

CloudNativePG

Release 1.27.0

EnterpriseDB Corporation

Sep 03, 2025

Contents

1 CloudNativePG **1**
- 1.1 Supported Kubernetes distributions . . . 2
- 1.2 Container images . . . 2
  - 1.2.1 Operator . . . 2
  - 1.2.2 Operands . . . 2
- 1.3 Main features . . . 2
- 1.4 About this guide . . . 4

2 Before You Start **5**
- 2.1 Kubernetes terminology . . . 5
- 2.2 PostgreSQL terminology . . . 6
- 2.3 Cloud terminology . . . 7
- 2.4 What to do next . . . 7

3 Use cases **9**
- 3.1 Case 1: Applications inside Kubernetes . . . 9
- 3.2 Case 2: Applications outside Kubernetes . . . 10

4 Architecture **13**
- 4.1 Synchronizing the state . . . 13
- 4.2 Kubernetes architecture . . . 14
  - 4.2.1 Multi-availability zone Kubernetes clusters . . . 14
  - 4.2.2 Single availability zone Kubernetes clusters . . . 15
  - 4.2.3 Reserving nodes for PostgreSQL workloads . . . 16
  - 4.2.4 Proposed node label . . . 17
  - 4.2.5 Proposed node taint . . . 17
- 4.3 PostgreSQL architecture . . . 17
  - 4.3.1 Read-write workloads . . . 19
  - 4.3.2 Read-only workloads . . . 19
- 4.4 Deployments across Kubernetes clusters . . . 19

5 Installation and upgrades **23**
- 5.1 Installation on Kubernetes . . . 23
  - 5.1.1 Directly using the operator manifest . . . 23
  - 5.1.2 Using the `cnpg` plugin for `kubectl` . . . 23
  - 5.1.3 Testing the latest development snapshot . . . 24
  - 5.1.4 Using the Helm Chart . . . 24

5.1.5 Using OLM . . . 24
5.2 Details about the deployment . . . 24
5.3 Upgrades . . . 25
5.3.1 Spread Upgrades . . . 26
5.3.2 In-place updates of the instance manager . . . 26
5.3.3 Compatibility among versions . . . 26
5.3.4 Upgrading to 1.28.0 or 1.27.x . . . 27
5.3.5 Upgrading to 1.27.0 or 1.26.1 . . . 27
5.3.6 Upgrading to 1.26 from a previous minor version . . . 27
5.3.7 Deprecation of backup metrics and fields in the `Cluster .status` . . . 28
5.3.8 Declarative Hibernation in the `cnpg` plugin . . . 29
5.3.9 Upgrading to 1.25 from a previous minor version . . . 29
5.3.10 Upgrading to 1.24 from a previous minor version . . . 29
5.3.11 From Replica Clusters to Distributed Topology . . . 29

6 Quickstart **31**
6.1 Part 1: Setup the local Kubernetes playground . . . 31
6.1.1 Minikube . . . 32
6.1.2 Kind . . . 32
6.2 Part 2: Install CloudNativePG . . . 32
6.3 Part 3: Deploy a PostgreSQL cluster . . . 32
6.4 Part 4: Monitor clusters with Prometheus and Grafana . . . 34
6.4.1 Installation . . . 34
6.4.2 Viewing with Prometheus . . . 37
6.4.3 Grafana Dashboard . . . 38

7 Image Catalog **41**
7.1 CloudNativePG Catalogs . . . 42
7.1.1 PostgreSQL Container Images . . . 42
7.1.2 PostGIS Container Images . . . 43

8 Bootstrap **45**
8.1 The `bootstrap` section . . . 46
8.2 The `externalClusters` section . . . 46
8.2.1 Password files . . . 47
8.3 Bootstrap an empty cluster (`initdb`) . . . 48
8.3.1 Passing Options to `initdb` . . . 49
8.3.2 Executing Queries After Initialization . . . 51
8.4 Bootstrap from another cluster . . . 52
8.4.1 Bootstrap from a backup (`recovery`) . . . 53
8.4.2 Bootstrap from a live cluster (`pg_basebackup`) . . . 53
8.4.3 Requirements . . . 54
8.4.4 About the replication user . . . 54
8.4.5 Username/Password authentication . . . 55
8.4.6 TLS certificate authentication . . . 55
8.4.7 Configure the application database . . . 56
8.4.8 Current limitations . . . 57

9 Importing Postgres databases **59**
9.1 How it works . . . 60
9.2 The `microservice` type . . . 60
9.3 The `monolith` type . . . 62
9.4 A practical example . . . 64
9.5 Import optimizations . . . 66
9.6 Customizing `pg_dump` and `pg_restore` Behavior . . . 66

9.7 Online Import and Upgrades . . . 67

10 Security **69**
10.1 Code . . . 69
10.2 Container . . . 70
10.2.1 Image Signatures . . . 70
10.2.2 Attestations . . . 71
10.2.3 Guidelines and Frameworks for Container Security . . . 71
10.3 Cluster . . . 71
10.3.1 Role Based Access Control (RBAC) . . . 71
10.3.2 Deployments and `ClusterRole` Resources . . . 72
10.3.3 Why Are ClusterRole Permissions Needed? . . . 73
10.3.4 Calls to the API server made by the instance manager . . . 73
10.3.5 Pod and Container Security Contexts . . . 74
10.3.6 Security Context Constraints . . . 75
10.3.7 Restricting Pod access using AppArmor . . . 75
10.3.8 Network Policies . . . 75
10.3.9 Exposed Ports . . . 76
10.3.10 PostgreSQL . . . 76
10.3.11 Storage . . . 77

11 Postgres instance manager **79**
11.1 Startup Probe . . . 79
11.1.1 Startup Probe Strategy . . . 80
11.2 Liveness Probe . . . 81
11.2.1 Primary Isolation . . . 82
11.3 Readiness Probe . . . 83
11.3.1 Readiness Probe Strategy . . . 84
11.4 Shutdown control . . . 84
11.4.1 Shutdown of the primary during a switchover . . . 85
11.5 Failover . . . 85
11.6 Disk Full Failure . . . 85

12 Scheduling **87**
12.1 Pod Affinity and Anti-Affinity . . . 87
12.1.1 Requiring Pod Anti-Affinity . . . 88
12.1.2 Topology Considerations . . . 89
12.1.3 Disabling Anti-Affinity Policies . . . 89
12.1.4 Fine-Grained Control with Custom Rules . . . 89
12.2 Node selection through `nodeSelector` . . . 89
12.3 Tolerations . . . 89
12.4 Isolating PostgreSQL workloads . . . 90

13 Resource management **91**

14 Failure Modes **95**
14.1 Self-Healing . . . 95
14.1.1 Primary Failure . . . 95
14.1.2 Standby Failure . . . 96
14.2 Manual Intervention . . . 96
14.2.1 Disabling Reconciliation . . . 96

15 Rolling Updates **97**
15.1 Automated updates (`unsupervised`) . . . 98
15.2 Manual updates (`supervised`) . . . 98

16 Replication **99**
16.1 Application-level replication . . . 99
16.2 A very mature technology . . . 99
16.3 Streaming replication support . . . 100
16.3.1 Continuous backup integration . . . 101
16.4 Synchronous Replication . . . 101
16.4.1 Quorum-based Synchronous Replication . . . 101
16.4.2 Example . . . 101
16.4.3 Migrating from Deprecated Synchronous Replication Implementation . . . 102
16.4.4 Priority-based Synchronous Replication . . . 102
16.4.5 Controlling `synchronous_standby_names` Content . . . 103
16.4.6 Examples . . . 103
16.4.7 Data Durability and Synchronous Replication . . . 104
16.4.8 Required Data Durability . . . 104
16.4.9 Preferred Data Durability . . . 105
16.5 Synchronous Replication (Deprecated) . . . 106
16.5.1 Select nodes for synchronous replication . . . 107
16.6 Replication slots . . . 108
16.6.1 Replication slots for High Availability . . . 109
16.6.2 User-Defined Replication slots . . . 109
16.6.3 Synchronization frequency . . . 110
16.6.4 Logical Decoding Slot Synchronization . . . 110
16.6.5 Behavior on PostgreSQL 17 and later . . . 111
16.6.6 Behavior on PostgreSQL 16 and earlier . . . 112
16.6.7 Capping the WAL size retained for replication slots . . . 112
16.6.8 Monitoring replication slots . . . 112

17 Logical Replication **113**
17.1 Publications . . . 114
17.1.1 Required Fields in the `Publication` Manifest . . . 115
17.1.2 Reconciliation and Status . . . 115
17.1.3 Removing a publication . . . 115
17.2 Subscriptions . . . 116
17.2.1 Required Fields in the `Subscription` Manifest . . . 117
17.2.2 Reconciliation and Status . . . 117
17.2.3 Removing a Subscription . . . 117
17.2.4 Resilience to Failovers . . . 118
17.3 Limitations . . . 118
17.3.1 Addressing Schema Replication . . . 118
17.3.2 Handling Sequences . . . 118
17.4 Example of live migration and major Postgres upgrade with logical replication . . . 118
17.4.1 Step 1: Setting Up the Publisher (`freddie`) . . . 119
17.4.2 Step 2: Setting Up the Subscriber (`king`) . . . 119

18 Backup **123**
18.1 Main Concepts . . . 124
18.1.1 WAL archive . . . 124
18.1.2 Cold and Hot backups . . . 124
18.2 Comparing Available Backup Options: Object Stores vs Volume Snapshots . . . 125
18.2.1 Object Store–Based Backups . . . 125
18.2.2 Volume Snapshots . . . 125
18.2.3 Choosing Between the Two . . . 125
18.2.4 Comparison Summary . . . 126
18.3 Backup Methods . . . 126

18.4 Backup from a Standby . . . 127
18.4.1 How It Works . . . 127
18.4.2 Forcing Backup on the Primary . . . 127
18.4.3 Overriding the Cluster-Wide Target . . . 128
18.5 Retention Policies . . . 128

19 WAL archiving **129**
19.1 Plugin-Based Architecture . . . 129
19.2 Supported Plugins . . . 129
19.3 Deprecation Notice: Native Barman Cloud . . . 129
19.4 About the archive timeout . . . 130

20 Recovery **131**
20.1 Recovery from an Object Store with the Barman Cloud Plugin . . . 132
20.2 Recovery from `VolumeSnapshot` Objects . . . 133
20.3 Recovery from a `Backup` object . . . 134
20.4 Additional Considerations . . . 135
20.5 Point in time recovery (PITR) . . . 135
20.5.1 PITR from an object store . . . 136
20.5.2 Point-in-Time Recovery (PITR) from `VolumeSnapshot` Objects . . . 137
20.5.3 Recovery targets . . . 138
20.6 Configure the application database . . . 139
20.7 How recovery works under the hood . . . 140
20.8 Restoring into a Cluster with a Backup Section . . . 141

21 Service Management **143**
21.1 Disabling Default Services . . . 144
21.2 Adding Your Own Services . . . 144
21.2.1 About Exposing Postgres Services . . . 145

22 PostgreSQL Configuration **147**
22.1 The `postgresql` section . . . 148
22.1.1 Write-Ahead Log Level . . . 150
22.1.2 Replication Settings . . . 150
22.1.3 Log control settings . . . 150
22.1.4 Shared Preload Libraries . . . 150
22.1.5 Managed extensions . . . 151
22.1.6 Enabling `auto_explain` . . . 152
22.1.7 Enabling `pg_stat_statements` . . . 152
22.1.8 Enabling `pgaudit` . . . 152
22.2 The `pg_hba` section . . . 153
22.3 The `pg_ident` section . . . 155
22.4 Dynamic Shared Memory settings . . . 156
22.4.1 POSIX shared memory . . . 156
22.4.2 System V shared memory . . . 156
22.5 Fixed parameters . . . 157

23 PostgreSQL Role Management **161**
23.1 Password management . . . 163
23.1.1 Password expiry, `VALID UNTIL` . . . 164
23.1.2 Password hashed . . . 164
23.2 Unrealizable role configurations . . . 164
23.3 Status of managed roles . . . 165

24 PostgreSQL Database Management **167**

24.1 Key Concepts . . . 167
24.1.1 Scope of Management . . . 167
24.1.2 Declarative `Database` Manifest . . . 167
24.1.3 Required Fields in the `Database` Manifest . . . 168
24.2 Reserved Database Names . . . 168
24.3 Reconciliation and Status . . . 168
24.4 Deleting a Database . . . 169
24.4.1 Deleting via `delete` Reclaim Policy . . . 169
24.4.2 Declaratively Setting `ensure: absent` . . . 170
24.5 Managing Extensions in a Database . . . 170
24.6 Managing Schemas in a Database . . . 171
24.7 Managing Foreign Data Wrappers (FDWs) In a Database . . . 172
24.8 Limitations and Caveats . . . 173
24.8.1 Renaming a database . . . 173
24.8.2 Creating vs. Altering a Database . . . 173
24.8.3 Replica Clusters . . . 173
24.8.4 Conflict Resolution . . . 174
24.8.5 Postgres Version Differences . . . 174
24.8.6 Manual Changes . . . 174

25 Tablespaces **175**
25.1 Declarative tablespaces . . . 175
25.2 Using declarative tablespaces . . . 176
25.3 Storage classes and tablespaces . . . 177
25.4 Tablespace ownership . . . 178
25.5 Backup and recovery . . . 179
25.6 Replica clusters . . . 180
25.7 Temporary tablespaces . . . 180
25.8 kubectl plugin support . . . 181
25.9 Limitations . . . 181

26 Operator configuration **183**
26.1 Available options . . . 184
26.2 Defining an operator config map . . . 185
26.3 Defining an operator secret . . . 185
26.4 Restarting the operator to reload configs . . . 186
26.5 pprof HTTP Server . . . 186

27 Instance pod configuration **189**
27.1 Projected volumes . . . 189
27.2 Ephemeral volumes . . . 190
27.2.1 Volume Claim Template for Temporary Storage . . . 191
27.2.2 Volume for shared memory . . . 191
27.3 Environment variables . . . 191

28 Storage **193**
28.1 Backup and recovery . . . 194
28.2 Benchmarking CloudNativePG . . . 194
28.3 Encryption at rest . . . 194
28.4 Persistent Volume Claim (PVC) . . . 195
28.5 Configuration via a storage class . . . 195
28.6 Configuration via a PVC template . . . 195
28.7 Volume for WAL . . . 196
28.8 Volumes for tablespaces . . . 197
28.9 Volume expansion . . . 197

28.9.1 Using the volume expansion Kubernetes feature . . . 197
28.9.2 Re-creating storage . . . 197
28.10 Static provisioning of persistent volumes . . . 198
28.11 Block storage considerations (Ceph/Longhorn) . . . 199

29 Labels and annotations **201**
29.1 Predefined labels . . . 201
29.2 Predefined annotations . . . 202
29.3 Prerequisites . . . 204
29.4 Defining cluster's metadata . . . 204
29.5 Current limitations . . . 205

30 Monitoring **207**
30.1 Monitoring Instances . . . 207
30.1.1 Monitoring with the Prometheus operator . . . 208
30.1.2 Enabling TLS on the Metrics Port . . . 209
30.1.3 Predefined set of metrics . . . 210
30.1.4 User defined metrics . . . 218
30.1.5 Example of a user defined metric . . . 218
30.1.6 Example of a user defined metric with predicate query . . . 219
30.1.7 Example of a user defined metric running on multiple databases . . . 220
30.1.8 Structure of a user defined metric . . . 221
30.1.9 Output of a user defined metric . . . 222
30.1.10 Default set of metrics . . . 223
30.1.11 Differences with the Prometheus Postgres exporter . . . 223
30.2 Monitoring the CloudNativePG operator . . . 223
30.2.1 Monitoring the operator with Prometheus . . . 223
30.3 How to inspect the exported metrics . . . 224
30.3.1 Using port forwarding . . . 224
30.3.2 Using curl . . . 224
30.4 Auxiliary resources . . . 225

31 Logging **227**
31.1 Cluster Logs . . . 228
31.2 Operator Logs . . . 228
31.3 PostgreSQL Logs . . . 228
31.4 PGAudit Logs . . . 229
31.5 Other Logs . . . 231

32 Certificates **233**
32.1 Operator-Managed Mode . . . 234
32.1.1 Server certificates . . . 234
32.1.2 Server CA secret . . . 234
32.1.3 Server TLS secret . . . 234
32.1.4 Server alternative DNS names . . . 234
32.1.5 Client certificates . . . 234
32.1.6 Client CA secret . . . 234
32.1.7 Client `streaming_replica` certificate . . . 235
32.2 User-provided certificates mode . . . 235
32.2.1 Server certificates . . . 235
32.2.2 Example . . . 235
32.2.3 Cert-manager example . . . 236
32.2.4 Client certificate . . . 239
32.2.5 Customizing the `streaming_replica` client certificate . . . 240
32.2.6 Cert-manager example . . . 240

33 Client TLS/SSL connections **245**
33.1 Issuing a new certificate . . . 246
33.2 Testing the connection via a TLS certificate . . . 246
33.3 About TLS protocol versions . . . 248

34 Connecting from an application **249**
34.1 DNS resolution . . . 249
34.2 Environment variables . . . 250
34.3 Secrets . . . 250

