgreSQL to start.

If `.spec.startDelay` is too low, the liveness probe will start working before the PostgreSQL startup, and the Pod could be restarted inappropriately.

4.9.2 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.stopDelay` , expressed in seconds, is the amount of time given to PostgreSQL to shut down. The value defaults to 30 seconds.

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 half of the time set in `.spec.stopDelay` .
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 half of the time set in `.spec.stopDelay` to complete the operation and then forcibly shut down.

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. During this time frame, the primary instance does not accept connections. The value defaults is greater than one year in seconds, big enough to simulate an infinite delay and therefore preserve data durability.

> **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.

4.9.3 Failover

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

4.10 Scheduling

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

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 *Pod affinity and anti-affinity*

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).

4.10.1 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-podaffinity/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:15.2

  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).

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"
```

4.10.2 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.

4.10.3 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.

4.11 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"

  resources:
    requests:
```

(continues on next page)

(continued from previous page)

```
      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.

page from the Kubernetes documentation.

4.12 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 contact EDB for support and assistance.

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

4.12.1 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
```

4.12.2 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.

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` .

4.12.3 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.

4.12.4 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 the support and assistance of EDB engineering team.

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.

4.13 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;
- 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.

4.13.1 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:

- `switchover` : a switchover operation is automatically performed, setting the most aligned replica as the new target primary, and shutting down the former primary pod (default).
- `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.

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.

4.13.2 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.

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 Using the `cnpg plugin for *kubectl*`<*Using the `cnpg* plugin for *kubectl*>` .

4.14 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.

4.14.1 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-levelreplication** , in contrast with *storage-level replication*.

4.14.2 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.

4.14.3 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
```

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.

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.

4.14.4 Synchronous replication

CloudNativePG supports the configuration of **quorum-basedsynchronousstreamingreplication** 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.

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 *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.

4.14.5 Replication slots for High Availability

Replication slots for High Availability

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 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:

- **PrimaryHAslot** : 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.
- **StandbyHAslot** : 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, introduced in CloudNativePG 1.18, can be enabled via configuration. For details, please refer to the *ReplicationSlotsConfiguration* . Here follows a brief description of the main options:

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

`.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 enabled.

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

  storage:
    size: 1Gi
```

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:

```yaml
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:
    highAvailability:
      enabled: true
    updateInterval: 300

  storage:
    size: 1Gi
```

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.

4.15 Backup and Recovery

CloudNativePG natively supports **online/hotbackup** of PostgreSQL clusters through continuous physical backup and WAL archiving. This means that the database is always up (no downtime required) and that you can recover at any point in time from the first available base backup in your system. The latter is normally referred to as "Point In Time Recovery" (PITR).

The operator can orchestrate a continuous backup infrastructure that is based on the *BarmanCredentials* 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* .

4.15.1 Cloud provider support

You can archive the backup files in any service that is supported by the Barman Cloud infrastructure. That is:

- AWS 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.

S3

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:

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

(continues on next page)

(continued from previous page)

```
          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:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
[...]
spec:
  serviceAccountTemplate:
    metadata:
      annotations:
        eks.amazonaws.com/role-arn: arn:[...]
        [...]
```

Other S3-compatible Object Storages providers

In case you're using S3-compatible object storage, like **MinIO** or **LinodeObjectStorage** , 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: "<destination path here>"
      endpointURL: "https://bucket.us-east1.linodeobjects.com"
      s3Credentials:
        [...]
```

In case you're using **DigitalOceanSpaces** , you will have to use the Path-style syntax. In this example, it will use the `bucket` from **DigitalOceanSpaces** in the region `SFO3` .

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "s3://[your-bucket-name]/[your-backup-folder]/"
```

(continues on next page)

(continued from previous page)

```
      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.

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:

```yaml
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
    - 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
```

(continues on next page)

(continued from previous page)

```
        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.

Azure Blob Storage

In order to access your storage account, you will need one of the following combinations of credentials:

- **Connection String**
- **Storageaccountname** and **Storage account access key**
- **Storageaccountname** and **Storage account SAS Token**
- **Storageaccountname** and **Azure AD Workload Identity**

properly configured.

Using **AzureADWorkloadIdentity** , 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 **Storageaccountaccesskey** or **StorageaccountSASToken** , 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 **accountname** , which is also called **storageaccountname** , 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 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:

```
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:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
```

(continues on next page)

(continued from previous page)

```
      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.

4.15.2 On-demand backups

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:
  cluster:
    name: pg-backup
```

The operator will start to orchestrate the cluster to take the required backup 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>
```

(continues on next page)

(continued from previous page)

```
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:
  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.

4.15.3 Scheduled backups

You can also schedule your backups periodically by creating a resource named `ScheduledBackup` . The latter is similar to a `Backup` but with an added field, called `schedule` .

This field is a *cron schedule* specification, which follows the same
format used in Kubernetes CronJobs .

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.

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.

ScheduledBackups can be suspended if needed by setting `.spec.suspend: true`, this will stop any new backup to be scheduled as long as the option is set 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
```

4.15.4 WAL archiving

WAL archiving is enabled as soon as you choose a destination path and you configure your cloud credentials.

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.

4.15.5 Backup from a standby

By default, backups will run on the primary instance of a `Cluster` .

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: **backupfromastandby** .

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.

You can use set backup target to `prefer-standby` as outlined in the example below:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  [...]
spec:
  backup:
    target: "prefer-standby"
```

The `prefer-standby` policy will ensure backups are run on the most up-to-date available secondary instance, falling back to the primary instance if no other instance is available.

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

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

```yaml
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.

4.15.6 Recovery

Cluster restores are not performed "in-place" on an existing cluster. You can use the data uploaded to the object storage to *bootstrap* a new cluster from a previously taken backup. The operator will orchestrate 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, please refer to the Bootstrap from a backup (`recovery )<Bootstrap from a backup (*recovery* )>` .

Important: If you are 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.

Under the hood, the operator will inject an init container in the first instance of the new cluster, and the init container will start 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 completed, the operator starts the Postgres instance in recovery mode: in this phase, PostgreSQL is up, albeit not able to accept connections, and the pod is healthy according to the liveness probe. Through the `restore_command` , PostgreSQL starts fetching WAL files from the archive (you can speed up this phase by setting the `maxParallel` option and enable 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 Point-In-Time-Recovery). Indeed, you can optionally specify a `recoveryTarget` to perform a point in time recovery. If left unspecified, the recovery will continue up to the latest available WAL on the default target timeline (`current` for PostgreSQL up to 11, `latest` for version 12 and above).

Once the recovery is complete, the operator will set the required superuser password into the instance. The new primary instance will start as usual, and the remaining instances will join the cluster as replicas.

The process is transparent for the user and it is managed by the instance manager running in the Pods.

Restoring into a cluster with a backup section

A manifest for a cluster restore may include a `backup` section. This means that the new cluster, after recovery, will start archiving WAL's and taking backups if configured to do so.

For example, the section below could be part of a manifest for a Cluster bootstrapping from Cluster `cluster-example-backup` , and would create a new folder in the storage bucket named `recoveredCluster` where the base backups and WAL's of the recovered cluster would be 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:
```

You should not re-use the exact same `barmanObjectStore` configuration for different clusters. There could 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 will not overwrite a storage bucket that contained information. A cluster that would overwrite existing storage will remain in state *Setting up primary* with Pods in an Error state. The pod logs will show: *ERROR: WAL archive check failed for server recoveredCluster: Expected empty archive*

4.15.7 Retention policies

CloudNativePG can manage the automated deletion of backup files from the backup object store, using **retentionpolicies** 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:
```

(continues on next page)

(continued from previous page)

```
  barmanObjectStore:
    destinationPath: "<destination path here>"
    s3Credentials:
      accessKeyId:
        name: aws-creds
        key: ACCESS_KEY_ID
      secretAccessKey:
        name: aws-creds
        key: ACCESS_SECRET_KEY
  retentionPolicy: "30d"
```

4.15.8 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* and *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 ra
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

4.15.9 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"
```

4.16 PostgreSQL Configuration

Users that are familiar with PostgreSQL are aware of the existence of the following two files to configure an instance:

- `postgresql.conf` : main run-time configuration file of PostgreSQL
- `pg_hba.conf` : clients authentication file

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` and `pg_hba.conf` settings via the `parameters` and the `pg_hba` 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.

A reference for custom settings usage is included in the samples, see
cluster-example-custom.yaml .

4.16.1 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"
# ...
```

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 **globaldefaultparameters** are:

```
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_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
archive_mode = on
full_page_writes = on
hot_standby = true
listen_addresses = *
port = 5432
restart_after_crash = false
ssl = on
```

(continues on next page)

(continued from previous page)

```
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
wal_level = logical
wal_log_hints = on
```

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 preloaded 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`

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 Enabling `auto_explain`<Enabling `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 Enabling `pg_stat_statements`<Enabling `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:
```

(continues on next page)

(continued from previous page)

```
    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 Enabling `pgaudit`<Enabling `pgaudit>` on PostgreSQL clusters. For further information, please refer to the

> Enabling `pgaudit`<Enabling `pgaudit>`

You can enable `pgaudit` by adding to the configuration a parameter that starts with `pgaudit.` as in the following example excerpt:

```
#
postgresql:
  parameters:
    pgaudit.log: "all, -misc"
    pgaudit.log_catalog: "off"
    pgaudit.log_parameter: "on"
    pgaudit.log_relation: "on"
#
```

4.16.2 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.

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)
```

Refer to the PostgreSQL documentation for *more information on `pg_hba.conf* <https://www.postgresql.org/docs/current/auth-pg-hba-conf.html>`__ .

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
```

(continues on next page)

(continued from previous page)

```
        key: data
      searchAttribute: uid
```

4.16.3 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.

4.16.4 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= **** **)
```

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
```

4.16.5 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_ciphers
- ssl_crl_file
- ssl_dh_params_file
- ssl_ecdh_curve
- ssl_key_file
- ssl_max_protocol_version
- ssl_min_protocol_version
- 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`
- `wal_level`
- `wal_log_hints`

4.17 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.

4.17.1 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

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.

4.17.2 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
```

4.17.3 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.

```
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
```

4.17.4 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:

```
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:

```
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.

4.17.5 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 programs command line, with␣
↪arguments separated by NUL 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.
```

(continues on next page)

(continued from previous page)

```
- `/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 – bash ``` Once inside execute:

```
curl localhost:6060/debug/pprof/
```

4.18 Instance Pod Configuration

4.18.1 Projected Volumes

CloudNativePG supports to mount custom files inside the Postgres pods through `.spec.projectedVolumeTemplate` , this is useful for several Postgres features and extensions that require additional data files. In CloudNativePG, `.spec.projectedVolumeTemplate` field is a

> *Projected Volumes* template in kubernetes, which allows user to mount

arbitrary data under `/projected` folder in Postgres pods.

Here is a simple example about 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 will be mounted as files into path `/projected/certificate/tls.crt` and `/projected/certificate/tls.key` in Postgres pod.

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

(continues on next page)
```

(continued from previous page)

```
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 using projected volume template to mount Secret and Configmap in the cluster-example-projected-volume.yaml deployment manifest.

4.18.2 Environment variables

Some system behavior can be customized using environment variables. One example is the `LDAPCONF` variable, which may 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 the user to set custom environment variables via the `env` and the `envFrom` stanza of the cluster specification.

The following is a definition of 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:
```

(continues on next page)

(continued from previous page)

```
  instances: 3

  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 will trigger a rolling update of the PostgreSQL Pods.

If the `env` or the `envFrom` section refers to a Secret or a ConfigMap, the operator will not detect any changes in them and will not trigger a rollout. The Kubelet use the same behavior with Pods, and the user is supposed to trigger the Pod rollout manually.

4.19 Storage

Storageisthemostcriticalcomponentinadatabaseworkload . Storage should be always 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: Kubernetes has its own specificities, when it comes to dynamically provisioned storage. These include *storage classes*, *persistent volumes*, and *persistent volume claims*. You need to own these concepts, on top of all the valuable knowledge you have 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 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 are using is recommended for database workloads. Our advice is to clearly set performance expectations by first benchmarking

the storage using tools such as Benchmarking the storage with fio , and then the database using Benchmarking the database with pgbench .

4.19.1 Benchmarking CloudNativePG

EDB maintains cnp-bench , an open source set of guidelines and Helm charts for benchmarking CloudNativePG in a controlled Kubernetes environment, before deploying the database in production.

Briefly, `cnp-bench` is designed to operate 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 the default benchmarking tool distributed along with PostgreSQL: `pgbench`

Important: Measuring both the storage and database performance is an activity that must be done **before the database goes in production**. However, such results are extremely valuable not only in the planning phase (e.g., capacity planning), but also in the production lifecycle, especially in emergency situations (when we don't have the luxury anymore to run this kind of tests). Databases indeed change and evolve over time, so does the distribution of data, potentially affecting performance: knowing the theoretical maximum throughput of sequential reads or writes will turn out to be extremely useful in those situations. Especially in shared-nothing contexts, where results do not vary due to the influence of external workloads. **Know your system, benchmark it.**

4.19.2 Persistent Volume Claim

The operator creates a persistent volume claim (PVC) for each PostgreSQL instance, with the goal to store the `PGDATA` , and then mounts it into each Pod.

Additionally, it supports the creation of clusters with a separate PVC on which to store PostgreSQL Write-Ahead Log (WAL), as explained in the

Volume for WAL below.

In CloudNativePG, the volumes attached to a single PostgreSQL instance are defined as **PVCgroup** .

4.19.3 Configuration via a storage class

The easier way to configure the storage for a PostgreSQL class is to just 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 will be 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
```

Important: CloudNativePG has been designed to be storage class agnostic. As usual, our recommendation is to properly benchmark the storage class in a controlled environment, before hitting production.

4.19.4 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
```

4.19.5 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` (historically known as `pg_xlog` in PostgreSQL), which contains the log of transactional changes occurred in the database, in the form of segment files.

Note: Normally, each segment is 16 MB in size, but the size can be configured through the *walSegmentSize* option, applied at cluster initialization time, as described in Bootstrap an empty cluster (`initdb )<Bootstrap an empty cluster (*initdb* )>` .