35 Connection pooling **251**
35.1 Architecture . . . 251
35.2 Quick start . . . 251
35.3 Pooler resource lifecycle . . . 253
35.4 Security . . . 253
35.4.1 Certificates . . . 253
35.5 Authentication . . . 254
35.6 Pod templates . . . 255
35.7 Service Template . . . 256
35.8 High availability (HA) . . . 257
35.9 PgBouncer configuration options . . . 257
35.10 Monitoring . . . 259
35.11 Logging . . . 267
35.12 Pausing connections . . . 267
35.13 Limitations . . . 268
35.13.1 Single PostgreSQL cluster . . . 268
35.13.2 Controlled configurability . . . 268

36 Replica clusters **269**
36.1 Basic Concepts . . . 269
36.1.1 About PostgreSQL Roles . . . 269
36.1.2 Bootstrapping a Replica Cluster . . . 270
36.1.3 Configuring Replication . . . 270
36.1.4 Defining an External Cluster . . . 271
36.1.5 Backup and Symmetric Architectures . . . 271
36.1.6 Distributed Architecture Flexibility . . . 271
36.2 Setting Up a Replica Cluster . . . 271
36.3 Distributed Topology . . . 272
36.3.1 Planning for a Distributed PostgreSQL Database . . . 272
36.3.2 Demoting a Primary to a Replica Cluster . . . 273
36.3.3 Promoting a Replica to a Primary Cluster . . . 274
36.4 Standalone Replica Clusters . . . 275
36.4.1 Main Differences with Distributed Topology . . . 276
36.4.2 Example of Standalone Replica Cluster using `pg_basebackup` . . . 276
36.4.3 Example of Standalone Replica Cluster from an object store . . . 277
36.4.4 Example using a Volume Snapshot . . . 278
36.5 Delayed replicas . . . 278

37 Kubernetes Upgrade and Maintenance **281**
37.1 Importance of Regular Updates . . . 281
37.2 Maintenance Operations in a Cluster . . . 281
37.3 Temporary PostgreSQL Cluster Degradation . . . 282
37.4 Pod Disruption Budgets . . . 282
37.5 PostgreSQL Clusters used for Development or Testing . . . 282
37.6 Node Maintenance Window . . . 283

37.6.1 Single instance clusters with `reusePVC` set to `false` . . . 283

38 PostgreSQL Upgrades **285**
38.1 Minor Version Upgrades . . . 285
38.1.1 Upgrading a Minor Version in CloudNativePG . . . 285
38.2 Major Version Upgrades . . . 285
38.3 Offline In-Place Major Upgrades . . . 286
38.3.1 Upgrade Process . . . 287
38.3.2 Post-Upgrade Actions . . . 287
38.3.3 Example: Performing a Major Upgrade . . . 288
38.3.4 Upgrade Process . . . 288

39 Kubectl Plugin **291**
39.1 Install . . . 291
39.1.1 Via the installation script . . . 291
39.1.2 Using the Debian or RedHat packages . . . 291
39.1.3 Debian packages . . . 292
39.1.4 RPM packages . . . 292
39.1.5 Using the Arch Linux User Repository (AUR) Package . . . 292
39.1.6 Using Krew . . . 293
39.1.7 Using Homebrew . . . 293
39.1.8 Supported Architectures . . . 293
39.1.9 Configuring auto-completion . . . 294
39.2 Use . . . 294
39.2.1 Generation of installation manifests . . . 294
39.2.2 Status . . . 295
39.2.3 Promote . . . 298
39.2.4 Certificates . . . 298
39.2.5 Restart . . . 298
39.2.6 Reload . . . 299
39.2.7 Maintenance . . . 299
39.2.8 Report . . . 299
39.2.9 report Operator . . . 299
39.2.10 report Cluster . . . 302
39.2.11 Logs . . . 303
39.2.12 Cluster logs . . . 303
39.2.13 Pretty . . . 304
39.2.14 Destroy . . . 306
39.2.15 Cluster Hibernation . . . 306
39.2.16 Benchmarking the database with pgbench . . . 307
39.2.17 Benchmarking the storage with fio . . . 307
39.2.18 Requesting a new physical backup . . . 307
39.2.19 Launching psql . . . 308
39.2.20 Snapshotting a Postgres cluster . . . 308
39.2.21 Using pgAdmin4 for evaluation/demonstration purposes only . . . 308
39.2.22 Logical Replication Publications . . . 310
39.2.23 Creating a new publication . . . 310
39.2.24 Dropping a publication . . . 311
39.2.25 Logical Replication Subscriptions . . . 311
39.2.26 Creating a new subscription . . . 312
39.2.27 Dropping a subscription . . . 313
39.2.28 Synchronizing sequences . . . 313
39.3 Integration with K9s . . . 314
39.4 Permissions required by the plugin . . . 314

39.4.1 Role examples . . . 314

40 Automated failover **317**
40.1 RTO and RPO impact . . . 318
40.2 Delayed failover . . . 318
40.3 Failover Quorum (Quorum-based Failover) . . . 319
40.3.1 How it works . . . 319
40.3.2 Quorum Failover Example Scenarios . . . 320
40.3.3 Scenario 1: Three-node cluster, failing pod(s) . . . 320
40.3.4 Scenario 2: Three-node cluster, network partition . . . 321
40.3.5 Scenario 3: Five-node cluster, network partition . . . 321
40.3.6 Scenario 4: Three-node cluster with remote synchronous replicas . . . 321
40.3.7 Scenario 5: Three-node cluster, preferred data durability, network partition . . . 323

41 Troubleshooting **325**
41.1 Before you start . . . 325
41.1.1 Kubernetes environment . . . 325
41.1.2 Useful utilities . . . 326
41.2 First steps . . . 326
41.3 Are there backups? . . . 326
41.4 Emergency backup . . . 326
41.5 Logs . . . 327
41.6 Operator information . . . 328
41.6.1 Gather more information about the operator . . . 328
41.7 Cluster information . . . 329
41.8 Pod information . . . 330
41.8.1 Gather and filter extra information about PostgreSQL pods . . . 331
41.9 Backup information . . . 332
41.10 Storage information . . . 332
41.11 Node information . . . 332
41.12 Conditions . . . 333
41.12.1 How to wait for a particular condition . . . 333
41.13 Networking . . . 334
41.14 PostgreSQL core dumps . . . 334
41.15 Some known issues . . . 335
41.15.1 Storage is full . . . 335
41.15.2 Pods are stuck in `Pending` state . . . 335
41.15.3 Replicas out of sync when no backup is configured . . . 336
41.15.4 Cluster stuck in `Creating new replica` . . . 336
41.15.5 Networking is impaired by installed Network Policies . . . 336
41.15.6 Error while bootstrapping the data directory . . . 337
41.15.7 Bootstrap job hangs in running status . . . 337
41.15.8 Replicas take over two minutes to reconnect after a failover . . . 337

42 Fencing **339**
42.1 How to fence instances . . . 339
42.2 How to lift fencing . . . 340
42.3 How fencing works . . . 340

43 Declarative hibernation **343**
43.1 Hibernation . . . 343
43.2 Rehydration . . . 344

44 PostGIS **345**
44.1 Basic concepts about a PostGIS cluster . . . 345

44.2 Create a new PostgreSQL cluster with PostGIS . . . 346

45 End-to-End Tests **349**

46 Container Image Requirements **353**
46.1 Image Tag Requirements . . . 354

47 Image Volume Extensions **355**
47.1 Benefits . . . 355
47.2 Requirements . . . 356
47.3 How it works . . . 356
47.4 How to add a new extension . . . 356
47.4.1 Adding a new extension to a `Cluster` resource . . . 357
47.4.2 Adding a new extension to a `Database` resource . . . 358
47.5 Advanced Topics . . . 358
47.5.1 Setting Custom Paths . . . 359
47.5.2 Multi-extension Images . . . 359
47.5.3 Including System Libraries . . . 360
47.6 Image Specifications . . . 360
47.7 Caveats . . . 361

48 CNPG-I **363**
48.1 Why CNPG-I? . . . 363
48.2 Registering a plugin . . . 363
48.2.1 Sidecar Container . . . 364
48.2.2 Standalone Deployment (recommended) . . . 364
48.2.3 Configuring TLS Certificates . . . 365
48.3 Using a plugin . . . 366
48.4 Community plugins . . . 366

49 Operator capability levels **367**
49.1 Level 1: Basic install . . . 368
49.1.1 Operator deployment via declarative configuration . . . 368
49.1.2 PostgreSQL cluster deployment via declarative configuration . . . 368
49.1.3 Override of operand images through the CRD . . . 368
49.1.4 Labels and annotations . . . 368
49.1.5 Self-contained instance manager . . . 368
49.1.6 Storage configuration . . . 369
49.1.7 Replica configuration . . . 369
49.1.8 Service Configuration . . . 369
49.1.9 Database configuration . . . 369
49.1.10 Configuration of Postgres roles, users, and groups . . . 370
49.1.11 Pod security standards . . . 370
49.1.12 Affinity . . . 370
49.1.13 Topology spread constraints . . . 370
49.1.14 Command-line interface . . . 370
49.1.15 Current status of the cluster . . . 370
49.1.16 Operator's certification authority . . . 370
49.1.17 Cluster's certification authority . . . 370
49.1.18 TLS connections . . . 371
49.1.19 Certificate authentication for streaming replication . . . 371
49.1.20 Continuous configuration management . . . 371
49.1.21 Import of existing PostgreSQL databases . . . 371
49.1.22 PostGIS clusters . . . 371
49.1.23 Basic LDAP authentication for PostgreSQL . . . 371

49.1.24 Multiple installation methods . . . 371
49.1.25 Convention over configuration . . . 371
49.2 Level 2: Seamless upgrades . . . 371
49.2.1 Operator Upgrade . . . 372
49.2.2 Upgrade of the managed workload . . . 372
49.2.3 Offline In-Place Major Upgrades of PostgreSQL . . . 372
49.2.4 Display cluster availability status during upgrade . . . 372
49.3 Level 3: Full lifecycle . . . 372
49.3.1 PostgreSQL WAL archive . . . 373
49.3.2 PostgreSQL Backups . . . 373
49.3.3 Backups from a standby . . . 373
49.3.4 Full restore from a backup . . . 373
49.3.5 Point-in-time recovery (PITR) from a backup . . . 374
49.3.6 Zero-Data-Loss Clusters Through Synchronous Replication . . . 374
49.3.7 Replica clusters . . . 374
49.3.8 Distributed Database Topologies . . . 374
49.3.9 Tablespace support . . . 374
49.3.10 Customizable Startup, Liveness, and Readiness Probes . . . 375
49.3.11 Rolling deployments . . . 375
49.3.12 Scale up and down of replicas . . . 375
49.3.13 Maintenance window and PodDisruptionBudget for Kubernetes nodes . . . 375
49.3.14 Fencing . . . 375
49.3.15 Hibernation . . . 375
49.3.16 Reuse of persistent volumes storage in pods . . . 375
49.3.17 CPU and memory requests and limits . . . 376
49.3.18 Connection pooling with PgBouncer . . . 376
49.3.19 Logical Replication . . . 376
49.4 Level 4: Deep insights . . . 376
49.4.1 Prometheus exporter with configurable queries . . . 376
49.4.2 Grafana dashboard . . . 376
49.4.3 Standard output logging of PostgreSQL error messages in JSON format . . . 376
49.4.4 Real-time query monitoring . . . 377
49.4.5 Audit . . . 377
49.4.6 Kubernetes events . . . 377
49.5 Level 5: Auto pilot . . . 377
49.5.1 Automated failover for self-healing . . . 377
49.5.2 Automated recreation of a standby . . . 377

50 Custom Pod Controller **379**
50.1 PVC resizing . . . 379
50.2 Primary Instances versus Replicas . . . 379
50.3 Coherence of PVCs . . . 380
50.4 Local storage, remote storage, and database size . . . 380

51 Examples **383**
51.1 Basics . . . 383
51.2 Backups . . . 390
51.3 Replica clusters . . . 393
51.4 PostGIS . . . 397
51.5 Managed roles . . . 398
51.6 Managed services . . . 399
51.7 Declarative tablespaces . . . 399
51.8 Pooler configuration . . . 401
51.9 Logical replication via declarative Publication and Subscription objects . . . 402

51.10 Declarative management of Postgres databases . . . 405

52 Networking **407**
52.1 Cross-namespace network policy for the operator . . . 407
52.2 Cross-cluster networking . . . 408

53 Benchmarking **409**
53.1 pgbench . . . 409
53.2 fio . . . 411

54 Frequently Asked Questions (FAQ) **413**
54.1 Running PostgreSQL in Kubernetes . . . 413
54.2 High availability . . . 416

55 API Reference **417**
55.1 Resource Types . . . 417
55.2 Backup . . . 417
55.3 Cluster . . . 417
55.4 ClusterImageCatalog . . . 418
55.5 Database . . . 418
55.6 FailoverQuorum . . . 418
55.7 ImageCatalog . . . 419
55.8 Pooler . . . 419
55.9 Publication . . . 419
55.10 ScheduledBackup . . . 420
55.11 Subscription . . . 420
55.12 AffinityConfiguration . . . 420
55.13 AvailableArchitecture . . . 421
55.14 BackupConfiguration . . . 421
55.15 BackupMethod . . . 422
55.16 BackupPhase . . . 422
55.17 BackupPluginConfiguration . . . 422
55.18 BackupSnapshotElementStatus . . . 423
55.19 BackupSnapshotStatus . . . 423
55.20 BackupSource . . . 423
55.21 BackupSpec . . . 423
55.22 BackupStatus . . . 424
55.23 BackupTarget . . . 425
55.24 BootstrapConfiguration . . . 425
55.25 BootstrapInitDB . . . 425
55.26 BootstrapPgBaseBackup . . . 427
55.27 BootstrapRecovery . . . 427
55.28 CatalogImage . . . 428
55.29 CertificatesConfiguration . . . 428
55.30 CertificatesStatus . . . 429
55.31 ClusterMonitoringTLSConfiguration . . . 429
55.32 ClusterSpec . . . 430
55.33 ClusterStatus . . . 432
55.34 ConfigMapResourceVersion . . . 434
55.35 DataDurabilityLevel . . . 435
55.36 DataSource . . . 435
55.37 DatabaseObjectSpec . . . 435
55.38 DatabaseObjectStatus . . . 436
55.39 DatabaseReclaimPolicy . . . 436
55.40 DatabaseRoleRef . . . 436

55.41 DatabaseSpec . . . 436
55.42 DatabaseStatus . . . 438
55.43 EmbeddedObjectMetadata . . . 438
55.44 EnsureOption . . . 438
55.45 EphemeralVolumesSizeLimitConfiguration . . . 439
55.46 ExtensionConfiguration . . . 439
55.47 ExtensionSpec . . . 439
55.48 ExternalCluster . . . 440
55.49 FDWSpec . . . 440
55.50 FailoverQuorumStatus . . . 441
55.51 ImageCatalogRef . . . 441
55.52 ImageCatalogSpec . . . 441
55.53 ImageInfo . . . 442
55.54 Import . . . 442
55.55 ImportSource . . . 442
55.56 InstanceID . . . 443
55.57 InstanceReportedState . . . 443
55.58 IsolationCheckConfiguration . . . 443
55.59 LDAPBindAsAuth . . . 443
55.60 LDAPBindSearchAuth . . . 444
55.61 LDAPConfig . . . 444
55.62 LDAPScheme . . . 444
55.63 LivenessProbe . . . 445
55.64 ManagedConfiguration . . . 445
55.65 ManagedRoles . . . 445
55.66 ManagedService . . . 445
55.67 ManagedServices . . . 446
55.68 Metadata . . . 446
55.69 MonitoringConfiguration . . . 447
55.70 NodeMaintenanceWindow . . . 447
55.71 OnlineConfiguration . . . 448
55.72 OptionSpec . . . 448
55.73 OptionSpecValue . . . 448
55.74 PasswordState . . . 449
55.75 PgBouncerIntegrationStatus . . . 449
55.76 PgBouncerPoolMode . . . 449
55.77 PgBouncerSecrets . . . 449
55.78 PgBouncerSpec . . . 450
55.79 PluginConfiguration . . . 450
55.80 PluginStatus . . . 450
55.81 PodTemplateSpec . . . 451
55.82 PodTopologyLabels . . . 451
55.83 PoolerIntegrations . . . 451
55.84 PoolerMonitoringConfiguration . . . 452
55.85 PoolerSecrets . . . 452
55.86 PoolerSpec . . . 452
55.87 PoolerStatus . . . 453
55.88 PoolerType . . . 453
55.89 PostgresConfiguration . . . 453
55.90 PrimaryUpdateMethod . . . 454
55.91 PrimaryUpdateStrategy . . . 454
55.92 Probe . . . 454
55.93 ProbeStrategyType . . . 455
55.94 ProbeWithStrategy . . . 455

55.95 ProbesConfiguration . . . 456
55.96 PublicationReclaimPolicy . . . 456
55.97 PublicationSpec . . . 456
55.98 PublicationStatus . . . 456
55.99 PublicationTarget . . . 457
55.100 PublicationTargetObject . . . 457
55.101 PublicationTargetTable . . . 457
55.102 RecoveryTarget . . . 458
55.103 ReplicaClusterConfiguration . . . 458
55.104 ReplicationSlotsConfiguration . . . 459
55.105 ReplicationSlotsHAConfiguration . . . 459
55.106 RoleConfiguration . . . 460
55.107 SQLRefs . . . 461
55.108 ScheduledBackupSpec . . . 461
55.109 ScheduledBackupStatus . . . 462
55.110 SchemaSpec . . . 462
55.111 SecretVersion . . . 463
55.112 SecretsResourceVersion . . . 463
55.113 ServiceAccountTemplate . . . 463
55.114 ServiceSelectorType . . . 464
55.115 ServiceTemplateSpec . . . 464
55.116 ServiceUpdateStrategy . . . 464
55.117 SnapshotOwnerReference . . . 464
55.118 SnapshotType . . . 465
55.119 StorageConfiguration . . . 465
55.120 SubscriptionReclaimPolicy . . . 465
55.121 SubscriptionSpec . . . 465
55.122 SubscriptionStatus . . . 466
55.123 SwitchReplicaClusterStatus . . . 466
55.124 SyncReplicaElectionConstraints . . . 466
55.125 SynchronizeReplicasConfiguration . . . 467
55.126 SynchronousReplicaConfiguration . . . 467
55.127 SynchronousReplicaConfigurationMethod . . . 468
55.128 TablespaceConfiguration . . . 468
55.129 TablespaceState . . . 468
55.130 TablespaceStatus . . . 469
55.131 Topology . . . 469
55.132 UsageSpec . . . 469
55.133 VolumeSnapshotConfiguration . . . 470

56 Supported releases **471**
56.1 Support Policy . . . 471
56.2 Naming Scheme . . . 472
56.3 Support status of CloudNativePG releases . . . 472
56.3.1 Supported PostgreSQL versions . . . 473
56.4 Upcoming releases . . . 473
56.5 Old releases . . . 474
56.6 What we mean by support . . . 474

57 Preview Versions **477**
57.1 Purpose of Release Candidates . . . 477
57.2 Community Involvement . . . 477
57.3 Usage Advisory . . . 477
57.4 Current Preview Version . . . 478

58 Release Note **479**
58.1 Release notes for CloudNativePG 1.27 . . . 480
58.1.1 Version 1.27.0 . . . 480
58.2 Release notes for CloudNativePG 1.26 . . . 481
58.2.1 Version 1.26.1 . . . 481
58.2.2 Version 1.26.0 . . . 482
58.3 Release notes for CloudNativePG 1.25 . . . 485
58.3.1 Version 1.25.3 . . . 485
58.3.2 Version 1.25.2 . . . 486
58.3.3 Version 1.25.1 . . . 488
58.3.4 Version 1.25.0 . . . 489
58.4 Release notes for 1.14.0 and earlier . . . 490
58.4.1 Version 1.14.0 . . . 491
58.4.2 Version 1.13.0 . . . 491
58.4.3 Version 1.12.0 . . . 492
58.4.4 Version 1.11.0 . . . 493
58.4.5 Version 1.10.0 . . . 495
58.4.6 Version 1.9.2 . . . 496
58.4.7 Version 1.9.1 . . . 496
58.4.8 Version 1.9.0 . . . 496
58.4.9 Version 1.8.0 . . . 497
58.4.10 Version 1.7.1 . . . 497
58.4.11 Version 1.7.0 . . . 498
58.4.12 Version 1.6.0 . . . 499
58.4.13 Version 1.5.1 . . . 499
58.4.14 Version 1.5.0 . . . 499
58.4.15 Version 1.4.0 . . . 500
58.4.16 Version 1.3.0 . . . 501
58.4.17 Version 1.2.1 . . . 501
58.4.18 Version 1.2.0 . . . 501
58.4.19 Version 1.1.0 . . . 502
58.4.20 Version 1.0.0 . . . 502

CHAPTER 1

CloudNativePG

CloudNativePG (CNPG) is an open-source operator

designed to manage PostgreSQL workloads on any supported Kubernetes cluster. It fosters cloud-neutrality through seamless deployment in private, public, hybrid, and multi-cloud environments via its *From Replica Clusters to Distributed Topology* feature.

Built around DevOps principles, CloudNativePG embraces declarative configuration and immutable infrastructure, ensuring reliability and automation in database management.

At its core, CloudNativePG introduces a custom Kubernetes resource called `Cluster` , representing a PostgreSQL cluster with:

- A single primary instance for write operations.
- Optional replicas for High Availability and read scaling.

These instances reside within a Kubernetes namespace, allowing applications to connect seamlessly using operator-managed services. Failovers and switchovers occur transparently, eliminating the need for manual intervention.

For applications inside the Kubernetes cluster, CNPG provides a microservice database approach, enabling co-location of PostgreSQL clusters and applications in the same namespace for optimized access. For applications outside the cluster, CNPG offers flexible connectivity through service templates and `LoadBalancer` services for direct TCP exposure. Additionally, 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. The source code repository is in GitHub .

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

1.1 Supported Kubernetes distributions

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

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

1.2 Container images

The CloudNativePG community maintains container images for both the operator and PostgreSQL (the operand).

1.2.1 Operator

The CloudNativePG operator container images are available on the cloudnative-pg

in three different flavors:

- Debian 12 distroless
- Red Hat UBI 9 micro (suffix `-ubi9`)

Red Hat UBI images are primarily intended for OLM consumption.

All container images are signed and include SBOM and provenance attestations, provided separately for each architecture.

1.2.2 Operands

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

The minimal

and standard

container images are signed and include SBOM and provenance attestations, provided separately for each architecture.

Weekly jobs ensure that critical vulnerabilities (CVEs) in the entire stack are promptly addressed.

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

1.3 Main features

- Direct integration with the Kubernetes API server for High Availability, eliminating the need for external tools.
- Self-healing capabilities, including:
  - Automated failover by promoting the most aligned replica.
  - Automatic recreation of failed replicas.
- Planned switchover of the primary instance by promoting a selected replica.
- Declarative management of key PostgreSQL configurations, including:
  - PostgreSQL settings.
  - Roles, users, and groups.
  - Databases, extensions, schemas, and foreign data wrappers (FDW).
  - Tablespaces (including temporary tablespaces).

- Flexible instance definition, supporting any number of instances (minimum 1 primary server).
- Scale-up/down capabilities to dynamically adjust cluster size.
- Read-Write and Read-Only Services, ensuring applications connect correctly:
  - *Read-Write Service*: Routes connections to the primary server.
  - *Read-Only Service*: Distributes connections among replicas for read workloads.
- Support for quorum-based and priority-based PostgreSQL Synchronous Replication.
- Replica clusters enabling PostgreSQL distributed topologies across multiple Kubernetes clusters (private, public, hybrid, and multi-cloud).
- Delayed Replica clusters for point-in-time access to historical data.
- Persistent volume management, including:
  - Support for Local Persistent Volumes with PVC templates.
  - Reuse of Persistent Volumes storage in Pods.
  - Separate volumes for WAL files and tablespaces.
- Backup and Recovery via CNPG-I Plugins:
  - Pluggable architecture for continuous physical backup and recovery.
  - Hot and cold base backups.
  - WAL archiving.
  - Full and Point-In-Time Recovery (PITR).
  - Scheduled and on-demand backups.
  - Backup from standbys to reduce primary load.
- Community-Supported Barman Cloud Plugin:
  - WAL archiving to object stores with support for full/PITR recovery.
  - Retention policies based on configurable recovery windows.
  - Supported as a CNPG-I plugin (recommended approach).
- Native Backup Methods:
  - Continuous backup and full/PITR recovery via volume snapshots (if supported by the storage class).
  - Native integration with Barman Cloud for object store backups via `.spec.backup.barmanObjectStore` (*deprecated since v1.26*).
- Offline in-place major upgrades of PostgreSQL
- Offline and online import of PostgreSQL databases, including major upgrades:
  - *Offline Import*: Direct restore from existing databases.
  - *Online Import*: PostgreSQL native logical replication via the `Subscription` resource.
- High Availability physical replication slots, including synchronization of user-defined replication slots and logical decoding failover.
- Parallel WAL archiving and restore, ensuring high-performance data synchronization in high-write environments.
- TLS support, including:

  - Secure connections and client certificate authentication.
  - Custom TLS certificates (integrated with `cert-manager`).
- Startup and readiness probes, including replica probes based on desired lag from the primary.
- Declarative rolling updates for:
  - PostgreSQL minor versions.
  - Operator upgrades (in-place or rolling updates).
- Standard output logging of PostgreSQL error messages in JSON format for easier integration with log aggregation tools.
- Prometheus-compatible metrics exporter (`metrics` port 9187) for custom monitoring.
- `cnpg` plugin for `kubectl` to simplify cluster operations.
- Cluster hibernation for resource efficiency in inactive states.
- Fencing of PostgreSQL clusters (full cluster or subset) to isolate instances when needed.
- Connection pooling with PgBouncer for improved database efficiency.
- OLM (Operator Lifecycle Manager) installation support for streamlined deployments.
- Multi-arch container images, including Software Bill of Materials (SBOM) and provenance attestations for security compliance.

Note

CloudNativePG does not use *StatefulSet* s for managing data persistence. Instead, it directly manages Persistent Volume Claims (PVCs). See *Custom Pod Controller* for more details.

1.4 About this guide

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

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

The CloudNativePG documentation is licensed under a Creative Commons Attribution 4.0 International License.

CloudNativePG is a Cloud Native Computing Foundation Sandbox project .



CHAPTER 2

Before You Start

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

2.1 Kubernetes terminology

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

Postgres Node
: A *Postgres node* is a Kubernetes worker node dedicated to running PostgreSQL workloads. This is achieved by applying the `node-role.kubernetes.io` label and taint, as *proposed by CloudNativePG* . It is also referred to as a `postgres` node.

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

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

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

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

Persistent Volume
: A *persistent volume* (PV) is a resource in a Kubernetes cluster that represents storage that has been either manually provisioned by an administrator or dynamically provisioned by a *storage class* controller. A PV is associated with a pod using a *persistent volume claim* and its lifecycle is independent of any pod that uses it. Normally, a

PV is a network volume, especially in the public cloud. A **local persistent volume* (LPV)* is a persistent volume that exists only on the particular node where the pod that uses it is running.

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

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

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

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

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

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

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

2.2 PostgreSQL terminology

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

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

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

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

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

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

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

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

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

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

RTO : Acronym for "recovery time objective", the amount of time a system can be unavailable without adversely impacting the application.

RPO : Acronym for "recovery point objective", a calculation of the level of acceptable data loss following a disaster recovery scenario.

2.3 Cloud terminology

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

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

2.4 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.

CHAPTER 3

Use cases

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

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

3.1 Case 1: Applications inside Kubernetes

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

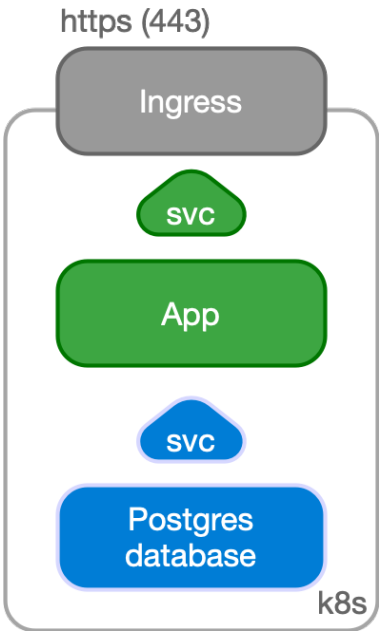


Fig. 1: Application and Database inside Kubernetes

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.

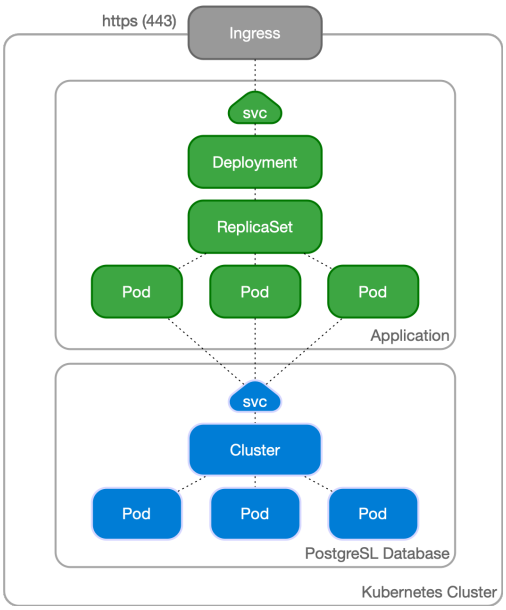


Fig. 2: Close-up view of application and database inside Kubernetes

3.2 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 (normally a `LoadBalancer` service type as explained in the *Service Management* page).

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

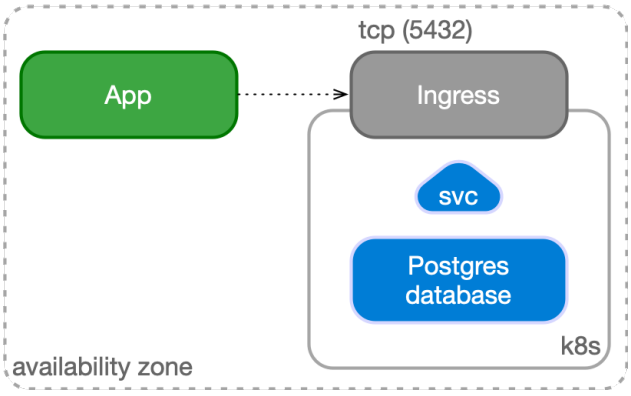


Fig. 3: Application outside Kubernetes

CHAPTER 4

Architecture

Hint

For a deeper understanding, we recommend reading our article on the CNCF blog post titled *Recommended Architectures for PostgreSQL in Kubernetes* , which provides valuable insights into best practices and design considerations for PostgreSQL deployments in Kubernetes.

This documentation page provides an overview of the key architectural considerations for implementing a robust business continuity strategy when deploying PostgreSQL in Kubernetes. These considerations include:

- ** *Multi-availability zone Kubernetes clusters*

vs. *Single availability zone Kubernetes clusters* **: Evaluating the differences between deploying in stretched clusters (across 3 or more availability zones) versus non-stretched clusters (within a single availability zone).

- *Reserving nodes for PostgreSQL workloads* : Isolating PostgreSQL workloads by dedicating specific worker nodes to `postgres` tasks, ensuring optimal performance and minimizing interference from other workloads.
- *PostgreSQL architecture* : Designing effective PostgreSQL deployments within a single Kubernetes cluster to meet high availability and performance requirements.
- *Deployments across Kubernetes clusters* : Planning and implementing PostgreSQL architectures that span multiple Kubernetes clusters to provide comprehensive disaster recovery capabilities.

4.1 Synchronizing the state

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

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

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

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

Important

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

where this topic was covered in detail.

4.2 Kubernetes architecture

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

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

Note

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

4.2.1 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.

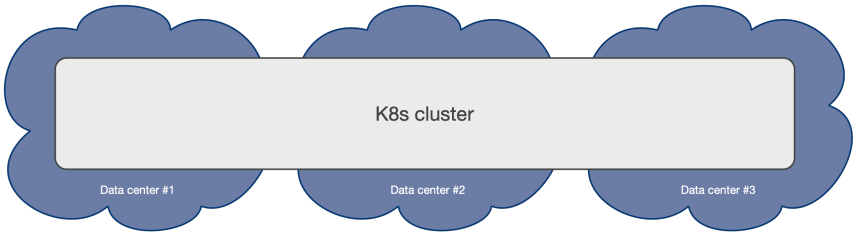


Fig. 1: Kubernetes cluster spanning over 3 independent data centers

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.

Additionally, you can leverage *Deployments across Kubernetes clusters*

to deploy distributed PostgreSQL topologies hosting "passive" *PostgreSQL replica clusters* in different regions and managing them via declarative configuration. This setup is ideal for disaster recovery (DR), read-only operations, or cross-region availability.

Important

Each operator deployment can only manage operations within its local Kubernetes cluster. For operations across Kubernetes clusters, such as controlled switchover or unexpected failover, coordination must be handled manually (through GitOps, for example) or by using a higher-level cluster management tool.

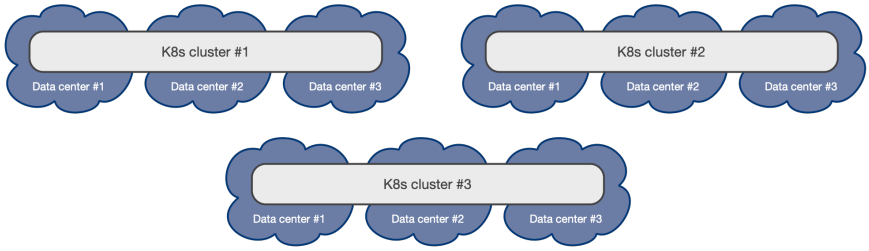


Fig. 2: Example of a multiple Kubernetes cluster architecture distributed over 3 regions each with 3 independent data centers

4.2.2 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 clusters are the only viable option when only **two data centers** are available within reach of a low-latency connection (typically in the same metropolitan area). Having only two zones prevents the creation of a multi-availability zone Kubernetes cluster, which requires a minimum of three zones. As a result, users must create two separate Kubernetes clusters in an active/passive configuration, with the second cluster primarily used for Disaster Recovery (see the *replica cluster feature*).

Hint

If you are at an 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*

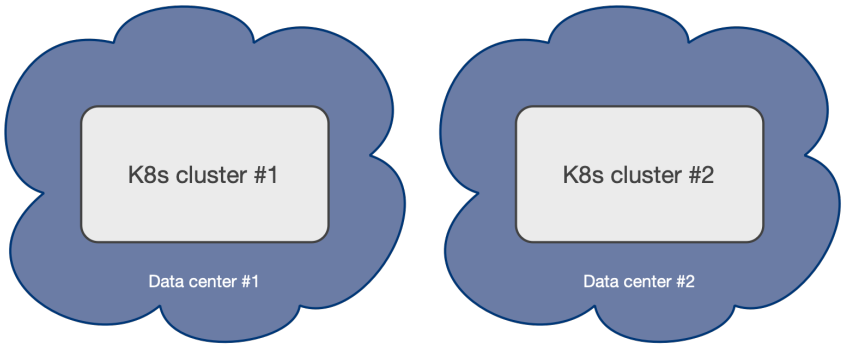


Fig. 3: Example of a Kubernetes architecture with only 2 data centers

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 *Deployments across Kubernetes clusters* to define a distributed topology hosting "passive" *PostgreSQL replica clusters* . As with other Kubernetes workloads in this scenario, promotion of a Kubernetes cluster as primary must be done manually.