While in most cases, having `pg_wal` on the same volume where `PGDATA` resides is fine, there are a few benefits from having WALs stored in a separate volume:

- **I/Operformance** : by storing WAL files on different storage than `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
- **morereliability** : by reserving dedicated disk space to WAL files, you can always be sure that exhaustion of space on the `PGDATA` volume will never interfere with WAL writing, ensuring that your PostgreSQL primary is correctly shut down.
- **finercontrol** : you can define the amount of space dedicated to both `PGDATA` and `pg_wal` , fine tune WAL configuration and checkpoints, even use a different storage class for cost optimization
- **betterI/Omonitoring** : you can constantly monitor the load and disk usage on both `PGDATA` and `pg_wal` , and set proper alerts that notify you in case, for example, `PGDATA` requires resizing

from the official PostgreSQL documentation for more information.

You can add a separate volume for WAL through the `.spec.walStorage` option, which 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* is not supported: once added, a separate volume for WALs cannot be removed from an existing Postgres cluster.

4.19.6 Volume expansion

Kubernetes exposes an API allowing expanding PVCs

that is enabled by default but 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 will apply 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 will need to 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

At the moment, Azure is not able to resize the PVC's volume without restarting the pod . 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 follow the workaround below to manually resize the volume in AKS.

Workaround for volume expansion on AKS

You can manually resize a PVC on AKS by following these procedures. As an example, let's suppose you have a cluster with 3 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 only be expanded while in "unattached" state, as described in the docs . This means, that to resize a disk used by a PostgreSQL cluster, you will need to perform a manual rollout, first cordoning the node that hosts the Pod using the PVC bound to the disk. This will prevent the Operator to recreate the Pod and immediately reattach it to its PVC before the background disk resizing has been completed.

First step is to edit the cluster definition applying the new size, let's say "2Gi", as follows:

```
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, we 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 recreated correctly and get in 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
```

Now verify the PVC expansion by running the following command, which should return "2Gi" as configured:

```
kubectl get pvc cluster-example-3 -o=jsonpath={.status.capacity.storage}
```

So, you can repeat these steps for the remaining Pods.

Important: Please leave the resizing of the disk associated with the primary instance as last disk, after promoting through a switchover a new resized Pod, using *kubectl cnpg promote* (e.g. *kubectl cnpg promote cluster-example 3* to promote *cluster-example-3* to primary).

Recreating storage

If the storage class does not support volume expansion, you can still regenerate your cluster on different PVCs, by allocating 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
```

In order to move the entire cluster to a different storage area, you need to recreate all the PVCs and all the Pods. Let's 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 recreate the cluster using different PVCs, you can edit the cluster definition to disable `resizeInUseVolumes` , and then recreate every instance in a different PVC.

As an example, to recreate the storage for `cluster-example-3` you can:

```
$ kubectl delete pvc/cluster-example-3 pod/cluster-example-3
```

Important: In case you have created a dedicated WAL volume, both PVCs will have to be deleted during this process. Additionally, the same procedure applies in case you want to regenerate the WAL volume PVC, which can be done by disabling *resizeInUseVolumes* also for the *.spec.walStorage* section.

For example (in case 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 will orchestrate the creation of 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
```

4.20 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, please refer to the Kubernetes documentation on *Labels and annotations* and *Labels and annotations* .

In short:

- an annotation is used to assign additional non-identifying information to resources with the goal to facilitate integration with external tools
- a label is used to group objects and query them through Kubernetes' native selector capability

You can select one or more labels and/or annotations you will use in your CloudNativePG deployments. Then you need to configure the operator so that when you define these labels and/or annotations in a cluster's metadata, they are automatically inherited by all resources created by it (including pods).

Note: Label and annotation inheritance is the technique adopted by CloudNativePG in lieu of alternative approaches such as pod templates.

4.20.1 Pre-requisites

By default, no label or annotation defined in the cluster's metadata is inherited by the associated resources. In order to enable label/annotation inheritance, you need to follow the instructions provided in the *Operator configuration* section.

Below we will continue on that example and limit 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. Remember that you can also use wildcards in naming and adopt strategies like *mycompany/** for all labels or annotations starting with *mycompany/* to be inherited.

4.20.2 Defining cluster's metadata

When defining the cluster, **before** any resource is deployed, you can properly 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 have been correctly set in the pods with:

```
kubectl get pods --show-labels
```

4.20.3 Current limitations

Currently, CloudNativePG does not automatically propagate labels or annotations deletions. Therefore, when an annotation or label is removed from a Cluster, which was previously propagated to the underlying pods, the operator will not automatically remove it on the associated resources.

4.21 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 .

4.21.1 Monitoring Instances

For each PostgreSQL instance, the operator provides an exporter of metrics for *Part 4 - Monitor clusters with Prometheus and Grafana* 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 "Default 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.

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 - flag indicating if replica cluster mode is enabled or disabled - flag indicating if a manual switchover is required
- 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
```

(continues on next page)

(continued from previous page)

```
#  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_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,␣
↪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)
```

(continues on next page)

(continued from previous page)

```
#  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

#  TYPE cnpg_collector_postgres_version gauge

cnpg_collector_postgres_version{cluster="cluster-example",full="13.4.0"} 13.4

#  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 go_gc_duration_seconds A summary of the pause duration of garbage collection␣
↪cycles.

#  TYPE go_gc_duration_seconds summary

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
```

(continues on next page)

(continued from previous page)

```
#  HELP go_info Information about the Go environment.

#  TYPE go_info gauge

go_info{version="go1.17.1"} 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

#  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
```

(continues on next page)

(continued from previous page)

```
#  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.

#  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
```

(continues on next page)

(continued from previous page)

```
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

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
```

(continues on next page)

(continued from previous page)

```
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* .

User defined metrics

This feature is currently in *beta* state and the format is inspired by the queries.yaml file 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

  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 **inthesamenamespaceastheCluster** 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:

```yaml
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:
            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
```

(continues on next page)

(continued from previous page)

```
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:

```yaml
<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.

4.21.2 Monitoring the operator

The operator internally exposes *Part 4 - Monitor clusters with Prometheus and Grafana* 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 example* by defining the following PodMonitor

resource:

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cnpg-controller-manager
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: cloudnative-pg
  podMetricsEndpoints:
    - port: metrics
```

4.21.3 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:

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: curl
spec:
  containers:
  - name: curl
    image: curlimages/curl:7.84.0
    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
```

4.21.4 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.
- `cnpg-prometheusrule.yaml` : a `PrometheusRule` with alerts for CloudNativePG. NOTE: this does not include inter-operation with notification services. Please refer to the Prometheus documentation .
- `grafana-configmap.yaml` : a ConfigMap containing the definition of the sample CloudNativePG Dashboard. Note the labels in the definition, which ensure that the Grafana deployment will find the ConfigMap.

In addition, we provide the "raw" sources for the Grafana dashboard and the Prometheus alert rules, for your reference:

- `alerts.yaml` : Prometheus rules with alerts
- `grafana-dashboard.json` : the CloudNativePG dashboard as a native Grafana JSON.

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.

4.22 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 (e.g. `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 generated

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. Please refer to the Kubernetes Logging Architecture

documentation.

Note: The *level* and *ts* field names can be renamed via the *log-field-level* and *log-field-timestamp* flags of the operator controller, should your log ingestion system require it. All you have to do is edit the *Deployment* definition of the *cloudnative-pg* operator.

4.22.1 Operator log

A log level can be specified in the cluster spec with the option `logLevel` and can be set to any of `error` , `warning` , `info` (default), `debug` or `trace` .

At the moment, the log level can only be set when an instance starts and can not be changed at runtime. If the value is changed in the cluster spec after the cluster was started, this will take effect only in the new pods and not the old ones.

4.22.2 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"
  },
```

(continues on next page)

(continued from previous page)

```
  "logging_pod": "cluster-example-1",
}
```

Internally, the operator relies on the PostgreSQL CSV log format. Please refer to the PostgreSQL documentation for more information about the CSV log format.

4.22.3 PGAudit logs

CloudNativePG has transparent and native support for Enabling `pgaudit`<Enabling `pgaudit>` on PostgreSQL clusters.

All you need to do is add the required `pgaudit` parameters to the `postgresql` section in the configuration of the cluster.

Important: It is unnecessary to add the PGAudit library to *shared_preload_libraries* . The library will be added automatically by CloudNativePG based on the presence of *pgaudit.** parameters in the postgresql configuration. The operator will detect and manage 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* command in all databases in the cluster that accept connections.

Here is an example of a PostgreSQL 13 `Cluster` deployment which will result 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, similar to `logging_collector` logs - except for `.record.audit` , which contains the PGAudit CSV message formatted as a JSON object

See the example below:

```
{
  "level": "info",
  "ts": 1627394507.8814096,
  "logger": "pgaudit",
  "msg": "record",
  "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",
}
```

Please refer to the PGAudit documentation for more details about each field in a record.

4.22.4 Other logs

All logs that are produced by the operator and its instances are in JSON format, with `logger` set accordingly to the process that produced them. Therefore, all the possible `logger` values are the following ones:

- `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` that has the aforementioned structures, all other possible values just have `msg` set to the escaped message that is logged.

4.23 Certificates

CloudNativePG has been designed to natively support TLS certificates. In order to set up a `Cluster` , the operator requires:

- a server Certification Authority (CA) certificate
- a server TLS certificate signed by the server Certification Authority
- a client Certification Authority certificate
- a streaming replication client certificate generated by the client Certification Authority

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, and 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 examples below)

You can also choose a hybrid approach, where only part of the certificates is generated outside CNPG.

4.23.1 Operator managed mode

By default, the operator generates a single Certification Authority 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 Server SSL certificate automatically

Server TLS Secret

The operator uses the generated self-signed CA to sign a server TLS certificate, 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 so that clients can verify their identity and connect securely.

Server alternative DNS names

You can specify DNS server alternative names that will be part of the generated server TLS secret in addition to the default ones.

Client Certificates

Client CA Secret

The same self-signed CA as the Server CA is used by default. The public part will be passed as `ssl_ca_file` to all the instances in order to be able to verify client certificates it signed. The private key will be 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` . This certificate will be passed as `sslcert` and `sslkey` in replicas' connection strings, to allow securely connecting to the primary instance.

4.23.2 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 Example* . In order 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 will still create and manage the two secrets related to client certificates.

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.

See below for a complete example.

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 `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 will use the provided server certificates for TLS connections.

Cert-manager Example

Here is a simple example about how to use *Cert-manager Example* 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
metadata:
```

(continues on next page)

(continued from previous page)

```
  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
```

A Secret named `my-postgres-server-cert` is created by cert-manager, containing 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 cluster-example-cert-manager.yaml deployment manifest.

Client Certificate

If required, you can also provide the two client certificates, generating them using a separate component such as *Cert-manager Example* or

hashicorp vault . In order 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 that should be used to verify client certificate.

Note: The operator will still create and manage the two secrets related to server certificates.

Note: As the Cluster is not in control of the client CA secret key, client certificates can not be generated using *kubectl cnpg certificate* anymore.

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.

Cert-manager Example

Here a simple example about how to use *Cert-manager Example* 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
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:
```

(continues on next page)

(continued from previous page)

```
    - client auth
  commonName: streaming_replica
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

A Secret named `my-postgres-client-cert` is created by cert-manager, containing 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:
    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 cluster-example-cert-manager.yaml deployment manifest.

4.24 Client TLS/SSL Connections

page for more details on how CloudNativePG supports TLS certificates.

The CloudNativePG operator has been designed to work with TLS/SSL for both encryption in transit and authentication, on server and client sides. Clusters created using the CNPG operator comes with a Certification Authority (CA) to create and sign TLS client certificates. Through the `cnpg` plugin for `kubectl` you can issue a new TLS client certificate which can be used to authenticate a user instead of using passwords.

Please refer to the following steps to authenticate via TLS/SSL certificates, which assume you have installed a cluster using the

> cluster-example-pg-hba.yaml

manifest. According to the convention over configuration paradigm, that file automatically creates an `app` database which is owned by a user called `app` (you can change this convention through the `initdb` configuration in the `bootstrap` section).

4.24.1 Issuing a new certificate

page 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 with:

```
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. This is necessary to leverage the `cert` authentication method for `hostssl` entries in `pg_hba.conf` .

4.24.2 Testing the connection via a TLS certificate

Now we will test this client certificate by configuring a demo client application that connects to our 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:
```

(continues on next page)

(continued from previous page)

```
    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: >
              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 will mount secrets managed by the CloudNativePG operator, including:

- `sslcert` : the TLS client public certificate
- `sslkey` : the TLS client certificate private key
- `sslrootcert` : the TLS Certification Authority certificate, that signed the certificate on the server to be used to verify the identity of the instances

They will be 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 inside the `~/.postgresql` directory of the current user, but we can use the sslkey, sslcert, sslrootcert options to point libpq to the actual location of the cryptographic material. The content of the above files is gathered from the secrets that were previously created by using the `cnpg` plugin for kubectl.

Now deploy the application:

```
kubectl create -f cert-test.yaml
```

Then we will use created Pod as PostgreSQL client to validate SSL connection and authentication using TLS certificates we just created.

A readiness probe has been configured to ensure that the application is ready when the database server can be reached.

You can verify that the connection works by executing 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. We will 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 15.2 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat
8.3.1-5), 64-bit
(1 row)
```

4.25 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 The `microservice type<The *microservice* type>` .

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.

Warning: The operator will create another service, named *[cluster name]-any* . That service is used internally to manage PostgreSQL instance discovery. It's not supposed to be used directly by applications.

4.25.1 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.

4.25.2 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

4.25.3 Secrets

The PostgreSQL operator will generate two `basic-auth` type secrets for every PostgreSQL cluster it deploys:

- `[cluster name]-superuser`
- `[cluster name]-app`

The secrets contain the username, password, and a working .pgpass file

respectively for the `postgres` user and the *owner* of the database.

The `-app` credentials are the ones that should be used by applications connecting to the PostgreSQL cluster.

The `-superuser` ones are supposed to be used only for administrative purposes.

4.26 Connection Pooling

CloudNativePG provides native support for connection pooling with
PgBouncer configuration options , one of the most popular open source connection poolers

for PostgreSQL, through the `Pooler` CRD.

In a nutshell, a `Pooler` in CloudNativePG is a deployment of PgBouncer pods that sits between your applications and a PostgreSQL service (for example the `rw` service), creating a separate, scalable, configurable, and highly available **databaseaccesslayer** .

4.26.1 Architecture

The following diagram highlights how the introduction of a database access layer based on PgBouncer changes the architecture of CloudNativePG, like an additional blade in a Swiss Army knife. Instead of directly connecting to the PostgreSQL primary service, applications can now connect to the equivalent service for PgBouncer, enabling reuse of existing connections for faster performance and better resource management on the PostgreSQL side.