Through the *replica cluster feature* , you can define a distributed PostgreSQL topology and coordinate a controlled switchover between data centers by first demoting the primary cluster and then promoting the replica cluster, without the need to re-clone the former primary. While failover is now fully declarative, automated failover across Kubernetes clusters is not within CloudNativePG's scope, as the operator can only function within a single Kubernetes cluster.

Important

CloudNativePG provides all the necessary primitives and probes to coordinate PostgreSQL active/passive topologies across different Kubernetes clusters through a higher-level operator or management tool.

4.2.3 Reserving nodes for PostgreSQL workloads

Whether you're operating in a multi-availability zone environment or, more critically, within a single availability zone, we strongly recommend isolating PostgreSQL workloads by dedicating specific worker nodes exclusively to `postgres` in production. A Kubernetes worker node dedicated to running PostgreSQL workloads is referred to as a **Postgres node** or `postgres` node. This approach ensures optimal performance and resource allocation for your database operations.

Hint

As a general rule of thumb, deploy Postgres nodes in multiples of three—ideally with one node per availability zone. Three nodes is an optimal number because it ensures that a PostgreSQL cluster with three instances (one primary and two standby replicas) is distributed across different nodes, enhancing fault tolerance and availability.

In Kubernetes, this can be achieved using node labels and taints in a declarative manner, aligning with Infrastructure as Code (IaC) practices: labels ensure that a node is capable of running `postgres` workloads, while taints help prevent any non-`postgres` workloads from being scheduled on that node.

Important

This methodology is the most straightforward way to ensure that PostgreSQL workloads are isolated from other workloads in terms of both computing resources and, when using locally attached disks, storage. While different PostgreSQL clusters may share the same node, you can take this a step further by using labels and taints to ensure that a node is dedicated to a single instance of a specific *Cluster* .

4.2.4 Proposed node label

CloudNativePG recommends using the `node-role.kubernetes.io/postgres` label. Since this is a reserved label (`*.kubernetes.io`), it can only be applied after a worker node is created.

To assign the `postgres` label to a node, use the following command:

```
kubectl label node <NODE-NAME> node-role.kubernetes.io/postgres=
```

To ensure that a `Cluster` resource is scheduled on a `postgres` node, you must correctly configure the `.spec.affinity.nodeSelector` stanza in your manifests. Here's an example:

```
spec:
  # <snip>
  affinity:
    # <snip>
    nodeSelector:
      node-role.kubernetes.io/postgres: ""
```

4.2.5 Proposed node taint

CloudNativePG recommends using the `node-role.kubernetes.io/postgres` taint.

To assign the `postgres` taint to a node, use the following command:

```
kubectl taint node <NODE-NAME> node-role.kubernetes.io/postgres=:NoSchedule
```

To ensure that a `Cluster` resource is scheduled on a node with a `postgres` taint, you must correctly configure the `.spec.affinity.tolerations` stanza in your manifests. Here's an example:

```
spec:
  # <snip>
  affinity:
    # <snip>
    tolerations:
    - key: node-role.kubernetes.io/postgres
      operator: Exists
      effect: NoSchedule
```

4.3 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 (optional) * `-r` : applications connect to any of the instances for read-only workloads (optional)
- 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

Important

You can configure the above services through the *managed.services* section in the *Cluster* configuration. This can be done by reducing the number of services and selecting the type (default is *ClusterIP*). For more details, please refer to the *Service Management*

below.

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

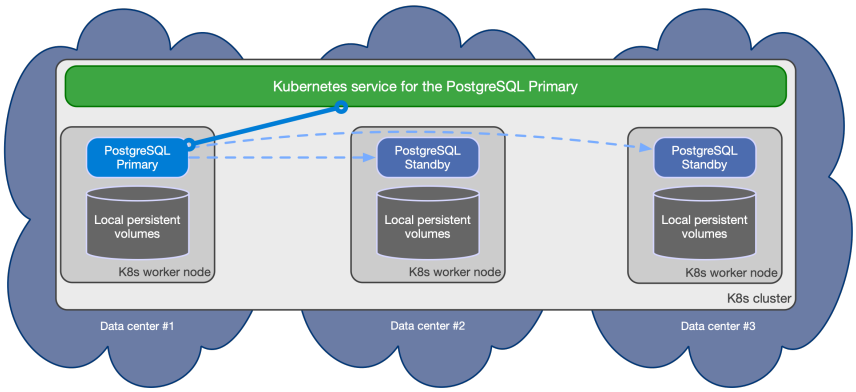


Fig. 4: Bird-eye view of the recommended shared nothing architecture for PostgreSQL in Kubernetes

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

See also

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

See also

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

See also

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

4.3.1 Read-write workloads

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

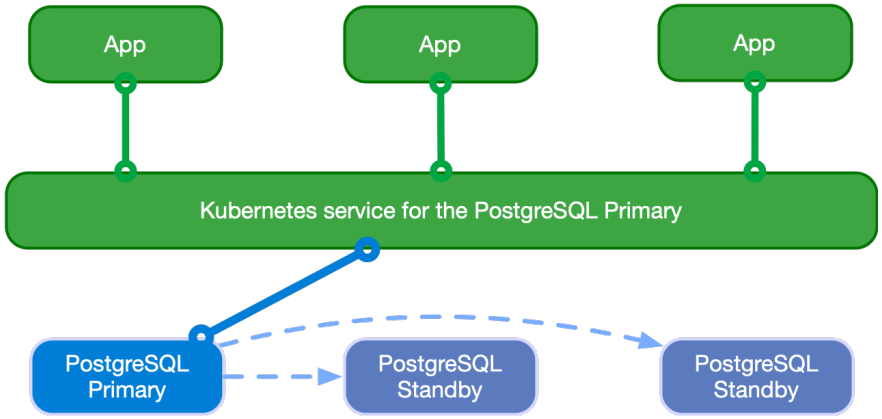


Fig. 5: Applications writing to the single primary

Applications can use the `-rw` suffix service.

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

4.3.2 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.

4.4 Deployments across Kubernetes clusters

Note

CloudNativePG supports deploying PostgreSQL across multiple Kubernetes clusters through a feature that allows you to define a distributed PostgreSQL topology using replica clusters, as described in this section.

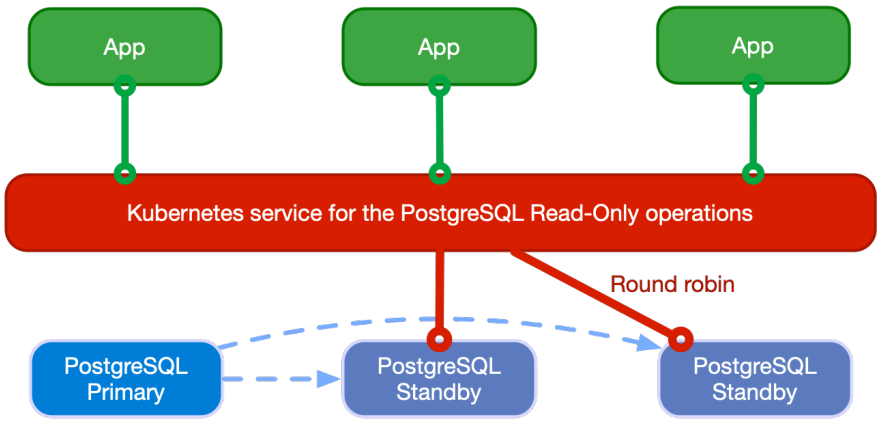


Fig. 6: Applications reading from hot standby replicas in round robin

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

However, for business continuity objectives it is fundamental to:

- reduce global **recovery point objectives** (PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode}) by storing PostgreSQL backup data in multiple locations, regions and possibly using different providers (Disaster Recovery)
- reduce global **recovery time objectives** (*RTO and RPO impact*) 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 Topology that is distributed across different Kubernetes clusters and is made up of a primary PostgreSQL cluster and one or more PostgreSQL replica clusters. This feature is called **distributed PostgreSQL topology with replica clusters**, and it enables multi-cluster deployments in private, public, hybrid, and multi-cloud contexts.

A replica cluster is a separate `Cluster` resource that is in continuous recovery, replicating from another source, either via WAL shipping from a WAL archive or via streaming replication from a primary or a standby (cascading).

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

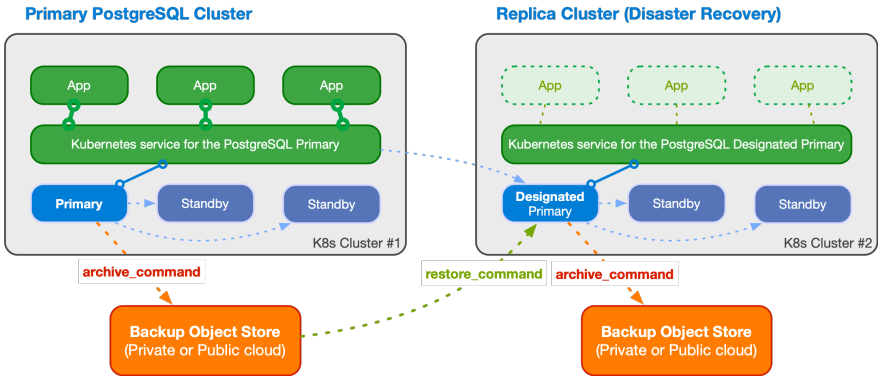


Fig. 7: An example of multi-cluster deployment with a primary and a replica cluster

A replica cluster can have the same architecture as the primary cluster. Instead of a primary instance, a replica cluster has a **designated primary** instance, which is a standby server with an arbitrary number of cascading standby servers

in streaming replication (symmetric architecture).

The designated primary can be promoted at any time, transforming the replica cluster into a primary cluster capable of accepting write connections. This is typically triggered by:

- **Human decision:** You choose to make the other PostgreSQL cluster (or the entire Kubernetes cluster) the primary. To avoid data loss and ensure that the former primary can follow without being re-cloned (especially with large data sets), you first demote the current primary, then promote the designated primary using the API provided by CloudNativePG.
- **Unexpected failure:** If the entire Kubernetes cluster fails, you might experience data loss, but you need to fail over to the other Kubernetes cluster by promoting the PostgreSQL replica cluster.

Warning

CloudNativePG cannot perform any cross-cluster automated failover, as it does not have authority beyond a single Kubernetes cluster. Such operations must be performed manually or delegated to a multi-cluster/federated cluster-aware authority.

Important

CloudNativePG allows you to control the distributed topology via declarative configuration, enabling you to automate these procedures as part of your Infrastructure as Code (IaC) process, including GitOps.

In the example above, the designated primary receives WAL updates via streaming replication (`primary_conninfo`). As a fallback, it can retrieve WAL segments from an object store using file-based WAL shipping—for instance, with the Barman Cloud plugin through `restore_command` and `barman-cloud-wal-restore`.

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

See also

Please refer to the *Replica Clusters* for more detailed information on how physical replica clusters operate and how to define a distributed topology with read-only clusters across different Kubernetes clusters. This approach can significantly enhance your global disaster recovery and high availability (HA) strategy.

CHAPTER 5

Installation and upgrades

5.1 Installation on Kubernetes

5.1.1 Directly using the operator manifest

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

You can install the latest operator manifest

for this minor release as follows:

```
kubectl apply --server-side -f \
  https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-1.27/
↪releases/cnpg-1.27.0.yaml
```

You can verify that with:

```
kubectl rollout status deployment \
  -n cnpg-system cnpg-controller-manager
```

5.1.2 Using the `cnpg` plugin for `kubectl`

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

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

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

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

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

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

5.1.3 Testing the latest development snapshot

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

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

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

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

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

```
curl -sSfL \
  https://raw.githubusercontent.com/cloudnative-pg/artifacts/release-1.27/manifests/
↪operator-manifest.yaml | \
  kubectl apply --server-side -f -
```

> **Important**
>
> Snapshots are not supported by the CloudNativePG Community, and are not intended for use in production.

5.1.4 Using the Helm Chart

The operator can be installed using the provided Helm chart .

5.1.5 Using OLM

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

directly from OperatorHub.io .

For deployments on Red Hat OpenShift, EDB offers and fully supports a certified version of CloudNativePG, available through the Red Hat OpenShift Container Platform .

5.2 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).

See also

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

5.3 Upgrades

Important

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

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

Upgrading CloudNativePG operator is a two-step process:

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

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

The second step is automatically triggered after updating the controller. By default, this initiates a rolling update of every deployed PostgreSQL cluster, upgrading one instance at a time to use the new instance manager. The rolling update concludes with a switchover, which is governed by the `primaryUpdateStrategy` option. The default value, `unsupervised` , completes the switchover automatically. If set to `supervised` , the user must manually promote the new primary instance using the `cnpg` plugin for `kubectl` .

See also

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

Important

In case *primaryUpdateStrategy* is set to the default value of *unsupervised* , an upgrade of the operator will trigger a switchover on your PostgreSQL cluster, causing a (normally negligible) downtime. If your PostgreSQL Cluster has only one instance, the instance will be automatically restarted as *supervised* value is not supported for *primaryUpdateStrategy* . In either case, your applications will have to reconnect to PostgreSQL.

The default rolling update behavior can be replaced with in-place updates of the instance manager. This approach does not require a restart of the PostgreSQL instance, thereby avoiding a switchover within the cluster. This feature, which is disabled by default, is described in detail below.

5.3.1 Spread Upgrades

By default, all PostgreSQL clusters are rolled out simultaneously, which may lead to a spike in resource usage, especially when managing multiple clusters. CloudNativePG provides two configuration options at the *operator level*

that allow you to introduce delays between cluster roll-outs or even between instances within the same cluster, helping to distribute resource usage over time:

- `CLUSTERS_ROLLOUT_DELAY` : Defines the number of seconds to wait between roll-outs of different PostgreSQL clusters (default: `0`).
- `INSTANCES_ROLLOUT_DELAY` : Defines the number of seconds to wait between roll-outs of individual instances within the same PostgreSQL cluster (default: `0`).

5.3.2 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, each instance manager in CloudNativePG supports the injection of a new executable that replaces the existing one after successfully completing an integrity verification phase and gracefully terminating all internal processes. Upon restarting with the new binary, the instance manager seamlessly 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.

5.3.3 Compatibility among versions

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

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

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

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

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

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

<!–

5.3.4 Upgrading to 1.28.0 or 1.27.x

Important

We strongly recommend that all CloudNativePG users upgrade to version 1.28.0, or at least to the latest stable version of your current minor release (e.g., 1.27.x).

- ->

5.3.5 Upgrading to 1.27.0 or 1.26.1

Important

We strongly recommend that all CloudNativePG users upgrade to version 1.27.0, or at least to the latest stable version of your current minor release (e.g., 1.26.1).

Version 1.27 introduces a change in the default behavior of the *Liveness Probe* : it now enforces the *shutdown of an isolated primary*

within the `livenessProbeTimeout` (30 seconds).

If this behavior is not suitable for your environment, you can disable the *isolation check* in the liveness probe with the following configuration:

```
spec:
  probes:
    liveness:
      isolationCheck:
        enabled: false
```

5.3.6 Upgrading to 1.26 from a previous minor version

Important

We strongly recommend that all CloudNativePG users upgrade to version 1.26.1, or at a minimum, to the latest stable version of your current minor release (for example, 1.25.x).

Warning

Due to changes in the startup probe for the manager component (#6623), upgrading the operator will trigger a restart of your PostgreSQL clusters, even if in-place updates are enabled (*ENABLE_INSTANCE_MANAGER_IN-PLACE_UPDATES=true*). Your applications will need to reconnect to PostgreSQL after the upgrade.

5.3.7 Deprecation of backup metrics and fields in the `Cluster .status`

With the transition to a backup and recovery agnostic approach based on CNPG-I plugins in CloudNativePG, which began with version 1.26.0 for Barman Cloud, we are starting the deprecation period for the following fields in the `.status` section of the `Cluster` resource:

- `firstRecoverabilityPoint`
- `firstRecoverabilityPointByMethod`
- `lastSuccessfulBackup`
- `lastSuccessfulBackupByMethod`
- `lastFailedBackup`

The following Prometheus metrics are also deprecated:

- `cnpg_collector_first_recoverability_point`
- `cnpg_collector_last_failed_backup_timestamp`
- `cnpg_collector_last_available_backup_timestamp`

Warning

If you have migrated to a plugin-based backup and recovery solution such as Barman Cloud, these fields and metrics are no longer synchronized and will not be updated. Users still relying on the in-core support for Barman Cloud and volume snapshots can continue to use these fields for the time being.

Under the new plugin-based approach, multiple backup methods can operate simultaneously, each with its own timeline for backup and recovery. For example, some plugins may provide snapshots without WAL archiving, while others support continuous archiving.

Because of this flexibility, maintaining centralized status fields in the `Cluster` resource could be misleading or confusing, as they would not accurately represent the state across all configured backup methods. For this reason, these fields are being deprecated.