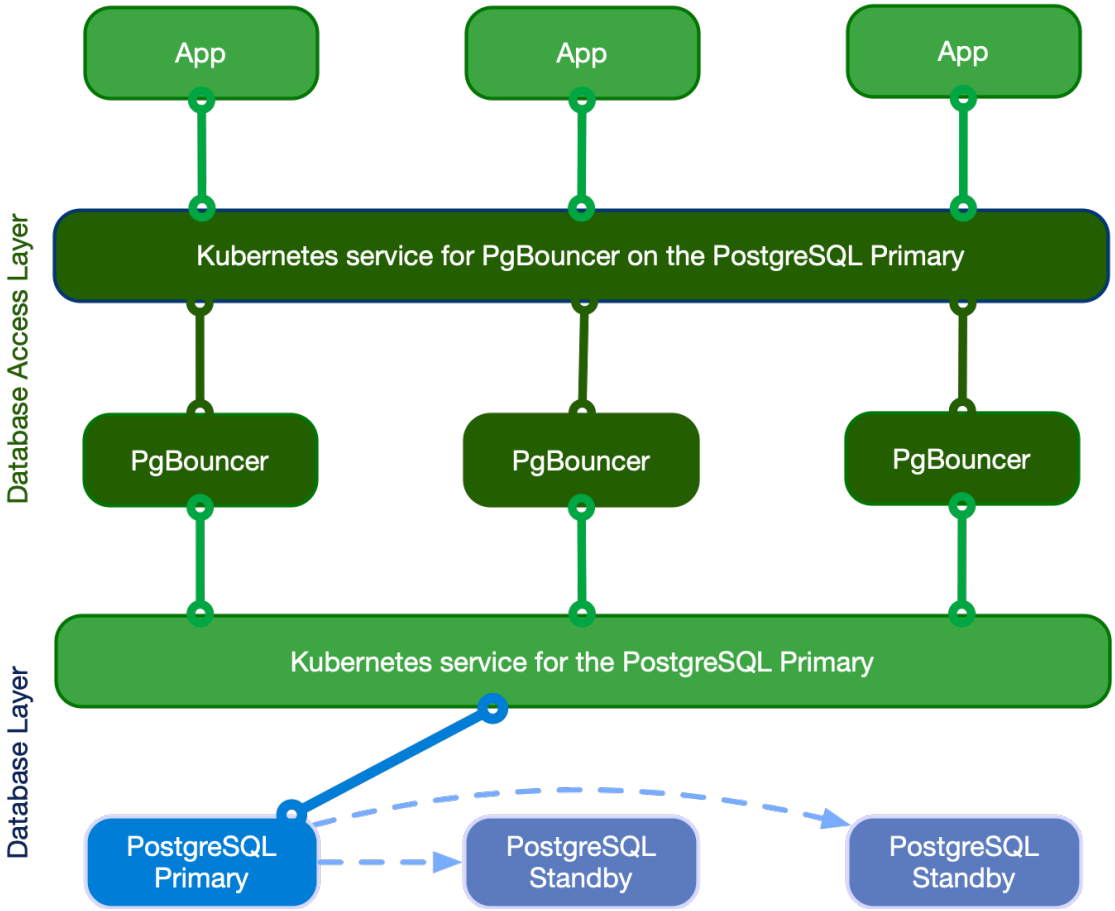

Fig. 12: Applications writing to the single primary via PgBouncer

4.26.2 Quickstart

The easiest way to explain how CloudNativePG implements a PgBouncer pooler is through an example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example

  instances: 3
  type: rw
```

(continues on next page)

(continued from previous page)

```
pgbouncer:
  poolMode: session
  parameters:
    max_client_conn: "1000"
    default_pool_size: "10"
```

Important: Pooler name should never match with any Cluster name within the same namespace.

This creates a new `Pooler` resource called `pooler-example-rw` (the name is arbitrary) that is strictly associated with the Postgres `Cluster` resource called `cluster-example` and pointing to the primary, identified by the read/write service (`rw` , therefore `cluster-example-rw`).

The `Pooler` must live in the same namespace of 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 - with a default pool size of 10 user/database pairs towards PostgreSQL.

Important: The *Pooler* only sets the * fallback database in PgBouncer, meaning that all parameters in the connection strings passed from the client are relayed to the PostgreSQL server (please refer to [“Section [databases]” in PgBouncer’s documentation](https://www.pgbouncer.org/config.html#section-databases)).

Additionally, CloudNativePG automatically creates a secret with the same name of the pooler containing the configuration files used with PgBouncer.

in the API reference.

4.26.3 Pooler resource lifecycle

`Pooler` resources are not `Cluster` -managed resources. You are supposed to create poolers manually when they are needed. Additionally, you can deploy multiple poolers per PostgreSQL Cluster.

What is important to note is that the lifecycles of the `Cluster` and the `Pooler` resources are currently independent: the deletion of the `Cluster` doesn’t imply the automatic deletion of the `Pooler` , and viceversa.

Important: Now that 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 (i.e. one per application).

4.26.4 Security

Any PgBouncer pooler is transparently integrated with CloudNativePG support for in-transit encryption via **TLSconnections** , both on the client (application) and server (PostgreSQL) side of the pool.

Specifically, PgBouncer automatically reuses the certificates of the PostgreSQL server. Moreover, it uses TLS client certificate authentication to connect to the PostgreSQL server to run the `auth_query` for clients’ password authentication (see the

Username/Password authentication below).

Containers run as the `pgbouncer` system user, and access to the `pgbouncer` database is only allowed via local connections, through `peer` authentication.

Certificates

By default, PgBouncer pooler will use the same certificates that are used by the cluster itself, but if the user provides those certificates the pooler will accept secrets with the following format:

1. Basic Auth
2. TLS
3. Opaque

In the Opaque case, it will look for specific keys that needs to be used, those keys are the following:

- tls.crt
- tls.key

So we can treat this secret as a TLS secret, and start from there.

4.26.5 Authentication

Passwordbasedauthentication is the only supported method for clients of PgBouncer in CloudNativePG.

Internally, our 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 does not have any pooler associated to it

Important: If you specify your own secrets the operator will not automatically integrate the Pooler.

To manually integrate the Pooler, in the case that you have 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, connect in each application database, then create the authentication function inside each of the application databases:

```
CREATE OR REPLACE FUNCTION user_search(uname TEXT)
  RETURNS TABLE (username name, passwd text)
  LANGUAGE sql SECURITY DEFINER AS
  SELECT username, passwd FROM pg_shadow WHERE username=$1;;

REVOKE ALL ON FUNCTION user_search(text)
  FROM public;

GRANT EXECUTE ON FUNCTION user_search(text)
  TO cnpg_pooler_pgbouncer;
```

4.26.6 PodTemplates

You can take advantage of pod templates specification in the `template` section of a `Pooler` resource. For details, please refer to `PoolerSpec section <api_reference.md#PoolerSpec>`__ in the API reference.

Through 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` .

Here an example of Pooler specifying PodAntiAffinity:

```yaml
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:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - pooler
            topologyKey: "kubernetes.io/hostname"
```

Note: *.spec.template.spec.containers* has to be explicitly set to *[]* when not modified, as it's a required field for a *PodSpec* . If *.spec.template.spec.containers* is not set the kubernetes api-server will return the following error when trying to apply the manifest: *error validating "pooler.yaml": error validating data: Validation-*

Error(Pooler.spec.template.spec): missing required field "containers"

Here an example setting resources and changing 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
```

4.26.7 High Availability (HA)

Thanks to Kubernetes' deployments, you can configure your pooler to run on a single instance or over multiple pods. The exposed service will make sure that your clients are randomly distributed over the available pods running PgBouncer - which will then automatically manage and reuse connections towards the underlying server (if using the `rw` service) or servers (if using the `ro` service with multiple replicas).

Warning: Please be aware of network hops in case your infrastructure spans multiple availability zones with high latency across them. Consider for example the case of your application running in zone 2, connecting to PgBouncer running in zone 3, pointing to the PostgreSQL primary in zone 1.

4.26.8 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 to correctly set the value of each option, as the operator does not validate them.

Below you can find a list of the PgBouncer options you are allowed to customize. Each of them contains a link to the PgBouncer documentation for that specific parameter. Unless differently stated here, 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
  Monitoring section below
- max_client_conn
- max_db_connections
- 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 will have the same configuration, aligned with the parameters in the specification. A mistake in these parameters could disrupt the operability of the **whole Pooler**. The operator **does not** validate the value of any option.

4.26.9 Monitoring

The PgBouncer implementation of the `Pooler` comes with a default Prometheus exporter that automatically 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`)

Similarly to the CloudNativePG instance, the exporter runs on port `9127` of each pod running PgBouncer, and also provides metrics related to the Go runtime (with prefix `go_*`). You can debug the exporter on a pod running PgBouncer through the following command:

```
kubectl exec -ti <PGBOUNCER_POD> -- curl 127.0.0.1:9127/metrics
```

An example of 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.
```

(continues on next page)

(continued from previous page)

```
#  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

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
```

(continues on next page)

(continued from previous page)

```
#  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.

#  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
```

(continues on next page)

(continued from previous page)

```
#  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

#  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.
```

(continues on next page)

(continued from previous page)

```
#  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.

#  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.
```

(continues on next page)

(continued from previous page)

```
#  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.

#  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
```

Like for `Clusters` , if you are using the *Prometheus Operator example*

you can configure it to scrape a specific Pooler by defining the following PodMonitor :

```yaml
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: <POOLER_NAME>
spec:
  selector:
    matchLabels:
      cnpg.io/poolerName: <POOLER_NAME>
  podMetricsEndpoints:
  - port: metrics
```

4.26.10 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"
  }
}
```

4.26.11 Pausing connections

The `Pooler` specification allows you to take advantage of PgBouncer's `PAUSE` and `RESUME` commands, using only declarative configuration - via the `paused` option, by default set to `false` . When set to `true` , the operator internally invokes the `PAUSE` command in PgBouncer, which:

1. closes all active connections towards the PostgreSQL server, after waiting for the queries to complete
2. pauses any new connection coming from the client

When the `paused` option is set back to `false` , the operator will invoke the `RESUME` command in PgBouncer, re-opening the taps towards the PostgreSQL service defined in the `Pooler` .

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. At the moment, you can achieve the same results by setting the *paused* attribute to *true* , then issuing the switchover command through the Using the `cnpg plugin for *kubectl*`*<Using the `cnpg* plugin for *kubectl>*` , and finally restoring the *paused* attribute to *false* .

4.26.12 Limitations

Single PostgreSQL cluster

The current implementation of the pooler is designed to work as part of a specific CloudNativePG cluster (a service, to be precise). It is not possible at the moment to create a pooler that spans over multiple clusters.

Controlled configurability

CloudNativePG transparently manages several configuration options that are used for the PgBouncer layer to communicate with PostgreSQL. Such options are not configurable from outside and include TLS certificates, authentication settings, `databases` section, and `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: We have reasons to believe that the adopted solution addresses the majority of use cases, while leaving room for the future implementation of a separate operator for PgBouncer to complete the gamma with more advanced and customized scenarios.

4.27 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" section containing more information about this capability - shows just an example of architecture that you can implement with replica clusters.

4.27.1 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 available options in terms of replication, both at bootstrap and continuous recovery level, are:

- 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 a Barman Cloud object store for recovery of the base backups and 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. Please refer to the *Bootstrap*

for information on how to clone a PostgreSQL server using either `pg_basebackup` (streaming) or `recovery` (object store).

If the external cluster contains a `barmanObjectStore` section:

- you'll be able to bootstrap the replica cluster from an object store using the `recovery` section

- CloudNativePG will automatically set the `restore_command` in the designated primary instance

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

4.27.2 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 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`

This **firstexample** 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
```

The **secondexample** 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.

```yaml
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.

```yaml
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.

4.27.3 Promoting the designated primary in the replica cluster

To promote the **designatedprimary** to **primary** , all we need to do is to disable the replica mode in the replica cluster through the option `spec.replica.enabled`

```
replica:
  enabled: false
  source: cluster-example
```

Once the replica mode is disabled, the replica cluster and the source cluster will become two separate clusters, and the **designatedprimary** in the replica cluster will be promoted to be that cluster's **primary** . We 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-replica-example
```

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 will become two independent clusters definitively.

4.28 Kubernetes Upgrade

Kubernetes clusters must be kept updated. This becomes even more important if you are self-managing your Kubernetes clusters, especially on **baremetal** .

Planning and executing regular updates is a way for your organization to clean up the technical debt and reduce the business risks, despite the introduction in your Kubernetes infrastructure of controlled downtimes that temporarily take out a node from the cluster for maintenance reasons (recommended reading: Embracing Risk

from the Site Reliability Engineering book).

For example, you might need to apply security updates on the Linux servers where Kubernetes is installed, or to replace a malfunctioning hardware component such as RAM, CPU, or RAID controller, or even upgrade the cluster to the latest version of Kubernetes.

Usually, maintenance operations in a cluster are performed one node at a time by:

1. evicting the workloads from the node to be updated (`drain`)
2. performing the actual operation (for example, system update)
3. re-joining the node to the cluster (`uncordon`)

The above process requires workloads to be either stopped for the entire duration of the upgrade or migrated to another node.

While the latest case is the expected one in terms of service reliability and self-healing capabilities of Kubernetes, there can be situations where it is advised to operate with a temporarily degraded cluster and wait for the upgraded node to be up again.

In particular, if your PostgreSQL cluster relies on **node-localstorage** - that is *storage which is local to the Kubernetes worker node where the PostgreSQL database is running*. Node-local storage (or simply *local storage*) is used to enhance performance.

Note: If your database files are on shared storage over the network, you may not need to define a maintenance window. If the volumes currently used by the pods can be reused by pods running on different nodes after the drain, the default self-healing behavior of the operator will work fine (you can then skip the rest of this section).

When using local storage for PostgreSQL, you are advised to temporarily put the cluster in **maintenancemode** 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.

4.28.1 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.

4.29 Exposing Postgres Services

This section explains how to expose a PostgreSQL service externally, allowing access to your PostgreSQL database **fromoutsideyourKubernetescluster** 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 **legacyapplications** 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.

```
apiVersion: v1
kind: ConfigMap
metadata:
```

(continues on next page)

(continued from previous page)

```
  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.