Instead, each plugin is responsible for exposing its own backup status information and providing metrics back to the instance manager for monitoring and operational awareness.

5.3.8 Declarative Hibernation in the `cnpg` plugin

In this release, the `cnpg` plugin for `kubectl` transitions from an imperative to a *declarative approach for cluster hibernation* . The `hibernate on` and `hibernate off` commands are now convenient shortcuts that apply declarative changes to enable or disable hibernation. The `hibernate status` command has been removed, as its purpose is now fulfilled by the standard `status` command.

5.3.9 Upgrading to 1.25 from a previous minor version

> **Warning**
>
> Every time you are upgrading to a higher minor release, make sure you go through the release notes and upgrade instructions of all the intermediate minor releases. For example, if you want to move from 1.23.x to 1.25, make sure you go through the release notes and upgrade instructions for 1.24 and 1.25.

No changes to existing 1.24 cluster configurations are required when upgrading to 1.25.

5.3.10 Upgrading to 1.24 from a previous minor version

5.3.11 From Replica Clusters to Distributed Topology

One of the key enhancements in CloudNativePG 1.24.0 is the upgrade of the replica cluster feature.

The former replica cluster feature, now referred to as the "Standalone Replica Cluster," is no longer recommended for Disaster Recovery (DR) and High Availability (HA) scenarios that span multiple Kubernetes clusters. Standalone replica clusters are best suited for read-only workloads, such as reporting, OLAP, or creating development environments with test data.

For DR and HA purposes, CloudNativePG now introduces the Distributed Topology strategy for replica clusters. This new strategy allows you to build PostgreSQL clusters across private, public, hybrid, and multi-cloud environments, spanning multiple regions and potentially different cloud providers. It also provides an API to control the switchover operation, ensuring that only one cluster acts as the primary at any given time.

This Distributed Topology strategy enhances resilience and scalability, making it a robust solution for modern, distributed applications that require high availability and disaster recovery capabilities across diverse infrastructure setups.

You can seamlessly transition from a previous replica cluster configuration to a distributed topology by modifying all the `Cluster` resources involved in the distributed PostgreSQL setup. Ensure the following steps are taken:

- Configure the `externalClusters` section to include all the clusters involved in the distributed topology. We strongly suggest using the same configuration across all `Cluster` resources for maintainability and consistency.
- Configure the `primary` and `source` fields in the `.spec.replica` stanza to reflect the distributed topology. The `primary` field should contain the name of the current primary cluster in the distributed topology, while the `source` field should contain the name of the cluster each `Cluster` resource is replicating from. It is important to note that the `enabled` field, which was previously set to `true` or `false` , should now be unset (default).

For more information, please refer to the *From Replica Clusters to Distributed Topology* .

CHAPTER 6

Quickstart

This section guides you through testing a PostgreSQL cluster on your local machine by deploying CloudNativePG on a local Kubernetes cluster using either 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/>`_ .

6.1 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*

6.1.1 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 toinstall 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` .

6.1.2 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
```

6.2 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.

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

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

  storage:
    size: 1Gi
```

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

(continues on next page)

(continued from previous page)

```
spec:
  instances: 3

  storage:
    size: 1Gi
```

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

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

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

```
kubectl get pods
```

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

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

Important

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

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

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

Important

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

6.4 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 , including the Alert Manager , and a Grafana deployment.

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

6.4.1 Installation

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

We need to add the `prometheus-community` helm chart repository, and then install the *Kube Prometheus stack* with our sample configuration.

```
#
#  Copyright © contributors to CloudNativePG, established as

#  CloudNativePG a Series of LF Projects, LLC.

#
#  Licensed under the Apache License, Version 2.0 (the "License");

#  you may not use this file except in compliance with the License.

#  You may obtain a copy of the License at

#

#      http://www.apache.org/licenses/LICENSE-2.0

#

#  Unless required by applicable law or agreed to in writing, software

#  distributed under the License is distributed on an "AS IS" BASIS,

#  WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.

#  See the License for the specific language governing permissions and

```

(continues on next page)

(continued from previous page)

```
#  limitations under the License.

#

#  SPDX-License-Identifier: Apache-2.0

#

#  -- here you can pass the whole values directly to the kube-prometheus-stack chart

enabled: true
kubeControllerManager:
  enabled: false
nodeExporter:
  enabled: false
defaultRules:
  create: true
  rules:
    alertmanager: false
    etcd: false
    configReloaders: false
    general: false
    k8s: true
    kubeApiserver: false
    kubeApiserverAvailability: false
    kubeApiserverSlos: false
    kubelet: true
    kubeProxy: false
    kubePrometheusGeneral: false
    kubePrometheusNodeRecording: false
    kubernetesApps: false
    kubernetesResources: false
    kubernetesStorage: false
    kubernetesSystem: false
    kubeScheduler: false
    kubeStateMetrics: false
    network: false
    node: true
    nodeExporterAlerting: false
    nodeExporterRecording: true
    prometheus: false
    prometheusOperator: false

# nodeSelector:
  #workload: monitor
prometheus:
  prometheusSpec:
    podMonitorSelectorNilUsesHelmValues: false
    ruleSelectorNilUsesHelmValues: false
    serviceMonitorSelectorNilUsesHelmValues: false
    probeSelectorNilUsesHelmValues: false
  #nodeSelector:
```

(continues on next page)

(continued from previous page)

```
    #workload: monitor
grafana:
  enabled: true
  # -- the grafana admin password
  adminPassword: prom-operator
  defaultDashboardsEnabled: false
  sidecar:
      dashboards:
        enabled: true
  #nodeSelector:
    #workload: monitor
alertmanager:
  enabled: true
  #alertManagerSpec:
    #nodeSelector:
      #workload: monitor
```

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, configured with the `kube-stack-config.yaml` file:

- From the Prometheus installation, you will have the Prometheus Operator watching for **any** `PodMonitor` (see **monitoring**).
- Alert Manager and Grafana are both enabled.

See also

For further information about the above helm commands, refer to 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
```

6.4.2 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/

You should find a series of metrics relating to CloudNativePG clusters. Please refer to the *monitoring section* for more information.

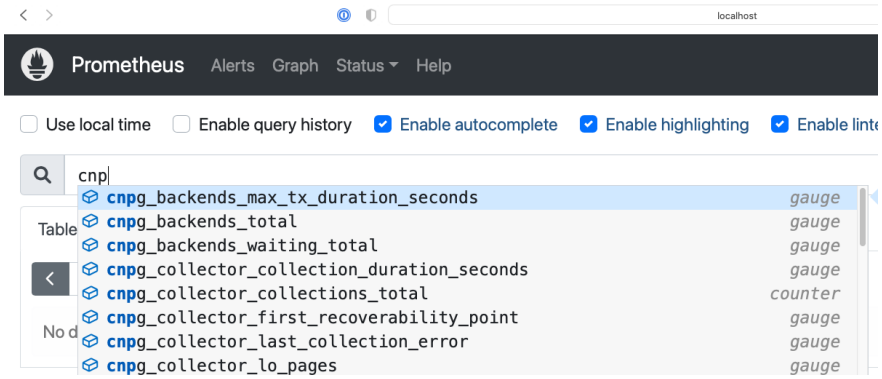


Fig. 1: local prometheus

You can also monitor the CloudNativePG operator by creating a PodMonitor to target it. See the relevant section in the *monitoring page* .

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

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

You should see the default alerts now:

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

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

6.4.3 Grafana Dashboard

In our installation so far, Grafana is deployed with no predefined dashboards.

To open Grafana, you can port-forward the grafana service:

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

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

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

CloudNativePG provides a default dashboard for Grafana in the dedicated grafana-dashboards . You can download the file grafana-dashboard.json

and manually import it via the GUI (menu: Dashboards > New > Import). You can now click on the `CloudNativePG` dashboard just created:



Fig. 2: local grafana

Warning

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

Note that in our local setup, Prometheus and Grafana are configured to automatically discover and monitor any CloudNativePG clusters deployed with the Monitoring feature enabled.

CHAPTER 7

Image Catalog

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

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

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

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

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

Example of a Namespaced ``ImageCatalog`` :

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: ImageCatalog
metadata:
  name: postgresql
  namespace: default
spec:
  images:
    - major: 15
      image: ghcr.io/cloudnative-pg/postgresql:15.6
    - major: 16
      image: ghcr.io/cloudnative-pg/postgresql:16.8
```

(continues on next page)

(continued from previous page)

```
    - major: 17
      image: ghcr.io/cloudnative-pg/postgresql:17.5
```

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

```
apiVersion: postgresql.cnpg.io/v1
kind: ClusterImageCatalog
metadata:
  name: postgresql
spec:
  images:
    - major: 15
      image: ghcr.io/cloudnative-pg/postgresql:15.6
    - major: 16
      image: ghcr.io/cloudnative-pg/postgresql:16.8
    - major: 17
      image: ghcr.io/cloudnative-pg/postgresql:17.5
```

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

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  imageCatalogRef:
    apiGroup: postgresql.cnpg.io
    kind: ImageCatalog
    name: postgresql
    major: 16
  storage:
    size: 1Gi
```

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

7.1 CloudNativePG Catalogs

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

7.1.1 PostgreSQL Container Images

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

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

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

7.1.2 PostGIS Container Images

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

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

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

CHAPTER 8

Bootstrap

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

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

The `initdb` bootstrap also provides the option to import one or more databases from an existing PostgreSQL cluster, even if it's outside Kubernetes or running a different major version of PostgreSQL. For more detailed information about this feature, please refer to the *Importing Postgres databases* section.

Important

Bootstrapping from an existing cluster enables the creation of a **replica cluster**—an independent PostgreSQL cluster that remains in continuous recovery, stays synchronized with the source cluster, and accepts read-only connections. For more details, refer to the *Replica Cluster section* .

Warning

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

Note

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

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

for details.

8.1 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 a base backup of an existing cluster and, if needed, replaying all the available WAL files or up to a given *point in time*
- `pg_basebackup` : create a PostgreSQL cluster by cloning an existing one of the same major version using `pg_basebackup` through the streaming replication protocol. This method is particularly useful for migrating databases to CloudNativePG, although meeting all requirements can be challenging. Be sure to review the warnings in the Bootstrap from a live cluster (`pg_basebackup ) <Bootstrap from a live cluster (*pg_basebackup* )>`

carefully.

Only one bootstrap method can be specified in the manifest. Attempting to define multiple bootstrap methods will result in validation errors.

In contrast to 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. Refer to the *replica cluster section* for more information.

Given the amount of possible backup methods and combinations of backup storage that the CloudNativePG operator provides for `recovery` , please refer to the dedicated Recovery from an object store for guidance on each method.

See also

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

for more information.

8.2 The `externalClusters` section

The `externalClusters` section of the cluster manifest can be used to configure access to one or more PostgreSQL clusters as *sources*. The primary use cases include:

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

Note

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

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

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

Note

A recovery object store is normally an AWS S3, Azure Blob Storage, or 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. The following table summarizes your options:

Content of externalClusters	pg_basebackup	recovery
Only streaming	✓	
Only object store		✓
Streaming and object store	✓	✓

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

See also

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

for more information.

8.2.1 Password files

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

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

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

8.3 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:

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

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

  storage:
    size: 1Gi
```

The above example of bootstrap will:

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

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

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

The supplied secret must comply with the specifications of the kubernetes.io/basic-auth . As a result, the `username` in the secret must match the one of the `owner` (for the application secret) and `postgres` for the superuser one.

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

```
apiVersion: v1
data:
  username: YXBw
  password: cGFzc3dvcmQ=
kind: Secret
metadata:
```

(continues on next page)

(continued from previous page)

```
  name: app-secret
type: kubernetes.io/basic-auth
```

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

Important

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

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

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

8.3.1 Passing Options to `initdb`

The PostgreSQL data directory is initialized using the initdb .

CloudNativePG enables you to customize the behavior of `initdb` to modify settings such as default locale configurations and data checksums.

Warning

CloudNativePG acts only as a direct proxy to *initdb* for locale-related options, due to the ongoing and significant enhancements in PostgreSQL's locale support. It is your responsibility to ensure that the correct options are provided, following the PostgreSQL documentation, and to verify that the bootstrap process completes successfully.

To include custom options in the `initdb` command, you can use the following parameters:

builtinLocale : When `builtinLocale` is set to a value, CloudNativePG passes it to the `--builtin-locale` option in `initdb` . This option controls the builtin locale, as defined in Locale Support

from the PostgreSQL documentation (default: empty). Note that this option requires `localeProvider` to be set to `builtin` . Available from PostgreSQL 17.

dataChecksums : When `dataChecksums` is set to `true` , CloudNativePG 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, CloudNativePG passes it to the `--encoding` option in `initdb` , which selects the encoding of the template database (default: `UTF8`).

icuLocale : When `icuLocale` is set to a value, CloudNativePG passes it to the `--icu-locale` option in `initdb` . This option controls the ICU locale, as defined in Locale Support

from the PostgreSQL documentation (default: empty). Note that this option requires `localeProvider` to be set to `icu` . Available from PostgreSQL 15.

icuRules : When `icuRules` is set to a value, CloudNativePG passes it to the `--icu-rules` option in `initdb` . This option controls the ICU locale, as defined in Locale Support from the PostgreSQL documentation (default: empty). Note that this option requires `localeProvider` to be set to `icu` . Available from PostgreSQL 16.

locale : When `locale` is set to a value, CloudNativePG passes it to the `--locale` option in `initdb` . This option controls the locale, as defined in Locale Support from the PostgreSQL documentation. By default, the locale parameter is empty. In this case, environment variables such as `LANG` are used to determine the locale. Be aware that these variables can vary between container images, potentially leading to inconsistent behavior.

localeCollate : When `localeCollate` is set to a value, CloudNativePG 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, CloudNativePG 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`).

localeProvider : When `localeProvider` is set to a value, CloudNativePG passes it to the `--locale-provider` option in `initdb` . This option controls the locale provider, as defined in Locale Support from the PostgreSQL documentation (default: empty, which means `libc` for PostgreSQL). Available from PostgreSQL 15.

walSegmentSize : When `walSegmentSize` is set to a value, CloudNativePG 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` :

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

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

Warning

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

8.3.2 Executing Queries After Initialization

You can specify a custom list of queries that will be executed once, immediately after the cluster is created and configured. These queries will be executed as the *superuser* (`postgres`) against three different databases, in this specific order:

1. The `postgres` database (`postInit` section)
2. The `template1` database (`postInitTemplate` section)
3. The application database (`postInitApplication` section)

For each of these sections, CloudNativePG provides two ways to specify custom queries, executed in the following order:

- As a list of SQL queries in the cluster's definition (`postInitSQL` , `postInitTemplateSQL` , and `postInitApplicationSQL` stanzas)
- As a list of Secrets and/or ConfigMaps, each containing a SQL script to be executed (`postInitSQLRefs` , `postInitTemplateSQLRefs` , and `postInitApplicationSQLRefs` stanzas). Secrets are processed before ConfigMaps.

Objects in each list will be processed sequentially.

Warning

Use the *postInit* , *postInitTemplate* , and *postInitApplication* options with extreme care, as queries are run as a superuser and can disrupt the entire cluster. An error in any of those queries will interrupt the bootstrap phase, leaving the cluster incomplete and requiring manual intervention.

Important

Ensure the existence of entries inside the ConfigMaps or Secrets specified in *postInitSQLRefs* , *postInitTemplateSQLRefs* , and *postInitApplicationSQLRefs* , otherwise the bootstrap will fail. Errors in any of those SQL files will prevent the bootstrap phase from completing successfully.

The following example runs a single SQL query as part of the `postInitSQL` stanza:

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

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

(continues on next page)

(continued from previous page)

```
  storage:
    size: 1Gi
```

The example below relies on `postInitApplicationSQLRefs` to specify a secret and a ConfigMap containing the queries to run after the initialization on the application database:

```
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

Within 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 commands like *psql* cannot.

8.4 Bootstrap from another cluster

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

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

Important

By default the *recovery* method strictly uses the *name* of the cluster in the *externalClusters* section to locate the main folder of the backup data within the object store, which is normally reserved for the name of the server. Backup plugins provide ways to specify a different one. For example, the Barman Cloud Plugin provides the serverName

(by default assigned to the value of `name` in the external cluster definition).

8.4.1 Bootstrap from a backup (`recovery`)

Given the variety of backup methods and combinations of backup storage options provided by the CloudNativePG operator for `recovery` , please refer to the dedicated Recovery from an object store for detailed guidance on each method.

8.4.2 Bootstrap from a live cluster (`pg_basebackup`)

The `pg_basebackup` bootstrap mode allows you to create a new cluster (*target*) as an exact physical copy of an existing and **binary-compatible** PostgreSQL instance (*source*) managed by CloudNativePG, using a valid *streaming replication* connection. The source instance can either be a primary or a standby PostgreSQL server. It's crucial to thoroughly review the requirements section below, as the pros and cons of PostgreSQL physical replication fully apply.

The primary use cases for this method include:

- Reporting and business intelligence clusters that need to be regenerated periodically (daily, weekly)
- Test databases containing live data that require periodic regeneration (daily, weekly, monthly) and anonymization
- Rapid spin-up of a standalone replica cluster
- Physical migrations of CloudNativePG clusters to different namespaces or Kubernetes clusters

Important

Avoid using this method, based on physical replication, to migrate an existing PostgreSQL cluster outside of Kubernetes into CloudNativePG, unless you are completely certain that all *Requirements* are met and the operation has been thoroughly tested. The CloudNativePG community does not endorse this approach for such use cases, and recommends using logical import instead. It is exceedingly rare that all requirements for physical replication are met in a way that seamlessly works with CloudNativePG.

Warning

In its current implementation, this method clones the source PostgreSQL instance, thereby creating a *snapshot*. Once the cloning process has finished, the new cluster is immediately started. Refer to *Current limitations* for more details.

Similar to the `recovery` bootstrap method, once the cloning operation is complete, the operator takes full ownership of the target cluster, starting from the first instance. This includes overriding certain configuration parameters as required by CloudNativePG, resetting the superuser password, creating the `streaming_replica` user, managing replicas, and more. The resulting cluster operates independently from the source instance.

Important

Configuring the network connection between the target and source instances lies outside the scope of CloudNativePG documentation, as it depends heavily on the specific context and environment.

The streaming replication client on the target instance, managed transparently by `pg_basebackup` , can authenticate on the source instance using one of the following methods:

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

Both authentication methods are detailed below.

8.4.3 Requirements

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

- target and source must have the same hardware architecture
- target and source must have the same major PostgreSQL version
- target and source must have the same tablespaces
- source must be configured with enough `max_wal_senders` to grant access from the target for this one-off operation by providing at least one *walsender* for the backup plus one for WAL streaming
- the network between source and target must be configured to enable the target instance to connect to the PostgreSQL port on the source instance
- source must have a role with `REPLICATION LOGIN` privileges and must accept connections from the target instance for this role in `pg_hba.conf` , preferably via TLS (see *About the replication user* below)
- target must be able to successfully connect to the source PostgreSQL instance using a role with `REPLICATION LOGIN` privileges

See also

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

and the High Availability, Load Balancing, and Replication

in the PostgreSQL documentation.

8.4.4 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.

8.4.5 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 17.5 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:17.5

  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 17.5).

8.4.6 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 17.5 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).

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-clone-tls
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:17.5

  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
```

8.4.7 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.