```
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
```

You can use cluster-expose-service.yaml and apply it using `kubectl` .

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.

4.29.1 Testing on Minikube

On Minikube you can setup the ingress controller running:

```
minikube addons enable ingress
```

Then, patch the `tcp-service` ConfigMap to redirect to the primary the connections on port 5432 of the Ingress:

```
kubectl patch configmap tcp-services -n kube-system \
  --patch {"data":{"5432":"default/cluster-example-rw:5432"}}
```

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: nginx-ingress-controller
        ports:
         - containerPort: 5432
           hostPort: 5432
```

and apply it to the `nginx-ingress-controller deployment` :

```
kubectl patch deployment nginx-ingress-controller --patch "$(cat patch.yaml)" -n kube-
↪system
```

You can access the primary from your machine running:

```
psql -h $(minikube ip) -p 5432 -U postgres
```

4.30 CloudNativePG Plugin

CloudNativePG provides a plugin for `kubectl` to manage a cluster in Kubernetes.

4.30.1 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 1.18.1 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/v1.18.1/
↪kubectl-cnpg_1.18.1_linux_x86_64.deb
```

Then, install from the local file using `dpkg` :

```
$ dpkg -i kubectl-cnpg_1.18.1_linux_x86_64.deb
(Reading database ... 16102 files and directories currently installed.)
Preparing to unpack kubectl-cnpg_1.18.1_linux_x86_64.deb ...
Unpacking cnpg (1.18.1) over (1.18.1) ...
Setting up cnpg (1.18.1) ...
```

RPM packages

As in the example for `.deb` packages, let's install the 1.18.1 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/v1.18.1/
↪kubectl-cnpg_1.18.1_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
=================================================================================================
Installing:
```

(continues on next page)

(continued from previous page)

```
 cnpg                x86_64                1.18.1-1                   @commandline          ␣
↪       14 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 *Using Krew* installed, you can simply run:

```
kubectl krew install 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

4.30.2 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-namespaces "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:

- **generalinformation** : 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
- **streamingreplication** : 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:15.2
Primary instance:   sandbox-2
Instances:          3
Ready instances:    3
Current Write LSN:  3AF/EAFA6168 (Timeline: 8 - WAL File: 00000008000003AF00000075)
```

(continues on next page)

(continued from previous page)

```
Continuous Backup status
First Point of Recoverability:  Not Available
Working WAL archiving:          OK
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:15.2
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
```

(continues on next page)

(continued from previous page)

```
archive_timeout = 5min
checkpoint_completion_target = 0.9
checkpoint_timeout = 900s
cluster_name = sandbox
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
```

(continues on next page)

(continued from previous page)

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

#  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 secrete 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

- **deploymentinformation** : the operator Deployment and operator Pod
- **configuration** : the Secrets and ConfigMaps in the operator namespace

- **events** : the Events in the operator namespace
- **webhookconfiguration** : the mutating and validating webhook configurations
- **webhookservice** : 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
```

You can verify that the confidential information is REDACTED:

```
cd report_operator_<TIMESTAMP>/manifests/
head cnpg-ca-secret.yaml
```

```
data:
  ca.crt: ""
  ca.key: ""
metadata:
  creationTimestamp: "2022-03-22T10:42:28Z"
```

(continues on next page)

(continued from previous page)

```
  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
```

```
data:
  ca.crt: LS0tLS1CRUdJTiBD...
  ca.key: LS0tLS1CRUdJTiBF...
metadata:
  creationTimestamp: "2022-03-22T10:42:28Z"
  managedFields:
  - apiVersion: v1
    fieldsType: FieldsV1
```

report Cluster

The `cluster` sub-command gathers the following:

- **clusterresources** : the cluster information, same as `kubectl get cluster -o yaml`
- **clusterpods** : pods in the cluster namespace matching the cluster name
- **clusterjobs** : jobs, if any, in the cluster namespace matching the cluster name
- **events** : events in the cluster namespace
- **podlogs** : logs for the cluster Pods (optional, off by default) in JSON-lines format
- **joblogs** : 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
```  ### 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  **clusterhibernation** .

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

```
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

```
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

```
###  Benchmarking the database with pgbench

Pgbench can be ran on an existing PostgreSQL cluster with following command:
```

kubectl cnpg pgbench – –time 30 –client 1 –jobs 1

```
Refer to the   :ref:`Benchmarking pgbench section <benchmarking.md#pgbench>`  for more␣
↪details.

###  Benchmarking the storage with fio

fio can be run on an existing storage class with following command:
```

kubectl cnpg fio -n

```
Refer to the   :ref:`Benchmarking fio section <benchmarking.md#fio>`  for more details.

###  Requesting a new base backup

The `kubectl cnpg backup`  command requests a new physical base backup for an existing␣
↪Postgres cluster by creating a new `Backup`  resource.

The following example requests an on-demand backup for a given cluster:

```shell
kubectl cnpg backup [cluster_name]
```
```

The created backup will be named after the request time:

edb_notranlate_50 By default, new created backup will use the backup target policy defined in cluster to choose which instance to run on. You can also use `--backup-target` option to override this policy. please refer to *Backup and Recovery*

for more information about backup target.

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 (15.2 (Debian 15.2-1.pgdg110+1))
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 (15.2 (Debian 15.2-1.pgdg110+1))
```

(continues on next page)

(continued from previous page)

```
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.

4.31 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 **failovermode** . 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.

4.31.1 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.

4.31.2 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.

4.32 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 Using the `cnpg plugin for *kubectl`<Using the `cnpg* plugin for *kubectl*>` page bookmarked!

4.32.1 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 *Storage space usage* , 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, Certmanager, . . .)
- 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:

- Using the `cnpg plugin for *kubectl`<Using the `cnpg* plugin for *kubectl*>` 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.

4.32.2 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.

4.32.3 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
```

(continues on next page)

(continued from previous page)

```
PostgreSQL Image:   ghcr.io/cloudnative-pg/postgresql:15.2-3
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
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
```

4.32.4 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 Using the `cnpg plugin for *kubectl*`<*Using the `cnpg* plugin for *kubectl*>` documentation.

Get PostgreSQL container image version:

```
kubectl describe cluster <CLUSTER_NAME> -n <NAMESPACE> | grep "Image Name"
```

Output:

```
Image Name:    ghcr.io/cloudnative-pg/postgresql:15.2-3
```

Note: Also you can use *kubectl-cnpg status -n <NAMESPACE> <CLUSTER_NAME>* to get the same information.

4.32.5 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",
  "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
```

4.32.6 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.

4.32.7 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.

4.32.8 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 .

4.32.9 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
```

(continues on next page)

(continued from previous page)

```
$ 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
```

4.32.10 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
```

4.33 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.

4.33.1 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"]
[...]
```

4.33.2 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 "*"
```

4.33.3 How fencing works

Once an instance is set for fencing, the procedure to shut down the `postmaster` process is initiated. This consists of an initial smart shutdown with a timeout set to `.spec.stopDelay` , followed by a fast shutdown if required. 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` .

4.34 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

4.34.1 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 .

4.34.2 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 postgis-example.yaml below provides some guidance on how the creation of a PostGIS cluster can be done.

> **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.

```yaml
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 (15.2 (Debian 15.2-1.pgdg110+1))
Type "help" for help.

app=# SELECT * FROM pg_available_extensions WHERE name ~ ^postgis ORDER BY 1;
           name           | default_version | installed_version |
↪  comment
```

(continues on next page)

(continued from previous page)

```
- ------------------------+-----------------+-------------------+-----------------------
↪-------------------------------------
 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
```

(continues on next page)

(continued from previous page)

```
(1 row)
```

4.35 End-to-End Tests

CloudNativePG is automatically tested after each commit via a suite of **End-to-end(E2E)tests** , which ensures that the operator correctly deploys and manages PostgreSQL clusters.

Moreover, the following Kubernetes versions are tested for each commit, helping detect bugs at an early stage of the development process:

- 1.26
- 1.25
- 1.24
- 1.23
- 1.22

The following PostgreSQL versions are tested:

- PostgreSQL 15
- PostgreSQL 14
- PostgreSQL 13
- PostgreSQL 12
- PostgreSQL 11

For each tested version of Kubernetes and PostgreSQL, a Kubernetes cluster is created using *Kind* , 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;
- **Serviceconnectivity:** * 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;
- **BackupandRestore:** * 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; * Wal-Restore;
- **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:** * Physical replica clusters; * Replication Slots; * Synchronous replication; * Scale-up and scale-down of a Cluster;
- **Plugin:** * Cluster Hibernation using CNPG plugin; * Fencing; * Creation of a connection certificate;
- **PostgresConfiguration:** * Manage PostgreSQL configuration changes; * Rolling updates when changing PostgreSQL images;
- **PodScheduling:** * Tolerations and taints; * Pod affinity using `NodeSelector` ;

- **ClusterMetadata:** * ConfigMap for Cluster Labels and Annotations; * Object metadata;
- **Recovery:** * Data corruption; * pg_basebackup;
- **ImportingDatabases:** * Microservice approach; * Monolith approach;
- **Storage:** * Storage expansion;
- **Security:** * AppArmor annotation propagation. Executed only on Azure environment;
- **Maintenance:** * Node Drain;

4.36 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 .

4.36.1 Image tag requirements

While the image name can be anything valid for Docker, the CloudNativePG operator relies on the *image tag* to detect the Postgres major version contained in the image.

The image tag must start with a valid PostgreSQL major version number (e.g. 14.5 or 15) optionally followed by a dot and the patch level.

This can be followed by any character combination that is valid and accepted in a Docker tag, preceded by a dot, an underscore, or a minus sign.

Examples of accepted image tags:

- `11.1`
- `12.3.2.1-1`

- `12.4`
- `13`
- `14.5-10`
- `15.0`

Warning: *latest* is not considered a valid tag for the image.

4.37 Operator Capability Levels

This section provides a summary of the capabilities implemented by CloudNativePG, classified using the Operator SDK definition of Capability Levels

framework.

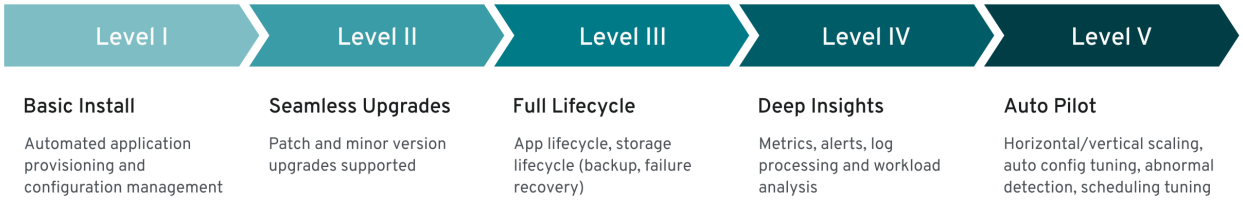

Fig. 13: Operator Capability Levels

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.

4.37.1 Level 1 - Basic Install

Capability level 1 involves **installation** and **configuration** of 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 which defines 4 major `CustomResourceDefinition` objects: `Cluster` , `Pooler` , `Backup` , and `ScheduledBackup` .

PostgreSQL cluster deployment via declarative configuration

A PostgreSQL cluster (operand) is defined 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 automatically creates the following resources: `Pod` , `Service` , `Secret` , `ConfigMap` ,`PersistentVolumeClaim` , `PodDisruptionBudget` , `ServiceAccount` , `RoleBinding` , `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, as well as digests in addition to tags for finer control of container image immutability.

Labels and annotations

The operator can be configured to support inheritance of labels and annotations that are defined in a cluster's metadata, with the goal to improve organizations of CloudNativePG deployment in your Kubernetes infrastructure.

Self-contained instance manager

Instead of relying on an external tool such as Patroni or Stolon to coordinate PostgreSQL instances in the Kubernetes cluster pods, 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 itself based on the PostgreSQL cluster topology. The instance manager also starts a web server that is invoked by the `kubelet` for probes. Unix signals invoked by the `kubelet` are filtered by the instance manager and, where appropriate, 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 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 automatically 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 automatically manages replication slots for all the replicas in the HA cluster, with an implementation that is inspired by the previously proposed patch for PostgreSQL called Failover slots .

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 automatically 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. Both the database name and the user name can be specified if required. Although no configuration is required to run the cluster, you can customize both PostgreSQL run-time configuration and PostgreSQL Host-Based Authentication rules in the `postgresql` section of the CR.

Pod Security Policies

For InfoSec requirements, the operator does not require privileged mode for any container and enforces 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

Command line interface

CloudNativePG does not have its own command line interface. It simply relies on the best command line interface for Kubernetes, `kubectl` , by providing a plugin called `cnpg` to enhance and simplify 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 automatically creates a certification authority for itself. It creates and signs with the operator certification authority a leaf certificate to be used by the webhook server, to ensure safe communication between the Kubernetes API Server and the operator itself.

Cluster's certification authority

The operator automatically creates a certification authority for every PostgreSQL cluster, which 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: this 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: this also includes integration with cert-manager.

Certificate authentication for streaming replication

The operator relies on TLS client certificate authentication to authorize streaming replication connections from the standby servers, 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 and makes sure that all instances are properly reloaded or restarted, depending on the configuration option. *Current limitation:* changes with `ALTER SYSTEM` are not detected, meaning that the cluster state is not enforced.

Import of existing PostgreSQL databases

Since version 1.16, the operator provides a declarative way to import existing Postgres databases in a new CloudNativePG `Cluster` in Kubernetes, using offline migrations. The same feature covers also 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 - which can be accessed via the network through a superuser connection. Import is from any supported version of Postgres and relies on `pg_dump` and `pg_restore` to be executed from the new cluster primary for all databases part of the operation and, if requested, for roles.

PostGIS clusters

CloudNativePG supports the installation of clusters with the
PostGIS

open source extension for geographical databases, 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 PostgreSQL documentation: LDAP authentication .

Multiple installation methods

The operator can be installed through a Kubernetes manifest via `kubectl apply`, to be used in a traditional Kubernetes installation in public and private cloud environments. Additionally, a Helm Chart for the operator is also available.

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 YAML code lines.

4.37.2 Level 2 - Seamless Upgrades

Capability level 2 is about enabling **updatesoftheoperatorandtheactualworkload** , in our case PostgreSQL servers. This includes **PostgreSQLminorreleaseupdates** (security and bug fixes normally) as well as **majoronlineupgrades** .

Upgrade of the operator

You can upgrade the operator seamlessly as a new deployment. A change in the operator does not require a change in the operand - thanks to the instance manager's injection. 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 do not 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 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 waits for the user to manually issue the switchover procedure (`supervised`) via the `cnpg` plugin for `kubectl` . Which setting to use depends on the business requirements as the operation might generate some downtime for the applications, 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` , `Upgrading cluster` , etc.

4.37.3 Level 3 - Full Lifecycle

Capability level 3 requires the operator to manage aspects of **businesscontinuity** and **scalability** . **Disasterrecovery** 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. **HighAvailability** is the other important component of business continuity that, through PostgreSQL native physical replication and hot standby replicas, 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 Hot Backups

The operator has been designed to provide application-level backups using PostgreSQL's native continuous hot backup technology based on physical base backups and continuous WAL archiving. Specifically, the operator currently supports only backups on object stores (AWS S3 and S3-compatible, Azure Blob Storage, Google Cloud Storage, and gateways like MinIO).

WAL archiving and base backups are defined at the cluster level, declaratively, through the `backup` parameter in the cluster definition, 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, does not require any further action from the user: the operator will automatically and transparently set the `archive_command` to rely on `barman-cloud-wal-archive` to ship WAL files to the defined endpoint. Users can decide the compression algorithm, as well as the number of parallel jobs to concurrently upload WAL files in the archive. In addition to that `Instance Manager` automatically checks the correctness of the archive destination, by performing `barman-cloud-check-wal-archive` command before beginning to ship the very first set of WAL files.

You can define base backups in two ways: on-demand (through the `Backup` custom resource definition) or scheduled (through the `ScheduledBackup` customer resource definition, using a cron-like syntax). They both 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.

Backups from a standby

The operator supports offloading base backups onto a standby without impacting the RPO of the database. This allows to preserve resources 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 taken using `barman-cloud-backup` . Once the bootstrap process is completed, the operator initiates the instance in recovery mode and replays all available WAL files from the specified archive, exiting recovery and starting as a primary. Subsequently, the operator will clone 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 will react accordingly, based on the number of available and ready PostgreSQL instances in the cluster, through the following formula for the quorum (`q`):

```
1 <= minSyncReplicas <= q <= maxSyncReplicas <= readyReplicas
```

Replica clusters

Define a cross Kubernetes cluster topology of PostgreSQL clusters, by taking advantage of PostgreSQL native streaming and cascading replication. Through the `replica` option, you can setup an independent cluster to be continuously replicating data from another PostgreSQL source of the same major version: such a source can be anywhere, as long as a direct streaming connection via TLS is allowed from the two endpoints. Moreover, the source can be even outside Kubernetes, running in a physical or virtual environment. Replica clusters can be created from a recovery object store (backup in Barman Cloud format) or via streaming through `pg_basebackup` . Both WAL file shipping and WAL streaming are allowed. Replica clusters dramatically improve the business continuity posture of your PostgreSQL databases in Kubernetes, spanning over multiple datacenters and opening up for hybrid and multi-cloud setups (currently, manual switchover across data centers is required, while waiting for Kubernetes federation native capabilities).

Liveness and readiness probes

The operator defines liveness and readiness probes for the Postgres Containers that are then invoked by the kubelet. They are 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 and, if a PostgreSQL cluster is exposed publicly, the Service will load-balance 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 automatically started up from the primary server and will participate in the cluster's HA infrastructure. The CRD declares a "scale" subresource that allows the user 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 will be 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 may 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 be used: 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 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.

Hibernation

CloudNativePG supports hibernation of a running PostgreSQL cluster

via the `cnpg` plugin. Hibernation shuts down all Postgres instances in the High Availability cluster, and keep 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 - where they exist.

Reuse of Persistent Volumes storage in Pods

When the operator needs to create a pod that has been deleted by the user or has been evicted by a Kubernetes maintenance operation, it reuses the `PersistentVolumeClaim` if available, avoiding the need to re-clone the data from the primary.

CPU and memory requests and limits

The operator allows administrators to control and manage resource usage by the cluster's pods, through the `resources` section of the manifest. In particular `requests` and `limits` values can be set for both CPU and RAM.

Connection pooling with PgBouncer

CloudNativePG provides native support for connection pooling with
PgBouncer configuration options , 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 which optimizes the query flow towards the instances and makes the usage 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.

4.37.4 Level 4 - Deep Insights

Capability level 4 is about **observability** : in particular, monitoring, alerting, trending, log processing. This might involve the use of external tools such as Prometheus, Grafana, Fluent Bit, as well as extensions in the PostgreSQL engine for the output of error logs directly in JSON format.

CloudNativePG has been designed to provide everything that is 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 and, via its own web server listening on the `metrics` port (9187), exposes an endpoint to export metrics for the *Part 4 - Monitor clusters with Prometheus and Grafana* monitoring and alerting tool. The operator supports custom monitoring queries defined as `ConfigMap` and/or `Secret` objects using a syntax that is compatible with the

postgres_exporter . CloudNativePG provides a set of basic monitoring

queries for PostgreSQL that can be integrated and adapted to your context.

Standard output logging of PostgreSQL error messages in JSON format

Every log message is delivered to standard output in JSON format, with the first level 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 Enabling `pg_stat_statements`<Enabling `pg_stat_statements>` , which enables tracking of planning and execution statistics of all SQL statements executed by a PostgreSQL server.
- the Enabling `auto_explain`<Enabling `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).

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, upgrading, and so on. Events can be displayed through the `kubectl describe` and `kubectl get events` command.

4.37.5 Level 5 - Auto Pilot

Capability level 5 is focused on **automatedscaling** , **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 will change 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 will initiate the required procedures to align itself with the requested status of the cluster, by either becoming the new primary or by following it. In case the former primary comes back up, the same mechanism will avoid 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

In case the pod hosting a standby has been removed, the operator initiates the procedure to recreate a standby server.

4.38 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.

4.38.1 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.

4.38.2 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.

4.38.3 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.

4.38.4 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 section for details).

Being generic, the `StatefulSet` doesn't allow this level of customization.

4.39 Examples

In this section, you can find some examples of configuration files to set up your PostgreSQL Cluster.

Important: These are here for demonstration and experimentation purposes, and can be executed on a personal Kubernetes cluster with Minikube or Kind as described in the *Quickstart* .

Basic cluster : cluster-example.yaml

a basic example of a cluster.

Custom cluster : cluster-example-custom.yaml

a basic cluster that uses the default storage class and custom parameters for the `postgresql.conf` and `pg_hba.conf` files.

Customized storage class : cluster-storage-class.yaml : a basic cluster that uses a specified storage class of `standard` .

Customized storage class and backups : **Prerequisites** : bucket storage should be available. The sample config is for AWS, please change to suit your setup : cluster-storage-class-with-backup.yaml a cluster with backups configured

Backup : **Prerequisites** : cluster-storage-class-with-backup.yaml

applied and Healthy : backup-example.yaml : an example of a backup that runs against the previous sample

Cluster with PVC (Persistent Volume Claim) configured : cluster-pvc-template.yaml : a basic cluster that with an explicit persistent volume claim template.

Full example : cluster-example-full.yaml : a cluster that sets most of the available options.

PostGIS example : postgis-example.yaml : an example of "PostGIS cluster" (see the PostGIS section for details.)

Replica cluster via streaming : **Prerequisites** : cluster-example.yaml

applied and Healthy : cluster-example-replica-streaming.yaml : a replica cluster following `cluster-example` with streaming replication.

Simple cluster with backup configured : **Prerequisites** : The configuration assumes `minio` is running and working. Please update `backup.barmanObjectStore` with your `minio` parameters or your cloud solution : cluster-example-with-backup.yaml

a basic cluster with backups configured.

Replica cluster via backup : **Prerequisites** : cluster-storage-class-with-backup.yaml applied and Healthy. And a backup cluster-example-trigger-backup.yaml

applied and Completed. : cluster-example-replica-from-backup-simple.yaml : a replica cluster following a cluster with backup configured.

Bootstrap cluster with SQL files : cluster-example-initdb-sql-refs.yaml : a cluster example that will execute a set of queries defined in a Secret and a ConfigMap right after the database is created.

Sample cluster with customized `pg_hba` configuration : cluster-example-pg-hba.yaml : a basic cluster that enables user `app` to authenticate using certificates.

Sample cluster with Secret and Configmap mounted using projected volume template : cluster-example-projected-volume.yaml

a basic cluster with existing Secret and Configmap mounted into Postgres pod using projected volume mount.

For a list of available options, please refer to the *API Reference* .

4.40 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 Benchmarking the database with pgbench
- the **storage** , by relying on Benchmarking the storage with fio

Important: *pgbench* and *fio* must be run in a staging or pre-production environment. Do not use these plug-ins 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.

4.40.1 pgbench

The kubectl CNPG plugin command `pgbench` executes a user-defined pgbench job on an existing Postgres Cluster. The command also accepts the `--dry-run` flag, this will output the job manifest without applying it.

Example usage:

```
kubectl cnpg pgbench \
  -n <namespace> <cluster-name> \
  --pgbench-job-name <pgbench-job> \
  --db-name <db-name> \
  -- --time 30 --client 1 --jobs 1
```

Example of how to run `pgbench` against a `Cluster` named `cluster-example` in the `pgbench` namespace:

```
kubectl cnpg pgbench \
   -n pgbench cluster-example \
   --pgbench-job-name pgbench-job \
   -- --time 30 --client 1 --jobs 1
```

Example of how to run `pgbench` on an existing database by using the `--db-name` flag and the `pgbench` namespace:

```
kubectl cnpg pgbench \
  -n pgbench cluster-example \
  --db-name pgbench \
  --pgbench-job-name pgbench-job \
  -- --time 30 --client 1 --jobs 1
```

The job status can be fetched by running:

```
kubectl get job/pgbench-job -n <namespace>

NAME                COMPLETIONS   DURATION   AGE
pgbench-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>
```

4.40.2 fio

The kubectl CNPG plugin command `fio` executes a fio job with default values and read operations. The command also accepts a `--dry-run` flag, this will output the fio manifest without applying it.

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 usage:

default