> **Important**
>
> While the *Cluster* is in recovery mode, no changes to the database, including the catalog, are permitted. This restriction includes any role overrides, which are deferred until the *Cluster* transitions to primary. During the recovery phase, roles remain as defined in the source cluster.

The example below configures the `app` database with the owner `app` and the password stored in the provided secret `app-secret` , following the bootstrap from a live cluster.

```
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 only **after recovery is completed**:

1. If the `app` database does not exist, it will be created.
2. If the `app` user does not exist, it will be created.
3. If the `app` user is not the owner of the `app` database, ownership will be granted to the `app` user.
4. If the `username` value matches the `owner` value in the secret, the password for the application user (the `app` user in this case) will be updated to the `password` value in the secret.

8.4.8 Current limitations

Snapshot copy

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

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

Note that this limitation applies only if the target cluster is not defined as a replica cluster.

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.

CHAPTER 9

Importing Postgres databases

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

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

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

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

- importing one or more databases from an existing PostgreSQL instance, even outside Kubernetes
- importing the database from any PostgreSQL version to one that is either the same or newer, enabling *major upgrades* of PostgreSQL (e.g. from version 13.x to version 17.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".

9.1 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:

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

The first import method is available via the `microservice` type, the second via 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 `pg_dump* <https://www.postgresql.org/docs/current/app-pgdump.html>`_

for further information.

9.2 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 -Fd`
- 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

In the figure below, a single PostgreSQL cluster containing *N* databases is imported into separate CloudNativePG clusters, with each cluster using a microservice import for one of the *N* source databases.

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.

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

(continues on next page)

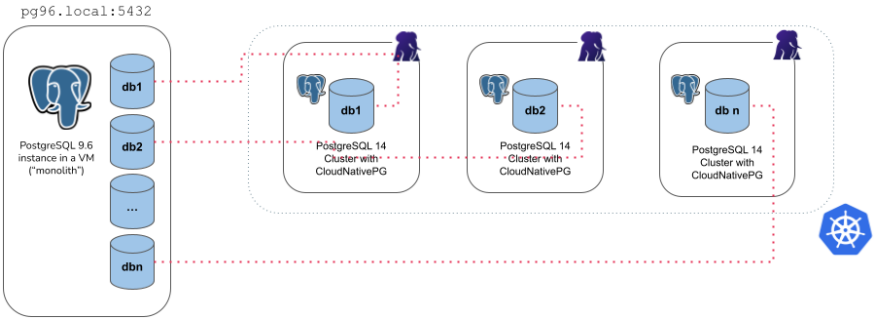


Fig. 1: Example of microservice import type

(continued from previous page)

```
  instances: 3

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

Warning

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

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

- It requires an `externalCluster` that points to an existing PostgreSQL instance containing the data to import (for more information, please refer to The `externalClusters section <The *externalClusters* section>`)
- Traffic must be allowed between the Kubernetes cluster and the `externalCluster` during the operation
- Connection to the source database must be granted with the specified user that needs to run `pg_dump` and read roles information (*superuser* is OK)
- Currently, the `pg_dump -Fd` 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`

Hint

The microservice approach adheres to CloudNativePG conventions and defaults for the destination cluster. If you do not set *initdb.database* or *initdb.owner* for the destination cluster, both parameters will default to *app* .

9.3 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 -Fd`
- create each of the selected databases and import data using `pg_restore`
- run `ANALYZE` on each imported database
- cleanup of the database dump files

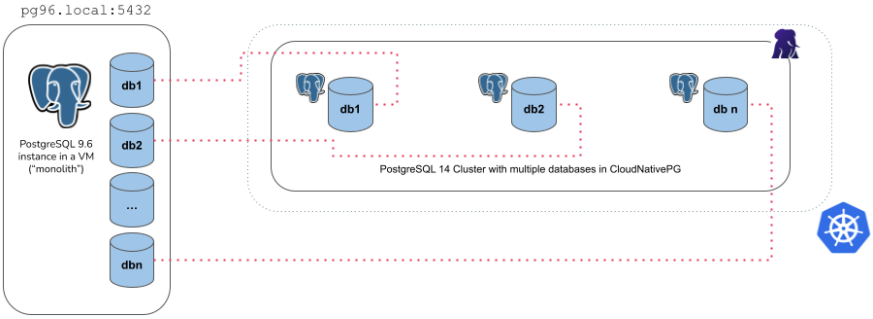


Fig. 2: Example of monolith import type

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 -Fd` result is stored temporarily inside the `dumps` folder in the `PGDATA` volume of the destination cluster's instances, 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` , `cnpg_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.
- The `postImportApplicationSQL` field is not supported

Hint

The databases and their owners are preserved exactly as they exist in the source cluster—no *app* database or user will be created during import. If your *bootstrap.initdb* stanza specifies custom *database* and *owner* values that do not match any of the databases or users being imported, the instance manager will create a new, empty application database and owner role with those specified names, while leaving the imported databases and owners unchanged.

9.4 A practical example

There is nothing to stop you from using the `monolith` approach to import a single database. It is interesting to see how the results of doing so would differ from using the `microservice` approach.

Given a source cluster, for example the following, with a database named `mydb` owned by role `me` :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 1

  postgresql:
    pg_hba:
      - host all all all trust

  storage:
    size: 1Gi

  bootstrap:
    initdb:
      database: mydb
      owner: me
```

We can import it via `microservice` :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-microservice
spec:
  instances: 1

  storage:
    size: 1Gi
```

(continues on next page)

(continued from previous page)

```
  bootstrap:
    initdb:
      import:
        type: microservice
        databases:
          - mydb
        source:
          externalCluster: cluster-example

  externalClusters:
    - name: cluster-example
      connectionParameters:
        host: cluster-example-rw
        dbname: postgres
```

as well as via monolith:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-monolith
spec:
  instances: 1

  storage:
    size: 1Gi

  bootstrap:
    initdb:
      import:
        type: monolith
        databases:
          - mydb
        roles:
          - me
        source:
          externalCluster: cluster-example

  externalClusters:
    - name: cluster-example
      connectionParameters:
        host: cluster-example-rw
        dbname: postgres
```

In both cases, the database's contents will be imported, but:

- In the microservice case, the imported database's name and owner both become `app` , or whichever configuration for the fields `database` and `owner` are set in the `bootstrap.initdb` stanza.
- In the monolith case, the database and owner are kept exactly as in the source cluster, i.e. `mydb` and `me` respectively. No `app` database nor user will be created. If there are custom settings for `database` and `owner` in the `bootstrap.initdb` stanza that don't match the source databases/owners to import, the instance manager will create a new empty application database and owner role, but will leave the imported databases/owners intact.

9.5 Import optimizations

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

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

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

Important

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

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

9.6 Customizing `pg_dump` and `pg_restore` Behavior

You can customize the behavior of `pg_dump` and `pg_restore` by specifying additional options using the `pgDumpExtraOptions` and `pgRestoreExtraOptions` parameters. For instance, you can enable parallel jobs to speed up data import/export processes, as shown in the following example:

```
# <snip>
bootstrap:
  initdb:
    import:
      type: microservice
      databases:
      - app
      source:
        externalCluster: cluster-example
      pgDumpExtraOptions:
      - --jobs=2
      pgRestoreExtraOptions:
      - --jobs=2
# <snip>
```

Warning

Use the *pgDumpExtraOptions* and *pgRestoreExtraOptions* fields with caution and at your own risk. These options are not validated or verified by the operator, and some configurations may conflict with its intended functionality or behavior. Always test thoroughly in a safe and controlled environment before applying them in production.

9.7 Online Import and Upgrades

Logical replication offers a powerful way to import any PostgreSQL database accessible over the network using the following approach:

- **Import Bootstrap with Schema-Only Option**: Initialize the schema in the target database before replication begins.
- **``Subscription`` Resource**: Set up continuous replication to synchronize data changes.

This technique can also be leveraged for performing major PostgreSQL upgrades with minimal downtime, making it ideal for seamless migrations and system upgrades.

For more details, including limitations and best practices, refer to the *Logical Replication Publications* section in the documentation.

CHAPTER 10

Security

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

Warning

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

page from the Kubernetes documentation.

See also

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

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

10.1 Code

CloudNativePG's source code undergoes systematic static analysis, including checks for security vulnerabilities, using the popular open-source linter for Go, GolangCI-Lint , directly integrated into the CI/CD pipeline. GolangCI-Lint can run multiple linters on the same source code.

The following tools are used to identify security issues:

- ** Golang Security Checker (`gosec`):** A linter that scans the abstract syntax tree of the source code against a set of rules designed to detect known vulnerabilities, threats, and weaknesses, such as hard-coded credentials, integer overflows, and SQL injections. GolangCI-Lint runs `gosec` as part of its suite.
- ** govulncheck :** This tool runs in the CI/CD pipeline and reports known vulnerabilities affecting Go code or the compiler. If the operator is built with a version of the Go compiler containing a known vulnerability, `govulncheck` will detect it.

- ** CodeQL :** Provided by GitHub, this tool scans for security issues and blocks any pull request with detected vulnerabilities. CodeQL is configured to review only Go code, excluding other languages in the repository such as Python or Bash.
- ** Snyk :** Conducts nightly code scans in a scheduled job and generates weekly reports highlighting any new findings related to code security and licensing issues.

The CloudNativePG repository has the *"Private vulnerability reporting"* option enabled in the *Security section* . This feature allows users to safely report security issues that require careful handling before being publicly disclosed. If you discover any security bug, please use this medium to report it.

Important

A failure in the static code analysis phase of the CI/CD pipeline will block the entire delivery process of CloudNativePG. Every commit must pass all the linters defined by GolangCI-Lint.

10.2 Container

Every container image in CloudNativePG is automatically built via CI/CD pipelines after every commit. These images include not only the operator's image but also the operands' images, specifically for every supported PostgreSQL version.

Important

All operand images are automatically and regularly rebuilt by our pipelines to incorporate the latest security updates at both the base image and package levels. This ensures that container images distributed to the community receive **patch-level updates** regularly.

During the CI/CD process, images are scanned using the following tools:

- ** Dockle :** Ensures best practices in the container build process.
- ** Snyk :** Detects security issues within the container and reports findings via the GitHub interface.

10.2.1 Image Signatures

The operator and operandimages are cryptographically signed using cosign , a signature tool from sigstore . This process is automated via GitHub Actions and leverages *short-lived tokens issued through OpenID Connect* .

The token issuer is `https://token.actions.githubusercontent.com` , and the signing identity corresponds to a GitHub workflow executed under the cloudnative-pg repository. This workflow uses the cosign-installer action

to streamline the signing process.

To verify the authenticity of an operator image, use the following `cosign` command with the image digest:

```
cosign verify ghcr.io/cloudnative-pg/cloudnative-pg@sha256:<DIGEST> \
  --certificate-identity-regexp="^https://github.com/cloudnative-pg/cloudnative-pg/" \
  --certificate-oidc-issuer="https://token.actions.githubusercontent.com"
```

10.2.2 Attestations

Container images include the following attestations for transparency and traceability:

- ** Software Bill of Materials (SBOM) :** A comprehensive list of software artifacts included in the image or used during its build process, formatted using the in-toto SPDX predicate standard .
- ** Provenance :** Metadata detailing how the image was built, following the SLSA Provenance

framework.

You can retrieve the SBOM for a specific image and platform using the following command:

```
docker buildx imagetools inspect <IMAGE> \
  --format {{ json (index .SBOM "<PLATFORM>").SPDX }}
```

This command outputs the SBOM in JSON format, providing a detailed view of the software components and build dependencies.

For the provenance, use:

```
docker buildx imagetools inspect <IMAGE> \
  --format {{ json (index .Provenance "<PLATFORM>").SLSA }}
```

10.2.3 Guidelines and Frameworks for Container Security

The following guidelines and frameworks have been considered for ensuring container-level security:

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

See also

For more information on the approach that EDB has taken regarding security at the container level in CloudNativePG, please refer to the blog article *Security and Containers in CloudNativePG* .

10.3 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).

10.3.1 Role Based Access Control (RBAC)

The operator interacts with the Kubernetes API server using a dedicated service account named `cnpg-manager` . This service account is typically installed in the operator namespace, commonly `cnpg-system` . However, the namespace may vary based on the deployment method (see the subsection below).

In the same namespace, there is a binding between the `cnpg-manager` service account and a role. The specific name and type of this role (either `Role` or `ClusterRole`) also depend on the deployment method. This role defines the necessary permissions required by the operator to function correctly. To learn more about these roles, you can use the `kubectl describe clusterrole` or `kubectl describe role` commands, depending on the deployment method.

Important

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

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

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

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

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

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

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

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

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

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

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

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

`nodes` : The operator needs to get the labels for Affinity and AntiAffinity so it can decide in which nodes a pod can be scheduled. This is useful, for example, to prevent the replicas from being scheduled in the same node - especially important if nodes are in different availability zones. This permission is also used to determine whether a node is scheduled, preventing the creation of pods on unscheduled nodes, or triggering a switchover if the primary lives in an unscheduled node.

10.3.2 Deployments and `ClusterRole` Resources

As mentioned above, each deployment method may have variations in the namespace location of the service account, as well as the names and types of role bindings and respective roles.

Via Kubernetes Manifest

When installing CloudNativePG using the Kubernetes manifest, permissions are set to `ClusterRoleBinding` by default. You can inspect the permissions required by the operator by running:

```
kubectl describe clusterrole cnpg-manager
```

Via OLM

From a security perspective, the Operator Lifecycle Manager (OLM) provides a more flexible deployment method. It allows you to configure the operator to watch either all namespaces or specific namespaces, enabling more granular permission management.

> **Note**
>
> OLM allows you to deploy the operator in its own namespace and configure it to watch specific namespaces used for CloudNativePG clusters. This setup helps to contain permissions and restrict access more effectively.

10.3.3 Why Are ClusterRole Permissions Needed?

The operator currently requires `ClusterRole` permissions to read `nodes` and `ClusterImageCatalog` objects. All other permissions can be namespace-scoped (i.e., `Role`) or cluster-wide (i.e., `ClusterRole`).

Even with these permissions, if someone gains access to the `ServiceAccount` , they will only have `get` , `list` , and `watch` permissions, which are limited to viewing resources. However, if an unauthorized user gains access to the `ServiceAccount` , it indicates a more significant security issue.

Therefore, it's crucial to prevent users from accessing the operator's `ServiceAccount` and any other `ServiceAccount` with elevated permissions.

10.3.4 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

10.3.5 Pod and Container Security Contexts

A Security Context

defines privilege and access control settings for a pod or container.

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

Likewise, Volume access does not require *privileged* mode nor `root` privileges. Proper permissions must be 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 manages the setting of security contexts for all pods and containers of a PostgreSQL cluster. The *Seccomp Profile* to be used for the PostgreSQL containers can be configured with the `spec.seccompProfile` section of the `Cluster` resource. If this section is left blank, the containers will use a seccompProfile `Type` of `RuntimeDefault` , that is, the container runtime default.

The security context of PostgreSQL containers using the default `seccompProfile` will look like this:

```
securityContext:
  allowPrivilegeEscalation: false
  capabilities:
    drop:
    - ALL
  privileged: false
  readOnlyRootFilesystem: true
  runAsNonRoot: true
```

(continues on next page)

(continued from previous page)

```
  seccompProfile:
    type: RuntimeDefault
```

10.3.6 Security Context Constraints

When running in an environment that is utilizing *Security Context Constraints (SCC)*

the operator does not explicitly set the security context of the PostgreSQL cluster pods, but rather allows the pods to inherit the restricted Security Context Constraints that are already defined.

10.3.7 Restricting Pod access using AppArmor

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

See also

edb_notranlate_7

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.

10.3.8 Network Policies

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

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

Important

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

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

section of the Kubernetes documentation for further information.

10.3.9 Exposed Ports

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

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

10.3.10 PostgreSQL

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

Warning

enableSuperuserAccess is set to *false* by default to improve the security-by-default posture of the operator, fostering a microservice approach where changes to PostgreSQL are performed in a declarative way through the *spec* of the *Cluster* resource, while providing developers with full powers inside the database through the database owner user.

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

Important

Please refer to the Password authentication

section in the PostgreSQL documentation for details.

Note

The operator supports toggling the *enableSuperuserAccess* option. When you disable it on a running cluster, the operator will ignore the content of the secret, remove it (if previously generated by the operator) and set the password of the *postgres* user to *NULL* (de facto disabling remote access through password authentication).

See the *Secrets* for more information.

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

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

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

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

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

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

Important

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

10.3.11 Storage

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

CHAPTER 11

Postgres instance manager

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

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

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

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

11.1 Startup Probe

The startup probe ensures that a PostgreSQL instance, whether a primary or standby, has fully started.

> **Note**
>
> By default, the startup probe uses pg_isready . However, the behavior can be customized by specifying a different startup strategy.

While the startup probe is running, the liveness and readiness probes remain disabled. Following Kubernetes standards, if the startup probe fails, the kubelet will terminate the container, which will then be restarted.

The `.spec.startDelay` parameter specifies the maximum time, in seconds, allowed for the startup probe to succeed.

By default, the `startDelay` is set to `3600` seconds. It is recommended to adjust this setting based on the time PostgreSQL needs to fully initialize in your specific environment.

> **Warning**
>
> Setting *.spec.startDelay* too low can cause the liveness probe to activate prematurely, potentially resulting in unnecessary Pod restarts if PostgreSQL hasn't fully initialized.

CloudNativePG configures the startup probe with the following default parameters:

```
failureThreshold: FAILURE_THRESHOLD
periodSeconds: 10
successThreshold: 1
timeoutSeconds: 5
```

The `failureThreshold` value is automatically calculated by dividing `startDelay` by `periodSeconds` .

You can customize any of the probe settings in the `.spec.probes.startup` section of your configuration.

> **Warning**
>
> Be sure that any custom probe settings are tailored to your cluster's operational requirements to avoid unintended disruptions.

> **Note**
>
> For more details on probe configuration, refer to the *probe API documentation* .

If you manually specify `.spec.probes.startup.failureThreshold` , it will override the default behavior and disable the automatic use of `startDelay` .

For example, the following configuration explicitly sets custom probe parameters, bypassing `startDelay` :

```
#  ... snip

spec:
  probes:
    startup:
      periodSeconds: 3
      timeoutSeconds: 3
      failureThreshold: 10
```

11.1.1 Startup Probe Strategy

In certain scenarios, you may need to customize the startup strategy for your PostgreSQL cluster. For example, you might delay marking a replica as started until it begins streaming from the primary or define a replication lag threshold that must be met before considering the replica ready.

To accommodate these requirements, CloudNativePG extends the `.spec.probes.startup` stanza with two optional parameters:

- `type` : specifies the criteria for considering the probe successful. Accepted values, in increasing order of complexity/depth, include:

  - `pg_isready`: marks the probe as successful when the `pg_isready` command exits with `0`. This is the default for primary instances and replicas.
  - `query`: marks the probe as successful when a basic query is executed on the `postgres` database locally.
  - `streaming`: marks the probe as successful when the replica begins streaming from its source and meets the specified lag requirements (details below).
- `maximumLag`: defines the maximum acceptable replication lag, measured in bytes (expressed as Kubernetes quantities). This parameter is only applicable when `type` is set to `streaming`. If `maximumLag` is not specified, the replica is considered successfully started as soon as it begins streaming.

Important

The *.spec.probes.startup.maximumLag* option is validated and enforced only during the startup phase of the pod, meaning it applies exclusively when the replica is starting.

Warning

Incorrect configuration of the *maximumLag* option can cause continuous failures of the startup probe, leading to repeated replica restarts. Ensure you understand how this option works and configure appropriate values for *failureThreshold* and *periodSeconds* to give the replica enough time to catch up with its source.

The following example requires a replica to have a maximum lag of 16Mi from the source to be considered started:

```
#  <snip>

probes:
  startup:
    type: streaming
    maximumLag: 16Mi
```

11.2 Liveness Probe

The liveness probe begins after the startup probe successfully completes. Its primary role is to ensure the PostgreSQL instance manager is operating correctly.

Following Kubernetes standards, if the liveness probe fails, the kubelet will terminate the container, which will then be restarted.

The amount of time before a Pod is classified as not alive is configurable via the `.spec.livenessProbeTimeout` parameter.

CloudNativePG configures the liveness probe with the following default parameters:

```
failureThreshold: FAILURE_THRESHOLD
periodSeconds: 10
successThreshold: 1
timeoutSeconds: 5
```

The `failureThreshold` value is automatically calculated by dividing `livenessProbeTimeout` by `periodSeconds`.

By default, `.spec.livenessProbeTimeout` is set to `30` seconds. This means the liveness probe will report a failure if it detects three consecutive probe failures, with a 10-second interval between each check.

You can customize any of the probe settings in the `.spec.probes.liveness` section of your configuration.

Warning

Be sure that any custom probe settings are tailored to your cluster's operational requirements to avoid unintended disruptions.

Note

For more details on probe configuration, refer to the *probe API documentation* .

If you manually specify `.spec.probes.liveness.failureThreshold` , it will override the default behavior and disable the automatic use of `livenessProbeTimeout` .

For example, the following configuration explicitly sets custom probe parameters, bypassing `livenessProbeTimeout` :

```
#  ... snip

spec:
  probes:
    liveness:
      periodSeconds: 3
      timeoutSeconds: 3
      failureThreshold: 10
```

11.2.1 Primary Isolation

CloudNativePG 1.27 introduces an additional behavior for the liveness probe of a PostgreSQL primary, which will report a failure if **both** of the following conditions are met:

1. The instance manager cannot reach the Kubernetes API server
2. The instance manager cannot reach **any** other instance via the instance manager's REST API

The effect of this behavior is to consider an isolated primary to be not alive and subsequently **shut it down** when the liveness probe fails.

It is **enabled by default** and can be disabled by adding the following:

```
spec:
  probes:
    liveness:
      isolationCheck:
        enabled: false
```

Important

Be aware that the default liveness probe settings—automatically derived from *livenessProbeTimeout* —might be aggressive (30 seconds). As such, we recommend explicitly setting the liveness probe configuration to suit your

environment.

The spec also accepts two optional network settings: `requestTimeout` and `connectionTimeout` , both defaulting to `1000` (in milliseconds). In cloud environments, you may need to increase these values. For example:

```
spec:
  probes:
    liveness:
      isolationCheck:
        enabled: true
        requestTimeout: "2000"
        connectionTimeout: "2000"
```

11.3 Readiness Probe

The readiness probe starts once the startup probe has successfully completed. Its primary purpose is to check whether the PostgreSQL instance is ready to accept traffic and serve requests at any point during the pod's lifecycle.

Note

By default, the readiness probe uses pg_isready . However, the behavior can be customized by specifying a different readiness strategy.

Following Kubernetes standards, if the readiness probe fails, the pod will be marked unready and will not receive traffic from any services. An unready pod is also ineligible for promotion during automated failover scenarios.

CloudNativePG uses the following default configuration for the readiness probe:

```
failureThreshold: 3
periodSeconds: 10
successThreshold: 1
timeoutSeconds: 5
```

If the default settings do not suit your requirements, you can fully customize the readiness probe by specifying parameters in the `.spec.probes.readiness` stanza. For example:

```
#  ... snip

spec:
  probes:
    readiness:
      periodSeconds: 3
      timeoutSeconds: 3
      failureThreshold: 10
```

Warning

Ensure that any custom probe settings are aligned with your cluster's operational requirements to prevent unintended disruptions.

> **Note**
>
> For more information on configuring probes, see the *probe API* .

11.3.1 Readiness Probe Strategy

In certain scenarios, you may need to customize the readiness strategy for your cluster. For example, you might delay marking a replica as ready until it begins streaming from the primary or define a maximum replication lag threshold before considering the replica ready.

To accommodate these requirements, CloudNativePG extends the `.spec.probes.readiness` stanza with two optional parameters: `type` and `maximumLag` . Please refer to the *Startup Probe Strategy*

section for detailed information on these options.

> **Important**
>
> Unlike the startup probe, the *.spec.probes.readiness.maximumLag* option is continuously monitored. A lagging replica may become unready if this setting is not appropriately tuned.

> **Warning**
>
> Incorrect configuration of the *maximumLag* option can lead to repeated readiness probe failures, causing serious consequences, such as:

```
- Exclusion of the replica from key operator features, such as promotion during␣
↪failover or participation in synchronous replication quorum. - Disruptions in read/
↪read-only services. - In longer failover times scenarios, replicas might be␣
↪declared unready, leading to a cluster stall requiring manual intervention.
```

The following example requires a replica to have a maximum lag of 64Mi from the source to be considered ready. It also provides approximately 300 seconds (30 failures × 10 seconds) for the startup probe to succeed:

```
#  <snip>

probes:
  readiness:
    type: streaming
    maximumLag: 64Mi
    failureThreshold: 30
    periodSeconds: 10
```

11.4 Shutdown control

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

The shutdown procedure is composed of two steps:

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

Important

In order to avoid any data loss in the Postgres cluster, which impacts the database PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode} , don't delete the Pod where the primary instance is running. In this case, perform a switchover to another instance first.

11.4.1 Shutdown of the primary during a switchover

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

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

Warning

The *.spec.switchoverDelay* option affects the PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode}

and *RTO and RPO impact* 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.

11.5 Failover

In case of primary pod failure, the cluster will go into failover mode. Please refer to the FailoverQuorum {postgresql-cnpg-io-v1-FailoverQuorum} for details.

11.6 Disk Full Failure

Storage exhaustion is a well known issue for PostgreSQL clusters. The PostgreSQL documentation

highlights the possible failure scenarios and the importance of monitoring disk usage to prevent it from becoming full.

The same applies to CloudNativePG and Kubernetes as well: the ClusterMonitoringTLSConfiguration {postgresql-cnpg-io-v1-ClusterMonitoringTLSConfiguration}

provides details on checking the disk space used by WAL segments and standard metrics on disk usage exported to Prometheus.

Important

In a production system, it is critical to monitor the database continuously. Exhausted disk storage can lead to a database server shutdown.

> **Note**
>
> The detection of exhausted storage relies on a storage class that accurately reports disk size and usage. This may not be the case in simulated Kubernetes environments like Kind or with test storage class implementations such as *csi-driver-host-path* .

If the disk containing the WALs becomes full and no more WAL segments can be stored, PostgreSQL will stop working. CloudNativePG correctly detects this issue by verifying that there is enough space to store the next WAL segment, and avoids triggering a failover, which could complicate recovery.

That allows a human administrator to address the root cause.

In such a case, if supported by the storage class, the quickest course of action is currently to:

1. Expand the storage size of the full PVC
2. Increase the size in the `Cluster` resource to the same value

Once the issue is resolved and there is sufficient free space for WAL segments, the Pod will restart and the cluster will become healthy.

See also the *Volume expansion* of the documentation.

CHAPTER 12

Scheduling

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

> **See also**
>
> Please refer to the *Kubernetes documentation*

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

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

section in the definition of the cluster, which supports:

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

12.1 Pod Affinity and Anti-Affinity

Kubernetes provides mechanisms to control where pods are scheduled using *affinity* and *anti-affinity* rules. These rules allow you to specify whether a pod should be scheduled on particular nodes (*affinity*) or avoided on specific nodes (*anti-affinity*) based on the workloads already running there. This capability is technically referred to as **inter-pod affinity/anti-affinity**.

By default, CloudNativePG configures cluster instances to preferably be scheduled on different nodes, while `pg-Bouncer` instances might still run on the same nodes.

For example, given the following `Cluster` specification:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:17.5

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

  storage:
    size: 1Gi
```

The `affinity` configuration applied in the instance pods will be:

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

With this setup, Kubernetes will *prefer* to schedule a 3-node PostgreSQL cluster across three different nodes, assuming sufficient resources are available.

12.1.1 Requiring Pod Anti-Affinity

You can modify the default behavior by adjusting the settings mentioned above.

For example, setting `podAntiAffinityType` to `required` will enforce `requiredDuringSchedulingIgnoredDuringExecution` instead of `preferredDuringSchedulingIgnoredDuringExecution`.

However, be aware that this strict requirement may cause pods to remain pending if resources are insufficient—this is particularly relevant when using Cluster Autoscaler for automated horizontal scaling in a Kubernetes cluster.

See also

For more details, refer to the *Kubernetes documentation*.

12.1.2 Topology Considerations

In cloud environments, you might consider using `topology.kubernetes.io/zone` as the `topologyKey` to ensure pods are distributed across different availability zones rather than just nodes. For more options, see Well-Known Labels, Annotations, and Taints .

12.1.3 Disabling Anti-Affinity Policies

If needed, you can disable the operator-generated anti-affinity policies by setting `enablePodAntiAffinity` to `false` .

12.1.4 Fine-Grained Control with Custom Rules

For scenarios requiring more precise control, you can specify custom pod affinity or anti-affinity rules using the `additionalPodAffinity` and `additionalPodAntiAffinity` configuration attributes. These custom rules will be added to those generated by the operator, if enabled, or used directly if the operator-generated rules are disabled.

Note

When using *additionalPodAntiAffinity* or *additionalPodAffinity* , you must provide the full *podAntiAffinity* or *podAffinity* structure expected by the Pod specification. The following YAML example demonstrates how to configure only one instance of PostgreSQL per worker node, regardless of which PostgreSQL cluster it belongs to:

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

12.2 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.

12.3 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.

See also

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

12.4 Isolating PostgreSQL workloads

Important

Before proceeding, please ensure you have read the *Architecture* section of the documentation.

While you can deploy PostgreSQL on Kubernetes in various ways, we recommend following these essential principles for production environments:

- **Exploit Availability Zones:** If possible, take advantage of availability zones (AZs) within the same Kubernetes cluster by distributing PostgreSQL instances across different AZs.
- **Dedicate Worker Nodes:** Allocate specific worker nodes for PostgreSQL workloads through the `node-role.kubernetes.io/postgres` label and taint, as detailed in the *Reserving nodes for PostgreSQL workloads*

section.

- **Avoid Node Overlap:** Ensure that no instances from the same PostgreSQL cluster are running on the same node.

As explained in greater detail in the previous sections, CloudNativePG provides the flexibility to configure pod anti-affinity, node selectors, and tolerations.

Below is a sample configuration to ensure that a PostgreSQL `Cluster` is deployed on `postgres` nodes, with its instances distributed across different nodes:

```
# <snip>
affinity:
  enablePodAntiAffinity: true
  topologyKey: kubernetes.io/hostname
  podAntiAffinityType: required
  nodeSelector:
    node-role.kubernetes.io/postgres: ""
  tolerations:
  - key: node-role.kubernetes.io/postgres
    operator: Exists
    effect: NoSchedule
# <snip>
```

Despite its simplicity, this setup ensures optimal distribution and isolation of PostgreSQL workloads, leading to enhanced performance and reliability in your production environment.

CHAPTER 13

Resource management

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

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

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

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

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

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

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

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

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

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

section in the Kubernetes documentation.

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

> **Note**
>
> When the quality of service is set to "Guaranteed", CloudNativePG sets the *PG_OOM_ADJUST_VALUE* for the *postmaster* process to *0* , in line with the PostgreSQL documentation . This allows the *postmaster* to retain its low Out-Of-Memory (OOM) score of *-997* , while its child processes run with an OOM score adjustment of *0* . As a result, if the OOM killer is triggered, it will terminate the child processes before the *postmaster* . This behavior helps keep the PostgreSQL instance alive for as long as possible and enables a clean shutdown procedure in the event of an eviction.