```
kubectl cnpg fio <fio-name>
```   with 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):

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.

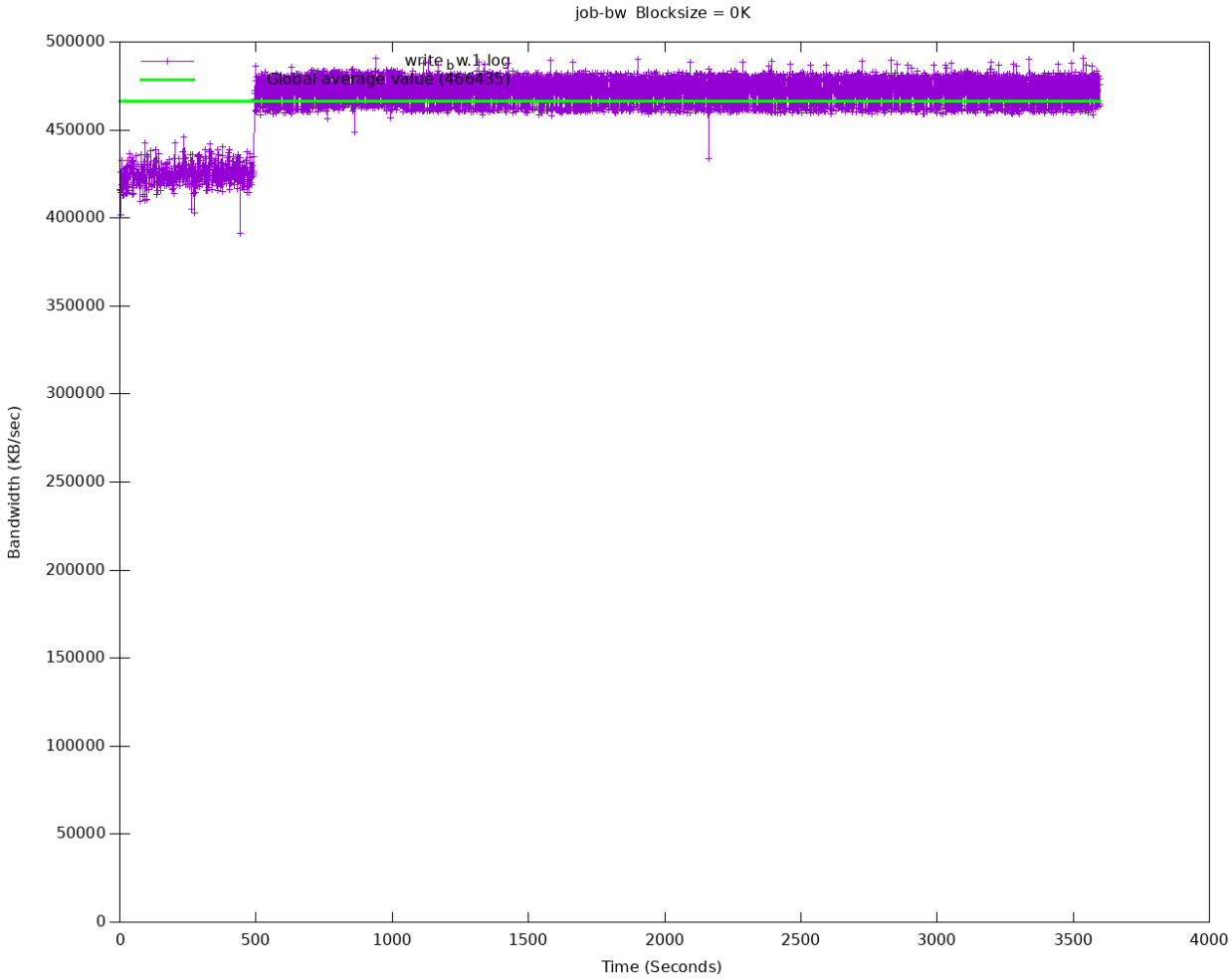

Fig. 14: Sequential writes bandwidth

4.41 Commercial support

CloudNativePG is an independent open source project which doesn't endorse any company.

This is a list of third-party companies and individuals who provide products or services related to CloudNativePG.

If you are providing commercial support for CloudNativePG, please edit this page to add yourself, or your organization to the list.

The list is provided in alphabetical order:

Company	URL	Regions/Countries
ScheduledBackup	enterprisedb.com	Global

4.42 Frequently Asked Questions (FAQ)

4.42.1 Running PostgreSQL in Kubernetes

EveryoneknowsthatstatefulworkloadslikePostgreSQLcannotruninKubernetes.Whydoyousaythecontrary?

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 .

CloudNativePGisbuiltonimmutableapplicationcontainers.Whatdoesitmean?

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"* .

WhatdoesCloudNativemean?

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

WhyshouldIuseanoperatorinsteadofrunningPostgreSQLasacontainer?

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.

YousaythatCloudNativePGisafullydeclarativeoperator.Whatdoyoumeanbythat?

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.

WhataretherequiredskillstorunPostgreSQLonKubernetes?

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.

WhyisntCloudNativePGusingStatefulSets?

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.

4.42.2 High availability

WhathappenstothePostgreSQLclusterswhentheoperatorpoddiesoritisnotavailableforacertainamountoftime?

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.

4.43 API Reference

CloudNativePG extends the Kubernetes API defining the following custom resources:

- `:ref:`Bootstrap from a backup (`recovery`)<Bootstrap from a backup (`recovery`)>``
- `:ref:`Multi-availability zone Kubernetes clusters<Multi-availability zone Kubernetes clusters>``
- `:ref:`Pooler resource lifecycle<Pooler resource lifecycle>``
- `:ref:`ScheduledBackup<ScheduledBackup>`

All the resources are defined in the `postgresql.cnpg.io/v1` API.

Please refer to the Configuration Samples " of the documentation for examples of usage.

Below you will find a description of the defined resources:

- *AffinityConfiguration*
- *AzureCredentials*
- Bootstrap from a backup (`recovery )<Bootstrap from a backup (*recovery* )>`
- *BackupConfiguration*
- *BackupList*
- *BackupSource*
- *BackupSpec*
- *BackupStatus*
- *BarmanCredentials*

- *BarmanObjectStoreConfiguration*
- *BootstrapConfiguration*
- *BootstrapInitDB*
- *BootstrapPgBaseBackup*
- *BootstrapRecovery*
- *CertificatesConfiguration*
- *CertificatesStatus*
- *Multi-availability zone Kubernetes clusters*
- *ClusterList*
- *ClusterSpec*
- *ClusterStatus*
- *ConfigMapKeySelector*
- *ConfigMapResourceVersion*
- *DataBackupConfiguration*
- *EmbeddedObjectMetadata*
- The `externalClusters section<The *externalClusters* section>`
- *GoogleCredentials*
- *Importing Postgres databases*
- *ImportSource*
- *InstanceID*
- *InstanceReportedState*
- *LDAPBindAsAuth*
- *LDAPBindSearchAuth*
- *LDAPConfig*
- *LocalObjectReference*
- Defining cluster's metadata
- *MonitoringConfiguration*
- *NodeMaintenanceWindow*
- *PgBouncerIntegrationStatus*
- *PgBouncerSecrets*
- *PgBouncerSpec*
- *PodTemplateSpec*
- *Pooler resource lifecycle*
- *PoolerIntegrations*
- *PoolerList*
- *PoolerSecrets*

- *PoolerSpec*
- *PoolerStatus*
- *PostInitApplicationSQLRefs*
- *PostgresConfiguration*
- *RecoveryTarget*
- *ReplicaClusterConfiguration*
- *ReplicationSlotsConfiguration*
- *ReplicationSlotsHAConfiguration*
- *RollingUpdateStatus*
- *S3Credentials*
- *ScheduledBackup*
- *ScheduledBackupList*
- *ScheduledBackupSpec*
- *ScheduledBackupStatus*
- *SecretKeySelector*
- *SecretVersion*
- *SecretsResourceVersion*
- *ServiceAccountTemplate*
- *StorageConfiguration*
- *SyncReplicaElectionConstraints*
- *Topology*
- *WalBackupConfiguration*

4.43.1 AffinityConfiguration

AffinityConfiguration contains the info we need to create the affinity rules for Pods

Name	Description	Type
`enablePodAntiAffinity`	Activates anti-affinity for the pods. The operator will define pods anti-affinity unless this field is explicitly set to false	*bool
`topologyKey`	TopologyKey to use for anti-affinity configuration. See k8s documentation for more info on that - *mandatory*	string
`nodeSelector`	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/	map[string]string

continues on next page

Table 7 – continued from previous page

Name	Description	Type
`nodeAffinity`	NodeAffinity describes node affinity scheduling rules for the pod. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity	*corev1.NodeAffinity
`tolerations`	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/	[]corev1.Toleration
`podAntiAffinityType`	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	string
additionalPodAntiAffinity	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.	*corev1.PodAntiAffinity
`additionalPodAffinity`	AdditionalPodAffinity allows to specify pod affinity terms to be passed to all the cluster's pods.	*corev1.PodAffinity

4.43.2 AzureCredentials

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

Name	Description	Type
`connectionString`	The connection string to be used	*SecretKeySelector
`storageAccount`	The storage account where to upload data	*SecretKeySelector
`storageKey`	The storage account key to be used in conjunction with the storage account name	*SecretKeySelector
`storageSasToken`	A shared-access-signature to be used in conjunction with the storage account name	*SecretKeySelector
inheritFromAzureAD	Use the Azure AD based authentication without providing explicitly the keys. - *mandatory*	bool

4.43.3 Backup

Backup is the Schema for the backups API

Name	Description	Type
metadata		metav1.ObjectMeta
`spec`	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	*BackupSpec*
`status`	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	*BackupStatus*

4.43.4 BackupConfiguration

BackupConfiguration defines how the backup of the cluster are taken. Currently the only supported backup method is barmanObjectStore. For details and examples refer to the Backup and Recovery section of the documentation

Name	Description	Type
barmanObjectStore	The configuration for the barman-cloud tool suite	*BarmanObjectStoreConfiguration
`retentionPolicy`	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.	string
`target`	The policy to decide which instance should perform backups. Available options are empty string, which will default to *primary* policy, *primary* to have backups run always on primary instances, *prefer-standby* to have backups run preferably on the most updated standby, if available.	BackupTarget

4.43.5 BackupList

BackupList contains a list of Backup

Name	Description	Type
metadata	Standard list metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	metav1.ListMeta
`items`	List of backups - *mandatory*	[]Backup

4.43.6 BackupSource

BackupSource contains the backup we need to restore from, plus some information that could be needed to correctly restore it.

Name	Description	Type
endpointCA	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.	*SecretKeySelector

4.43.7 BackupSpec

BackupSpec defines the desired state of Backup

Name	Description	Type
cluster	The cluster to backup	*LocalObjectReference*
`target`	The policy to decide which instance should perform this backup. If empty, it defaults to *cluster.spec.backup.target*. Available options are empty string, which will default to *primary* policy, *primary* to have backups run always on primary instances, *prefer-standby* to have backups run preferably on the most updated standby, if available.	BackupTarget

4.43.8 BackupStatus

BackupStatus defines the observed state of Backup

Name	Description	Type
`endpointCA`	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.	*SecretKeySelector
`endpointURL`	Endpoint to be used to upload data to the cloud, overriding the automatic endpoint discovery	string
destinationPath	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.	string
`serverName`	The server name on S3, the cluster name is used if this parameter is omitted	string
`encryption`	Encryption method required to S3 API	string
`backupId`	The ID of the Barman backup	string
`backupName`	The Name of the Barman backup	string
`phase`	The last backup status	BackupPhase
`startedAt`	When the backup was started	*metav1.Time

continues on next page

Table 14 – continued from previous page

Name	Description	Type
`stoppedAt`	When the backup was terminated	*metav1.Time
`beginWal`	The starting WAL	string
`endWal`	The ending WAL	string
`beginLSN`	The starting xlog	string
`endLSN`	The ending xlog	string
`error`	The detected error	string
`commandOutput`	Unused. Retained for compatibility with old versions.	string
`commandError`	The backup command output in case of error	string
`instanceID`	Information to identify the instance where the backup has been taken from	*InstanceID

4.43.9 BarmanCredentials

BarmanCredentials an object containing the potential credentials for each cloud provider

Name	Description	Type
googleCredentials	The credentials to use to upload data to Google Cloud Storage	*GoogleCredentials
`s3Credentials`	The credentials to use to upload data to S3	*S3Credentials
`azureCredentials`	The credentials to use to upload data to Azure Blob Storage	*AzureCredentials

4.43.10 BarmanObjectStoreConfiguration

BarmanObjectStoreConfiguration contains the backup configuration using Barman against an S3-compatible object storage

Name	Description	Type
`endpointURL`	Endpoint to be used to upload data to the cloud, overriding the automatic endpoint discovery	string
`endpointCA`	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	*SecretKeySelector
destinationPath	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 - *mandatory*	string
`serverName`	The server name on S3, the cluster name is used if this parameter is omitted	string
`wal`	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.	*WalBackupConfiguration

continues on next page

Table 16 – continued from previous page

Name	Description	Type
`data`	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.	*DataBackupConfiguration
`tags`	Tags is a list of key value pairs that will be passed to the Barman –tags option.	map[string]string
`historyTags`	HistoryTags is a list of key value pairs that will be passed to the Barman –history-tags option.	map[string]string

4.43.11 BootstrapConfiguration

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.

Name	Description	Type
`initdb`	Bootstrap the cluster via initdb	*BootstrapInitDB
`recovery`	Bootstrap the cluster from a backup	*BootstrapRecovery
pg_basebackup	Bootstrap the cluster taking a physical backup of another compatible PostgreSQL instance	*BootstrapPgBaseBackup

4.43.12 BootstrapInitDB

BootstrapInitDB is the configuration of the bootstrap process when initdb is used Refer to the Bootstrap page of the documentation for more information.

Name	Description	Type
`database`	Name of the database used by the application. Default: *app*. - *mandatory*	string
`owner`	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the *database* key. - *mandatory*	string
`secret`	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	*LocalObjectReference
`options`	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.	[]string
`dataChecksums`	Whether the *-k* option should be passed to initdb, enabling checksums on data pages (default: *false*)	*bool
`encoding`	The value to be passed as option *–encoding* for initdb (default:*UTF8*)	string

continues on next page

Table 18 – continued from previous page

Name	Description	Type
`localeCollate`	The value to be passed as option *–lc-collate* for initdb (default:*C*)	string
`localeCType`	The value to be passed as option *–lc-ctype* for initdb (default:*C*)	string
`walSegmentSize`	The value in megabytes (1 to 1024) to be passed to the *–wal-segsize* option for initdb (default: empty, resulting in PostgreSQL default: 16MB)	int
`postInitSQL`	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)	[]string
`postInitApplicationSQL`	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)	[]string
`postInitTemplateSQL`	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)	[]string
`import`	Bootstraps the new cluster by importing data from an existing PostgreSQL instance using logical backup (*pg_dump* and *pg_restore*)	*Import
postInitApplicationSQLRefs	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)	*PostInitApplicationSQLRefs

4.43.13 BootstrapPgBaseBackup

BootstrapPgBaseBackup contains the configuration required to take a physical backup of an existing PostgreSQL cluster

Name	Description	Type
`source`	The name of the server of which we need to take a physical backup - *mandatory*	string
database	Name of the database used by the application. Default: *app*. - *mandatory*	string
`owner`	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the *database* key. - *mandatory*	string
`secret`	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	*LocalObjectReference

4.43.14 BootstrapRecovery

BootstrapRecovery contains the configuration required to restore the backup with the specified name and, after having changed the password with the one chosen for the superuser, will use it to bootstrap a full cluster cloning all the instances from the restored primary. Refer to the Bootstrap page of the documentation for more information.

Name	Description	Type
`backup`	The backup we need to restore	*BackupSource
`source`	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	string
recoveryTarget	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	*RecoveryTarget
`database`	Name of the database used by the application. Default: *app*. - *mandatory*	string
`owner`	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the *database* key. - *mandatory*	string
`secret`	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	*LocalObjectReference

4.43.15 CertificatesConfiguration

CertificatesConfiguration contains the needed configurations to handle server certificates.

Name	Description	Type
`serverCASecret`	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. 	string

continues on next page

Table 21 – continued from previous page

Name	Description	Type
`serverTLSSecret`	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.	string
replicationTLSSecret	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.	string
`clientCASecret`	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. 	string
`serverAltDNSNames`	The list of the server alternative DNS names to be added to the generated server TLS certificates, when required.	[]string

4.43.16 CertificatesStatus

CertificatesStatus contains configuration certificates and related expiration dates.

Name	Description	Type
expirations	Expiration dates for all certificates.	map[string]string

4.43.17 Cluster

Cluster is the Schema for the PostgreSQL API

Name	Description	Type
metadata		metav1.ObjectMeta
`spec`	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	*ClusterSpec*

continues on next page

Table 23 – continued from previous page

Name	Description	Type
`status`	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	*ClusterStatus*

4.43.18 ClusterList

ClusterList contains a list of Cluster

Name	Description	Type
metadata	Standard list metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	metav1.ListMeta
`items`	List of clusters - *mandatory*	[]Cluster

4.43.19 ClusterSpec

ClusterSpec defines the desired state of Cluster

Name	Description	Type
`description`	Description of this PostgreSQL cluster	string
`inheritedMetadata`	Metadata that will be inherited by all objects related to the Cluster	*EmbeddedObjectMetadata
`imageName`	Name of the container image, supporting both tags (*<image>:<tag>*) and digests for deterministic and repeatable deployments (*<image>:<tag>@sha256:<digestValue>*)	string
`imagePullPolicy`	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/containers/images#updating-images	corev1.PullPolicy
`postgresUID`	The UID of the *postgres* user inside the image, defaults to *26*	int64
`postgresGID`	The GID of the *postgres* user inside the image, defaults to *26*	int64
`instances`	Number of instances required in the cluster - *mandatory*	int
`minSyncReplicas`	Minimum number of instances required in synchronous replication with the primary. Undefined or 0 allow writes to complete when no standby is available.	int

continues on next page

Table 25 – continued from previous page

Name	Description	Type
`maxSyncReplicas`	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.	int
`postgresql`	Configuration of the PostgreSQL server	*PostgresConfiguration*
`replicationSlots`	Replication slots management configuration	*ReplicationSlotsConfiguration
`bootstrap`	Instructions to bootstrap this cluster	*BootstrapConfiguration
`replica`	Replica cluster configuration	*ReplicaClusterConfiguration
`superuserSecret`	The secret containing the superuser password. If not defined a new secret will be created with a randomly generated password	*LocalObjectReference
`enableSuperuserAccess`	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*. Enabled by default.	*bool
`certificates`	The configuration for the CA and related certificates	*CertificatesConfiguration
`imagePullSecrets`	The list of pull secrets to be used to pull the images	[]LocalObjectReference
`storage`	Configuration of the storage of the instances	*StorageConfiguration*
`serviceAccountTemplate`	Configure the generation of the service account	*ServiceAccountTemplate
`walStorage`	Configuration of the storage for PostgreSQL WAL (Write-Ahead Log)	*StorageConfiguration
`startDelay`	The time in seconds that is allowed for a PostgreSQL instance to successfully start up (default 30)	int32
`stopDelay`	The time in seconds that is allowed for a PostgreSQL instance to gracefully shutdown (default 30)	int32
`switchoverDelay`	The time in seconds that is allowed for a primary PostgreSQL instance to gracefully shutdown during a switchover. Default value is 40000000, greater than one year in seconds, big enough to simulate an infinite delay	int32
`failoverDelay`	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	int32
`affinity`	Affinity/Anti-affinity rules for Pods	*AffinityConfiguration*
`resources`	Resources requirements of every generated Pod. Please refer to https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ for more information.	corev1.ResourceRequirements
`primaryUpdateStrategy`	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*)	PrimaryUpdateStrategy

continues on next page

Table 25 – continued from previous page

Name	Description	Type
`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* - default) or in-place (*restart*)	PrimaryUpdateMethod
`backup `	The configuration to be used for backups	*BackupConfiguration
`nodeMaintenanceWindow `	Define a maintenance window for the Kubernetes nodes	*NodeMaintenanceWindow
`monitoring `	The configuration of the monitoring infrastructure of this cluster	*MonitoringConfiguration
`externalClusters `	The list of external clusters which are used in the configuration	[]ExternalCluster
`logLevel `	The instances' log level, one of the following values: error, warning, info (default), debug, trace	string
projectedVolumeTemplate	Template to be used to define projected volumes, projected volumes will be mounted under */projected* base folder	*corev1.ProjectedVolumeSource
`env `	Env follows the Env format to pass environment variables to the pods created in the cluster	[]corev1.EnvVar
`envFrom `	EnvFrom follows the EnvFrom format to pass environment variables sources to the pods to be used by Env	[]corev1.EnvFromSource

4.43.20 ClusterStatus

ClusterStatus defines the observed state of Cluster

Name	Description	Type
`instances `	The total number of PVC Groups detected in the cluster. It may differ from the number of existing instance pods.	int
`readyInstances `	The total number of ready instances in the cluster. It is equal to the number of ready instance pods.	int
`instancesStatus `	InstancesStatus indicates in which status the instances are	map[utils.PodStatus][]string
`instancesReportedState `	The reported state of the instances during the last reconciliation loop	map[PodName]InstanceReportedState
`timelineID `	The timeline of the Postgres cluster	int
`topology `	Instances topology.	*Topology*
`latestGeneratedNode `	ID of the latest generated node (used to avoid node name clashing)	int
`currentPrimary `	Current primary instance	string
`targetPrimary `	Target primary instance, this is different from the previous one during a switchover or a failover	string
`pvcCount `	How many PVCs have been created by this cluster	int32
`jobCount `	How many Jobs have been created by this cluster	int32
`danglingPVC `	List of all the PVCs created by this cluster and still available which are not attached to a Pod	[]string
`resizingPVC `	List of all the PVCs that have ResizingPVC condition.	[]string

continues on next page

Table 26 – continued from previous page

Name	Description	Type
`initializingPVC`	List of all the PVCs that are being initialized by this cluster	[]string
`healthyPVC`	List of all the PVCs not dangling nor initializing	[]string
`unusablePVC`	List of all the PVCs that are unusable because another PVC is missing	[]string
`writeService`	Current write pod	string
`readService`	Current list of read pods	string
`phase`	Current phase of the cluster	string
`phaseReason`	Reason for the current phase	string
`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	*SecretsResourceVersion*
`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	*ConfigMapResourceVersion*
`certificates`	The configuration for the CA and related certificates, initialized with defaults.	*CertificatesStatus*
`firstRecoverabilityPoint`	The first recoverability point, stored as a date in RFC3339 format	string
`lastSuccessfulBackup`	Stored as a date in RFC3339 format	string
`lastFailedBackup`	Stored as a date in RFC3339 format	string
`cloudNativePGCommitHash`	The commit hash number of which this operator running	string
`currentPrimaryTimestamp`	The timestamp when the last actual promotion to primary has occurred	string
currentPrimaryFailingSinceTimestamp	The timestamp when the primary was detected to be unhealthy This field is reported when spec.failoverDelay is populated or during online upgrades	string
`targetPrimaryTimestamp`	The timestamp when the last request for a new primary has occurred	string
`poolerIntegrations`	The integration needed by poolers referencing the cluster	*PoolerIntegrations
`cloudNativePGOperatorHash`	The hash of the binary of the operator	string
`onlineUpdateEnabled`	OnlineUpdateEnabled shows if the online upgrade is enabled inside the cluster	bool
`azurePVCUpdateEnabled`	AzurePVCUpdateEnabled shows if the PVC online upgrade is enabled for this cluster	bool
`conditions`	Conditions for cluster object	[]metav1.Condition
`instanceNames`	List of instance names in the cluster	[]string

4.43.21 ConfigMapKeySelector

ConfigMapKeySelector contains enough information to let you locate the key of a ConfigMap

Name	Description	Type
key	The key to select - *mandatory*	string

4.43.22 ConfigMapResourceVersion

ConfigMapResourceVersion is the resource versions of the secrets managed by the operator

Name	Description	Type
metrics	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	map[string]string

4.43.23 DataBackupConfiguration

DataBackupConfiguration is the configuration of the backup of the data directory

Name	Description	Type
`compression `	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*.	CompressionType
`encryption `	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*	EncryptionType
immediateCheckpoint	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.	bool
`jobs `	The number of parallel jobs to be used to upload the backup, defaults to 2	*int32

4.43.24 EmbeddedObjectMetadata

EmbeddedObjectMetadata contains metadata to be inherited by all resources related to a Cluster

Name	Description	Type
`labels `		map[string]string
annotations		map[string]string

4.43.25 ExternalCluster

ExternalCluster represents the connection parameters to an external cluster which is used in the other sections of the configuration

Name	Description	Type
`name `	The server name, required - *mandatory*	string
connectionParameters	The list of connection parameters, such as dbname, host, username, etc	map[string]string
`sslCert `	The reference to an SSL certificate to be used to connect to this instance	*corev1.SecretKeySelector
`sslKey `	The reference to an SSL private key to be used to connect to this instance	*corev1.SecretKeySelector
`sslRootCert `	The reference to an SSL CA public key to be used to connect to this instance	*corev1.SecretKeySelector
`password `	The reference to the password to be used to connect to the server	*corev1.SecretKeySelector
`barmanObjectStore `	The configuration for the barman-cloud tool suite	*BarmanObjectStoreConfiguration

4.43.26 GoogleCredentials

GoogleCredentials is the type for the Google Cloud Storage credentials. This needs to be specified even if we run inside a GKE environment.

Name	Description	Type
`gkeEnvironment `	If set to true, will presume that it's running inside a GKE environment, default to false. - *mandatory*	bool
applicationCredentials	The secret containing the Google Cloud Storage JSON file with the credentials	*SecretKeySelector

4.43.27 Import

Import contains the configuration to init a database from a logic snapshot of an externalCluster

Name	Description	Type
`source `	The source of the import - *mandatory*	*ImportSource*
`type `	The import type. Can be *microservice* or *monolith*. - *mandatory*	SnapshotType
`databases `	The databases to import - *mandatory*	[]string
`roles `	The roles to import	[]string
postImportApplicationSQL	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.	[]string

4.43.28 ImportSource

ImportSource describes the source for the logical snapshot

Name	Description	Type
externalCluster	The name of the externalCluster used for import - *mandatory*	string

4.43.29 InstanceID

InstanceID contains the information to identify an instance

Name	Description	Type
`podName `	The pod name	string
ContainerID	The container ID	string

4.43.30 InstanceReportedState

InstanceReportedState describes the last reported state of an instance during a reconciliation loop

Name	Description	Type
`isPrimary `	indicates if an instance is the primary one - *mandatory*	bool
timeLineID	indicates on which TimelineId the instance is	int

4.43.31 LDAPBindAsAuth

LDAPBindAsAuth provides the required fields to use the bind authentication for LDAP

Name	Description	Type
prefix	Prefix for the bind authentication option	string
suffix	Suffix for the bind authentication option	string

4.43.32 LDAPBindSearchAuth

LDAPBindSearchAuth provides the required fields to use the bind+search LDAP authentication process

Name	Description	Type
`baseDN `	Root DN to begin the user search	string
`bindDN `	DN of the user to bind to the directory	string
`bindPassword `	Secret with the password for the user to bind to the directory	*corev1.SecretKeySelector
searchAttribute	Attribute to match against the username	string
`searchFilter `	Search filter to use when doing the search+bind authentication	string

4.43.33 LDAPConfig

LDAPConfig contains the parameters needed for LDAP authentication

Name	Description	Type
`server `	LDAP hostname or IP address	string
`port `	LDAP server port	int
`scheme `	LDAP schema to be used, possible options are *ldap* and *ldaps*	LDAPScheme
`tls `	Set to 'true' to enable LDAP over TLS. 'false' is default	bool
`bindAsAuth `	Bind as authentication configuration	*LDAPBindAsAuth
bindSearchAuth	Bind+Search authentication configuration	*LDAPBindSearchAuth

4.43.34 LocalObjectReference

LocalObjectReference contains enough information to let you locate a local object with a known type inside the same namespace

Name	Description	Type
name	Name of the referent. - *mandatory*	string

4.43.35 Metadata

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.

Name	Description	Type
`labels `	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	map[string]string
annotations	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	map[string]string

4.43.36 MonitoringConfiguration

MonitoringConfiguration is the type containing all the monitoring configuration for a certain cluster

Name	Description	Type
`disableDefaultQueries `	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.	*bool
customQueriesConfigMap	The list of config maps containing the custom queries	[]ConfigMapKeySelector
`customQueriesSecret `	The list of secrets containing the custom queries	[]SecretKeySelector
`enablePodMonitor `	Enable or disable the *PodMonitor*	bool

4.43.37 NodeMaintenanceWindow

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.

Name	Description	Type
inProgress	Is there a node maintenance activity in progress? - *mandatory*	bool
`reusePVC `	Reuse the existing PVC (wait for the node to come up again) or not (recreate it elsewhere - when *instances* >1) - *mandatory*	*bool

4.43.38 PgBouncerIntegrationStatus

PgBouncerIntegrationStatus encapsulates the needed integration for the pgbouncer poolers referencing the cluster

Name	Description	Type
secrets		[]string

4.43.39 PgBouncerSecrets

PgBouncerSecrets contains the versions of the secrets used by pgbouncer

Name	Description	Type
authQuery	The auth query secret version	*SecretVersion*

4.43.40 PgBouncerSpec

PgBouncerSpec defines how to configure PgBouncer

Name	Description	Type
`poolMode`	The pool mode - *mandatory*	PgBouncerPoolMode
authQuerySecret	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_shadow WHERE username=$1") has to be specified and no automatic CNPG Cluster integration will be triggered.	*LocalObjectReference
`authQuery`	The query that will be used to download the hash of the password of a certain user. Default: "SELECT username, passwd FROM user_search($1)". In case it is specified, also an AuthQuerySecret has to be specified and no automatic CNPG Cluster integration will be triggered.	string
`parameters`	Additional parameters to be passed to PgBouncer - please check the CNPG documentation for a list of options you can configure	map[string]string
`pg_hba`	PostgreSQL Host Based Authentication rules (lines to be appended to the pg_hba.conf file)	[]string
`paused`	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.	*bool

4.43.41 PodTemplateSpec

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

Name	Description	Type
metadata	Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata	Defining cluster's metadata
`spec`	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	corev1.PodSpec

4.43.42 Pooler

Pooler is the Schema for the poolers API

Name	Description	Type
metadata		metav1.ObjectMeta
`spec`		*PoolerSpec*
`status`		*PoolerStatus*

4.43.43 PoolerIntegrations

PoolerIntegrations encapsulates the needed integration for the poolers referencing the cluster

Name	Description	Type
pgBouncerIntegration		*PgBouncerIntegrationStatus*

4.43.44 PoolerList

PoolerList contains a list of Pooler

Name	Description	Type
metadata		metav1.ListMeta
`items`	• *mandatory*	[]Pooler

4.43.45 PoolerSecrets

PoolerSecrets contains the versions of all the secrets used

Name	Description	Type
`serverTLS`	The server TLS secret version	*SecretVersion*
`serverCA`	The server CA secret version	*SecretVersion*
`clientCA`	The client CA secret version	*SecretVersion*
pgBouncerSecrets	The version of the secrets used by PgBouncer	*PgBouncerSecrets

4.43.46 PoolerSpec

PoolerSpec defines the desired state of Pooler

Name	Description	Type
`cluster`	This is the cluster reference on which the Pooler will work. Pooler name should never match with any cluster name within the same namespace. - *mandatory*	*LocalObjectReference*
`type`	Which instances we must forward traffic to? - *mandatory*	PoolerType
instances	The number of replicas we want - *mandatory*	int32
`template`	The template of the Pod to be created	*PodTemplateSpec
pgbouncer	The PgBouncer configuration - *mandatory*	*PgBouncerSpec

4.43.47 PoolerStatus

PoolerStatus defines the observed state of Pooler

Name	Description	Type
`secrets`	The resource version of the config object	*PoolerSecrets
instances	The number of pods trying to be scheduled	int32

4.43.48 PostInitApplicationSQLRefs

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

Name	Description	Type
`secretRefs`	SecretRefs holds a list of references to Secrets	[]SecretKeySelector
configMapRefs	ConfigMapRefs holds a list of references to ConfigMaps	[]ConfigMapKeySelector

4.43.49 PostgresConfiguration

PostgresConfiguration defines the PostgreSQL configuration

Name	Description	Type
`parameters`	PostgreSQL configuration options (postgresql.conf)	map[string]string
`pg_hba`	PostgreSQL Host Based Authentication rules (lines to be appended to the pg_hba.conf file)	[]string
syncReplicaElectionConstraint	Requirements to be met by sync replicas. This will affect how the "synchronous_standby_names" parameter will be set up.	*SyncReplicaElectionConstraints*
`promotionTimeout`	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	int32
`shared_preload_libraries`	Lists of shared preload libraries to add to the default ones	[]string
`ldap`	Options to specify LDAP configuration	*LDAPConfig

4.43.50 RecoveryTarget

RecoveryTarget allows to configure the moment where the recovery process will stop. All the target options except TargetTLI are mutually exclusive.

Name	Description	Type
`backupID`	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.	string
`targetTLI`	The target timeline ("latest" or a positive integer)	string
`targetXID`	The target transaction ID	string
`targetName`	The target name (to be previously created with *pg_create_restore_point*)	string
`targetLSN`	The target LSN (Log Sequence Number)	string
`targetTime`	The target time as a timestamp in the RFC3339 standard	string
targetImmediate	End recovery as soon as a consistent state is reached	*bool
`exclusive`	Set the target to be exclusive (defaults to true)	*bool

4.43.51 ReplicaClusterConfiguration

ReplicaClusterConfiguration encapsulates the configuration of a replica cluster

Name	Description	Type
enabled	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 Replication page of the documentation for more information. - *mandatory*	bool
`source`	The name of the external cluster which is the replication origin - *mandatory*	string

4.43.52 ReplicationSlotsConfiguration

ReplicationSlotsConfiguration encapsulates the configuration of replication slots

Name	Description	Type
highAvailability	Replication slots for high availability configuration	*ReplicationSlotsHAConfiguration
`updateInterval`	Standby will update the status of the local replication slots every *updateInterval* seconds (default 30).	int

4.43.53 ReplicationSlotsHAConfiguration

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.

Name	Description	Type
`enabled`	If enabled, 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 (default), 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. This can only be set at creation time. - *mandatory*	bool
slotPrefix	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_*.	string

4.43.54 RollingUpdateStatus

RollingUpdateStatus contains the information about an instance which is being updated

Name	Description	Type
imageName	The image which we put into the Pod - *mandatory*	string
startedAt	When the update has been started	metav1.Time

4.43.55 S3Credentials

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

Name	Description	Type
`accessKeyId `	The reference to the access key id	*SecretKeySelector
`secretAccessKey `	The reference to the secret access key	*SecretKeySelector
`region `	The reference to the secret containing the region name	*SecretKeySelector
`sessionToken `	The references to the session key	*SecretKeySelector
inheritFromIAMRole	Use the role based authentication without providing explicitly the keys. - *mandatory*	bool

4.43.56 ScheduledBackup

ScheduledBackup is the Schema for the scheduledbackups API

Name	Description	Type
metadata		metav1.ObjectMeta
`spec `	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	*ScheduledBackupSpec*
`status `	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	*ScheduledBackupStatus*

4.43.57 ScheduledBackupList

ScheduledBackupList contains a list of ScheduledBackup

Name	Description	Type
metadata	Standard list metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#types-kinds	metav1.ListMeta
`items `	List of clusters - *mandatory*	[]ScheduledBackup

4.43.58 ScheduledBackupSpec

ScheduledBackupSpec defines the desired state of ScheduledBackup

Name	Description	Type
`suspend `	If this backup is suspended or not	*bool
`immediate `	If the first backup has to be immediately start after creation or not	*bool
`schedule `	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 - *mandatory*	string
`cluster `	The cluster to backup	*LocalObjectReference*
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 	string
`target `	The policy to decide which instance should perform this backup. If empty, it defaults to *cluster.spec.backup.target*. Available options are empty string, which will default to *primary* policy, *primary* to have backups run always on primary instances, *prefer-standby* to have backups run preferably on the most updated standby, if available.	BackupTarget

4.43.59 ScheduledBackupStatus

ScheduledBackupStatus defines the observed state of ScheduledBackup

Name	Description	Type
`lastCheckTime `	The latest time the schedule	*metav1.Time
lastScheduleTime	Information when was the last time that backup was successfully scheduled.	*metav1.Time
nextScheduleTime	Next time we will run a backup	*metav1.Time

4.43.60 SecretKeySelector

SecretKeySelector contains enough information to let you locate the key of a Secret

Name	Description	Type
key	The key to select - *mandatory*	string

4.43.61 SecretVersion

SecretVersion contains a secret name and its ResourceVersion

Name	Description	Type
`name `	The name of the secret	string
version	The ResourceVersion of the secret	string

4.43.62 SecretsResourceVersion

SecretsResourceVersion is the resource versions of the secrets managed by the operator

Name	Description	Type
`superuserSecretVersion `	The resource version of the "postgres" user secret	string
replicationSecretVersion	The resource version of the "streaming_replica" user secret	string
applicationSecretVersion	The resource version of the "app" user secret	string
`caSecretVersion `	Unused. Retained for compatibility with old versions.	string
`clientCaSecretVersion `	The resource version of the PostgreSQL client-side CA secret version	string
`serverCaSecretVersion `	The resource version of the PostgreSQL server-side CA secret version	string
`serverSecretVersion `	The resource version of the PostgreSQL server-side secret version	string
`barmanEndpointCA `	The resource version of the Barman Endpoint CA if provided	string

continues on next page

Table 68 – continued from previous page

Name	Description	Type
`metrics`	A map with the versions of all the secrets used to pass metrics. Map keys are the secret names, map values are the versions	map[string]string

4.43.63 ServiceAccountTemplate

ServiceAccountTemplate contains the template needed to generate the service accounts

Name	Description	Type
metadata	Metadata are the metadata to be used for the generated service account - *mandatory*	Defining cluster's metadata

4.43.64 StorageConfiguration

StorageConfiguration is the configuration of the storage of the PostgreSQL instances

Name	Description	Type
`storageClass`	StorageClass to use for database data (*PGDATA*). Applied after evaluating the PVC template, if available. If not specified, generated PVCs will be satisfied by the default storage class	*string
`size`	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.	string
resizeInUseVolumes	Resize existent PVCs, defaults to true	*bool
`pvcTemplate`	Template to be used to generate the Persistent Volume Claim	*corev1.PersistentVolumeClaimSpec

4.43.65 SyncReplicaElectionConstraints

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.

Name	Description	Type
`enabled`	This flag enables the constraints for sync replicas - *mandatory*	bool
nodeLabelsAntiAffinity	A list of node labels values to extract and compare to evaluate if the pods reside in the same topology or not	[]string

4.43.66 Topology

Topology contains the cluster topology

Name	Description	Type
successfullyExtracted	SuccessfullyExtracted indicates if the topology data was extract. It is useful to enact fallback behaviors in synchronous replica election in case of failures	bool
`instances`	Instances contains the pod topology of the instances	map[PodName]PodTopologyLabels

4.43.67 WalBackupConfiguration

WalBackupConfiguration is the configuration of the backup of the WAL stream

Name	Description	Type
compression	Compress a WAL file before sending it to the object store. Available options are empty string (no compression, default), *gzip*, *bzip2* or *snappy*.	CompressionType
`encryption`	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*	EncryptionType
maxParallel	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.	int

4.44 Supported releases

- This page lists the status, timeline and policy for currently supported releases of CloudNativePG*.

We support the latest two minor releases, and we aim to create a new minor release every two months. Each release is supported until the declared "End of Life" date in the table below , which considers an additional month of assistance to allow for 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 do not contain backward incompatible changes.

- *Support policy*
- *Naming scheme*
- *Support status of CloudNativePG releases*
- *What we mean by support*

4.44.1 Support policy

We produce new builds of CloudNativePG for each commit.

Roughly 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 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, please refer to the

> *What we mean by support* below.

Type	Support Level	Quality and Recommended Use
Development Build	No support	Dangerous, may not be fully reliable. Useful to experiment with.
Minor Release	Support provided until 1 month after the N+2 minor release (ex. 1.15 supported until 1 month after 1.17.0 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 will not 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 CloudNativePG 1.19* .

4.44.2 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` .

4.44.3 Support status of CloudNativePG releases

Version	Currently Supported	Release Date	End of Life	Supported Kubernetes Versions	Tested, but not s
1.19.x	Yes	February 14, 2023	~ July 15, 2023	1.23, 1.24, 1.25, 1.26	1.22
1.18.x	Yes	November 10, 2022	~ May 13, 2023	1.23, 1.24, 1.25, 1.26	1.22
main	No, development only				

The list of supported Kubernetes versions in the above table depends on what the CloudNativePG maintainers think is reasonable to support and to test.

At the moment, the CloudNativePG community does not support any additional Kubernetes distribution, like Red Hat OpenShift (this might change in the future. If it did change, that would be reflected in an official policy written by the CloudNativePG maintainers).

The list of supported Postgres versions in the above table generally depends on what PostgreSQL versions were supported by the Community at the time the minor version was released.

4.44.4 Upcoming releases

Version	Release Date	End of Life	Supported Kubernetes Versions
1.20.0	April 13, 2023	•	•
1.21.0	June 15, 2023	•	•

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.

4.44.5 Old releases

Version	Release Date	End of Life	Compatible Kubernetes Versions
1.15.x	April 21, 2022	October 6, 2022	1.21, 1.22, 1.23
1.16.x	July 7, 2022	December 21, 2022	1.22, 1.23, 1.24
1.17.x	September 6, 2022	March 20, 2023	1.22, 1.23, 1.24

4.44.6 What we mean by support

Our support window is roughly five months for each release branch (latest two minor releases, plus an additional month), given that we produce a new final release every two months.

In the below diagram, `release-1.16` is an example of a release branch.

For example, imagining that the latest release is `v1.16.0` , you can expect support for both `v1.16.0` and `v1.15.0` .

Only the last patch release of each branch is supported.

```
- -----+----------------------------------------------> main (trunk development)
       \             \
        \             \
         \             \             v1.16.0
          \             \            Jul 7, 2022                    ^
           \             \----------+---------------> release-1.16  |
            \                                                       | SUPPORTED
             \                                                      | RELEASES
              \   v1.15.0                                           | = the two
               \  Apr 21, 2022                                      |   last
                +-------------------+---------------> release-1.15  |   releases
                                                                    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 back-port important bug fixes — including security fixes - to all currently supported releases. Before back-porting a patch, we ask ourselves: *"Does this back-port improve ``CloudNativePG``, bearing in mind that we really value stability for already-released versions?"*

If you are looking for professional support, please refer to the
Missing tablespace support . The vendors listed there might provide service level

agreements that included extended support timeframes.

4.45 Release notes

History of user-visible changes for CloudNativePG, classified for each minor release.

- *Release notes for CloudNativePG 1.19*
- *Release notes for CloudNativePG 1.18*

For information on the community support policy for CloudNativePG, please refer to *Supported releases* .

Older releases:

- *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.

4.46 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.

4.46.1 Version 1.19.1

Releasedate: 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 betwen 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.

4.46.2 Version 1.19.0

Releasedate: 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)

4.47 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.

4.47.1 Version 1.18.3

Releasedate: 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 betwen 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)

4.47.2 Version 1.18.2

Releasedate: 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)

4.47.3 Version 1.18.1

Releasedate: 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)

4.47.4 Version 1.18.0

Releasedate: Nov 10, 2022

Features:

- **Cluster-managedphysicalreplicationslotsforHighAvailability** : automatically manages physical replication slots for each hot standby replica in the High Availability cluster, both in the primary and the standby (#740)
- **Postgresclusterhibernation** : 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)

4.48 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. **NoneoftheversionslistedhereexistsforCloudNativePG.**

4.48.1 Version 1.14.0

Releasedate: 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

4.48.2 Version 1.13.0

Releasedate: 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

4.48.3 Version 1.12.0

Releasedate: 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

4.48.4 Version 1.11.0

Releasedate: 15 December 2021

Features:

- **ParallelWALarchivingandrestore:** 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
- **Defaultsetofmetricsformonitoring:** 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

4.48.5 Version 1.10.0

Releasedate: 11 November 2021

Features:

- **ConnectionPoolingwithPgBouncer** : 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
- **BackupRetentionPolicies** : support definition of recovery window retention policies for backups (e.g. '30d' to ensure a recovery window of 30 days)
- **In-Placeupdatesoftheoperator** : 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

4.48.6 Version 1.9.2

Releasedate: 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

4.48.7 Version 1.9.1

Releasedate: 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

4.48.8 Version 1.9.0

Releasedate: 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

4.48.9 Version 1.8.0

Releasedate: 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

4.48.10 Version 1.7.1

Releasedate: 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

4.48.11 Version 1.7.0

Releasedate: 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

4.48.12 Version 1.6.0

Releasedate: 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.

4.48.13 Version 1.5.1

Releasedate: 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.

4.48.14 Version 1.5.0

Releasedate: 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

4.48.15 Version 1.4.0

Releasedate: 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

4.48.16 Version 1.3.0

Releasedate: 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

4.48.17 Version 1.2.1

Releasedate: 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)

4.48.18 Version 1.2.0

Releasedate: 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

4.48.19 Version 1.1.0

Releasedate: 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

4.48.20 Version 1.0.0

Releasedate: 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

4.49 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.

4.49.1 Version 1.17.5

Releasedate: March 20, 2023

> **Warning:** Version 1.17 has reached End-of-Life (EOL). Version 1.17.5 is the last that will be released for the 1.17 minor version.

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)

4.49.2 Version 1.17.4

Releasedate: 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)

4.49.3 Version 1.17.3

Releasedate: 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)

4.49.4 Version 1.17.2

Releasedate: 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)

4.49.5 Version 1.17.1

Releasedate: 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)

4.49.6 Version 1.17.0

Releasedate: Sep 6, 2022 (minor release)

Features:

- **SeparatevolumeforWALfiles:** 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)

4.50 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.

4.50.1 Version 1.16.5

Releasedate: Dec 21, 2022

> **Warning:** Version 1.16 has reached End-of-Life (EOL). Version 1.16.5 is the last that will be released for the 1.16 minor version.

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)

4.50.2 Version 1.16.4

Releasedate: 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)

4.50.3 Version 1.16.3

Releasedate: 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)

4.50.4 Version 1.16.2

Releasedate: 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)

4.50.5 Version 1.16.1

Releasedate: 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

4.50.6 Version 1.16.0

Releasedate: Jul 7, 2022 (minor release)

Features:

- **OfflinedataimportandmajorupgradesforPostgreSQL:** 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

4.51 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:** Version 1.15 has reached End-of-Life (EOL). Version 1.15.5 is the last that will be released for the 1.15 minor version.

4.51.1 Version 1.15.5

Releasedate: 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)

4.51.2 Version 1.15.4

Releasedate: 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)

4.51.3 Version 1.15.3

Releasedate: 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

4.51.4 Version 1.15.2

Releasedate: 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

4.51.5 Version 1.15.1

Releasedate: 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`

4.51.6 Version 1.15.0

Releasedate: 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
- **LDAPauthentication:** 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 .