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

  storage:
    size: 1Gi
```

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

A reasonable starting value for `shared_buffers` is 25% of the memory in your system. For example: if your

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

section in the PostgreSQL documentation.

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

page from the Kubernetes documentation.

CHAPTER 14

Failure Modes

Note

In previous versions of CloudNativePG, this page included specific failure scenarios. Since these largely follow standard Kubernetes behavior, we have streamlined the content to avoid duplication of information that belongs to the underlying Kubernetes stack and is not specific to CloudNativePG.

CloudNativePG adheres to standard Kubernetes principles for self-healing and high availability. We assume familiarity with core Kubernetes concepts such as storage classes, PVCs, nodes, and Pods. For CloudNativePG-specific details, refer to the *Postgres instance manager* , which covers startup, liveness, and readiness probes, as well as the *Self-Healing* section below.

Important

If you are running CloudNativePG in production, we strongly recommend seeking professional support .

14.1 Self-Healing

14.1.1 Primary Failure

If the primary Pod fails:

- The operator promotes the most up-to-date standby with the lowest replication lag.
- The `-rw` service is updated to point to the new primary.
- The failed Pod is removed from the `-r` and `-rw` services.
- Standby Pods begin replicating from the new primary.
- The former primary uses `pg_rewind` to re-synchronize if its PVC is available; otherwise, a new standby is created from a backup of the new primary.

14.1.2 Standby Failure

If a standby Pod fails:

- It is removed from the `-r` and `-ro` services.
- The Pod is restarted using its PVC if available; otherwise, a new Pod is created from a backup of the current primary.
- Once ready, the Pod is re-added to the `-r` and `-ro` services.

14.2 Manual Intervention

For failure scenarios not covered by automated recovery, manual intervention may be required.

Important

Do not perform manual operations without professional support .

14.2.1 Disabling Reconciliation

To temporarily disable the reconciliation loop for a PostgreSQL cluster, use the `cnpg.io/reconciliationLoop` annotation:

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

Use this annotation **with extreme caution** and only during emergency operations.

Warning

This annotation should be removed as soon as the issue is resolved. Leaving it in place prevents the operator from executing self-healing actions, including failover.

CHAPTER 15

Rolling Updates

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

Important

Only upgrades for PostgreSQL minor releases are supported.

Rolling upgrades are started when:

- the user changes the `imageName` attribute of the cluster specification;
- the *Image Catalog* is updated with a new image for the major used by the cluster;
- a change in the PostgreSQL configuration requires a restart to be applied;
- a change on the `Cluster .spec.resources` values
- a change in size of the persistent volume claim on AKS
- after the operator is updated, to ensure the Pods run the latest instance manager (unless *in-place updates are enabled*).

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

The primary is the last node to be upgraded.

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

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

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

15.1 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:

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

There's no one-size-fits-all configuration for the update method, as that depends on several factors like the actual workload of your database, the requirements in terms of PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode} and *RTO and RPO impact*, 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.

15.2 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. Keep in mind that image upgrades can not be applied with an in-place restart, so a switchover is required in such cases.

You can trigger a switchover with:

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

You can trigger a restart with:

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

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

CHAPTER 16

Replication

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

Important

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

16.1 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-level replication**, in contrast with *storage-level replication*.

16.2 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) introduced WAL streaming and read-only replicas through *hot standby*. In 2011, PostgreSQL 9.1 brought synchronous replication at the transaction level, supporting PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode} =0 clusters. Cascading replication was added in PostgreSQL 9.2 (2012). The foundations for *Logical Replication Publications* were established in PostgreSQL 9.4 (2014), and version

10 (2017) introduced native support for the publisher/subscriber pattern to replicate data from an origin to a destination. The table below summarizes these milestones.

Version	Year	Feature
8.2	2006	Warm Standby with WAL shipping
9.0	2010	Hot Standby and physical streaming replication
9.1	2011	Synchronous replication (priority-based)
9.2	2012	Cascading replication
9.4	2014	Foundations of logical replication
10	2017	Logical publisher/subscriber and quorum-based synchronous replication

This table highlights key PostgreSQL replication features and their respective versions.

16.3 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 map=cnpg_streaming_replica
hostssl replication streaming_replica all cert map=cnpg_streaming_replica
```

See also

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

in the documentation.

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

See also

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

below.

16.3.1 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.

16.4 Synchronous Replication

CloudNativePG supports both quorum-based and priority-based synchronous replication for PostgreSQL .

> **Warning**
>
> By default, synchronous replication pauses write operations if the required number of standby nodes for WAL replication during transaction commits is unavailable. This behavior prioritizes data durability and aligns with PostgreSQL DBA best practices. However, if self-healing is a higher priority than strict data durability in your setup, this setting can be adjusted. For details on managing this behavior, refer to the *Data Durability and Synchronous Replication*

section.

> **Important**
>
> The **failover quorum* feature* (experimental) can be used alongside synchronous replication to improve data durability and safety during failover events.

Direct configuration of the `synchronous_standby_names` option is not permitted. However, CloudNativePG automatically populates this option with the names of local pods, while also allowing customization to extend synchronous replication beyond the `Cluster` resource. This can be achieved through the *.spec.postgresql.synchronous* .

Synchronous replication is disabled by default (the `synchronous` stanza is not defined). When defined, two options are mandatory:

- `method` : either `any` (quorum) or `first` (priority)
- `number` : the number of synchronous standby servers that transactions must wait for responses from

16.4.1 Quorum-based Synchronous Replication

In PostgreSQL, quorum-based synchronous replication ensures that transaction commits wait until their WAL records are replicated to a specified number of standbys. To enable this, set the `method` to `any` .

This replication method is the most common setup for a CloudNativePG cluster.

16.4.2 Example

The example below, based on a typical `cluster-example` configuration with three instances, sets up quorum-based synchronous replication with at least one instance:

```
postgresql:
  synchronous:
    method: any
    number: 1
```

With this configuration, CloudNativePG automatically sets the content of `synchronous_standby_names` as follows:

```
ANY 1 (cluster-example-2, cluster-example-3, cluster-example-1)
```

16.4.3 Migrating from Deprecated Synchronous Replication Implementation

This section outlines how to migrate from the deprecated quorum-based synchronous replication format to the newer, more robust implementation in CloudNativePG.

Given the following manifest:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: angus
spec:
  instances: 3
  minSyncReplicas: 1
  maxSyncReplicas: 1

  storage:
    size: 1G
```

You can update it to the new format as follows:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: angus
spec:
  instances: 3

  storage:
    size: 1G

  postgresql:
    synchronous:
      method: any
      number: 1
      dataDurability: required
```

To prioritize self-healing over strict data durability, set `dataDurability` to `preferred` instead.

16.4.4 Priority-based Synchronous Replication

PostgreSQL's priority-based synchronous replication makes transaction commits wait until their WAL records are replicated to the requested number of synchronous standbys chosen based on their priorities. Standbys listed earlier in the `synchronous_standby_names` option are given higher priority and considered synchronous. If a current synchronous standby disconnects, it is immediately replaced by the next-highest-priority standby. To use this method, set `method` to `first` .

Important

Currently, this method is most useful when extending synchronous replication beyond the current cluster using the *maxStandbyNamesFromCluster* , *standbyNamesPre* , and *standbyNamesPost* options explained below.

16.4.5 Controlling `synchronous_standby_names` Content

By default, CloudNativePG populates `synchronous_standby_names` with the names of local pods in a `Cluster` resource, ensuring synchronous replication within the PostgreSQL cluster. You can customize the content of `synchronous_standby_names` based on your requirements and replication method (quorum or priority) using the following optional parameters in the `.spec.postgresql.synchronous` stanza:

- `maxStandbyNamesFromCluster` : the maximum number of pod names from the local `Cluster` object that can be automatically included in the `synchronous_standby_names` option in PostgreSQL.
- `standbyNamesPre` : a list of standby names (specifically `application_name`) to be prepended to the list of local pod names automatically listed by the operator.
- `standbyNamesPost` : a list of standby names (specifically `application_name`) to be appended to the list of local pod names automatically listed by the operator.

> **Warning**
>
> You are responsible for ensuring the correct names in *standbyNamesPre* and *standbyNamesPost* . CloudNativePG expects that you manage any standby with an *application_name* listed here, ensuring their high availability. Incorrect entries can jeopardize your PostgreSQL database uptime.

16.4.6 Examples

Here are some examples, all based on a `cluster-example` with three instances:

If you set:

```yaml
postgresql:
  synchronous:
    method: any
    number: 1
    maxStandbyNamesFromCluster: 1
    standbyNamesPre:
      - angus
```

The content of `synchronous_standby_names` will be:

```
ANY 1 (angus, cluster-example-2)
```

If you set:

```yaml
postgresql:
  synchronous:
    method: any
    number: 1
    maxStandbyNamesFromCluster: 0
    standbyNamesPre:
      - angus
      - malcolm
```

The content of `synchronous_standby_names` will be:

```
ANY 1 (angus, malcolm)
```

If you set:

```
postgresql:
  synchronous:
    method: first
    number: 2
    maxStandbyNamesFromCluster: 1
    standbyNamesPre:
      - angus
    standbyNamesPost:
      - malcolm
```

The `synchronous_standby_names` option will look like:

```
FIRST 2 (angus, cluster-example-2, malcolm)
```

16.4.7 Data Durability and Synchronous Replication

The `dataDurability` option in the `.spec.postgresql.synchronous` stanza controls the trade-off between data safety and availability for synchronous replication. It can be set to `required` or `preferred` , with the default being `required` if not specified.

Important

preferred can only be used when *standbyNamesPre* and *standbyNamesPost* are unset.

16.4.8 Required Data Durability

When `dataDurability` is set to `required` , PostgreSQL only considers transactions committed once WAL (Write-Ahead Log) records have been replicated to the specified number of synchronous standbys. This setting prioritizes data safety over availability, meaning write operations will pause if the required number of synchronous standbys is unavailable. This ensures zero data loss (RPO=0) but may reduce database availability during network disruptions or standby failures.

Synchronous standbys are selected in this priority order:

1. Healthy instances
2. Unhealthy instances
3. Primary

The list is then truncated based on `maxStandbyNamesFromCluster` if this value is set, prioritizing healthy instances and ensuring `synchronous_standby_names` is populated.

Example

Consider the following example:

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

(continues on next page)

(continued from previous page)

```
    synchronous:
      method: any
      number: 1
      dataDurability: required
```

1. Initial state. The content of `synchronous_standby_names` is:

```
ANY 1 ("foo-2","foo-3","foo-1")
```

2. `foo-2` becomes unavailable. It gets pushed back in priority:

```
ANY 1 ("foo-3","foo-2","foo-1")
```

3. `foo-3` also becomes unavailable. The list contains no healthy standbys:

```
ANY 1 ("foo-2","foo-3","foo-1")
```

At this point no write operations will be allowed until at least one of the standbys is available again.

4. When the standbys are available again, `synchronous_standby_names` will be back to the initial state.

16.4.9 Preferred Data Durability

When `dataDurability` is set to `preferred` , the required number of synchronous instances adjusts based on the number of available standbys. PostgreSQL will attempt to replicate WAL records to the designated number of synchronous standbys, but write operations will continue even if fewer than the requested number of standbys are available.

Important

Make sure you have a clear understanding of what *ready/available* means for a replica and set your expectations accordingly. By default, a replica is considered ready when it has successfully connected to the source at least once. However, CloudNativePG allows you to configure startup and readiness probes for replicas based on maximum lag. For more details, please refer to the *Postgres instance manager* .

This setting balances data safety with availability, enabling applications to continue writing during temporary standby unavailability—hence, it's also known as *self-healing mode*.

Warning

This mode may result in data loss if all standbys become unavailable.

With `preferred` data durability, **only healthy replicas** are included in `synchronous_standby_names` .

Example

Consider the following example. For demonstration, we'll use a cluster named `bar` with 5 instances and 2 synchronous standbys:

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

(continues on next page)

(continued from previous page)

```
  name: bar
spec:
  instances: 5
  postgresql:
    synchronous:
      method: any
      number: 2
      dataDurability: preferred
```

1. Initial state. The content of `synchronous_standby_names` is:

```
ANY 2 ("bar-2","bar-3", "bar-4", "bar-5")
```

2. `bar-2` and `bar-3` become unavailable. They are removed from the list:

```
ANY 2 ("bar-4", "bar-5")
```

3. `bar-4` also becomes unavailable. It gets removed from the list. Since the number of available standbys is less than the requested number, the requested amount gets reduced:

```
ANY 1 ("bar-5")
```

4. `bar-5` also becomes unavailable. `synchronous_standby_names` becomes empty, disabling synchronous replication completely. Write operations will continue, but with the risk of potential data loss in case of a primary failure.
5. When the replicas are back, `synchronous_standby_names` will be back to the initial state.

16.5 Synchronous Replication (Deprecated)

Warning

Prior to CloudNativePG 1.24, only the quorum-based synchronous replication implementation was supported. Although this method is now deprecated, it will not be removed anytime soon. The new method prioritizes data durability over self-healing and offers more robust features, including priority-based synchronous replication and full control over the *synchronous_standby_names* option. It is recommended to gradually migrate to the new configuration method for synchronous replication, as explained in the previous paragraph.

Important

The deprecated method and the new method are mutually exclusive.

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

Important

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

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

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

Where:

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

Warning

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

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

Important

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

16.5.1 Select nodes for synchronous replication

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

See also

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

Warning

The *.spec.postgresql.syncReplicaElectionConstraint* option only applies to the legacy implementation of synchronous replication (see *Synchronous Replication (Deprecated)*).

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 (PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode}).

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

Important

The self-healing enforcement still applies while defining additional constraints for synchronous replica election (see *Zero-Data-Loss Clusters Through Synchronous Replication*).

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

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

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

16.6 Replication slots

Replication slots

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

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

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

- the replication slots automatically created for the High Availability of the Postgres cluster (see *Replication slots for High Availability*

for details)

- *User-Defined Replication slots* created on the primary

16.6.1 Replication slots for High Availability

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

In CloudNativePG, we use the terms:

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

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

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

`.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)

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

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

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

  storage:
    size: 1Gi
```

16.6.2 User-Defined Replication slots

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

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

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

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

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

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

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

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

16.6.3 Synchronization frequency

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

```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:
    updateInterval: 300

  storage:
    size: 1Gi
```

16.6.4 Logical Decoding Slot Synchronization

CloudNativePG can synchronize logical decoding (replication) slots across all nodes in a high-availability cluster, ensuring seamless continuation of logical replication after a failover or switchover. This feature is disabled by default, and enabling it requires two steps.

The first step is to enable logical decoding slot synchronization:

```
# ...
replicationSlots:
  highAvailability:
    synchronizeLogicalDecoding: true
```

The second step involves configuring PostgreSQL parameters: the required configuration depends on your PostgreSQL version, as explained below.

When enabled, the operator automatically manages logical decoding slot states during failover and switchover, preventing slot invalidation and avoiding data loss for logical replication clients.

16.6.5 Behavior on PostgreSQL 17 and later

For PostgreSQL 17 and newer, CloudNativePG transparently manages the *synchronized_standby_slots* .

You must enable both `sync_replication_slots` and `hot_standby_feedback` in your PostgreSQL configuration:

```
#  ...

postgresql:
  parameters:
    # ...
    hot_standby_feedback: on
    sync_replication_slots: on
```

Additionally, you must create the logical replication `Subscription` with the `failover` option enabled, for example:

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

#  ...

spec:

#  ...

  parameters:
    failover: true

#  ...
```

When configured, logical WAL sender processes send decoded changes to plugins only after the specified replication slots confirm receiving and flushing the relevant WAL, ensuring that:

- logical replication slots do not consume changes until they are safely received by the replicas of the publisher, and
- logical replication clients can seamlessly reconnect to a promoted standby without missing data after failover.

For more details on logical replication slot synchronization, see the PostgreSQL documentation on *Logical Replication Failover* .

16.6.6 Behavior on PostgreSQL 16 and earlier

For PostgreSQL 16 and older versions, CloudNativePG uses the *pg_failover_slots*

to maintain synchronization of logical replication slots across failovers.

16.6.7 Capping the WAL size retained for replication slots

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

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

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

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

16.6.8 Monitoring replication slots

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

See also

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

CHAPTER 17

Logical Replication

PostgreSQL extends its replication capabilities beyond physical replication, which operates at the level of exact block addresses and byte-by-byte copying, by offering logical replication . Logical replication replicates data objects and their changes based on a defined replication identity, typically the primary key.

Logical replication uses a publish-and-subscribe model, where subscribers connect to publications on a publisher node. Subscribers pull data changes from these publications and can re-publish them, enabling cascading replication and complex topologies.

Important

To protect your logical replication subscribers after a failover of the publisher cluster in CloudNativePG, ensure that replication slot synchronization for logical decoding is enabled. Without this, your logical replication clients may lose data and fail to continue seamlessly after a failover. For configuration details, see *Replication: Logical Decoding Slot Synchronization* .

This flexible model is particularly useful for:

- Online data migrations
- Live PostgreSQL version upgrades
- Data distribution across systems
- Real-time analytics
- Integration with external applications

Note

For more details, examples, and limitations, please refer to the *official PostgreSQL documentation on Logical Replication* .

CloudNativePG enhances this capability by providing declarative support for key PostgreSQL logical replication objects:

- **Publications** via the `Publication` resource
- **Subscriptions** via the `Subscription` resource

17.1 Publications

In PostgreSQL's publish-and-subscribe replication model, a ***publication***

is the source of data changes. It acts as a logical container for the change sets (also known as *replication sets*) generated from one or more tables within a database. Publications can be defined on any PostgreSQL 10+ instance acting as the *publisher*, including instances managed by popular DBaaS solutions in the public cloud. Each publication is tied to a single database and provides fine-grained control over which tables and changes are replicated.

For publishers outside Kubernetes, you can create publications using SQL

or leverage the *cnpg publication create* .

When managing `Cluster` objects with **CloudNativePG**, PostgreSQL publications can be defined declaratively through the `Publication` resource.

Note

Please refer to the *API Reference*

for the full list of attributes you can define for each `Publication` object.

Suppose you have a cluster named `freddie` and want to replicate all tables in the `app` database. Here's a `Publication` manifest:

```
apiVersion: postgresql.cnpg.io/v1
kind: Publication
metadata:
  name: freddie-publisher
spec:
  cluster:
    name: freddie
  dbname: app
  name: publisher
  target:
    allTables: true
```

In the above example:

- The publication object is named `freddie-publisher` (`metadata.name`).
- The publication is created via the primary of the `freddie` cluster (`spec.cluster.name`) with name `publisher` (`spec.name`).
- It includes all tables (`spec.target.allTables: true`) from the `app` database (`spec.dbname`).

Important

While *allTables* simplifies configuration, PostgreSQL offers fine-grained control for replicating specific tables or targeted data changes. For advanced configurations, consult the *PostgreSQL documentation* . Additionally, refer to the *CloudNativePG API reference*

for details on declaratively customizing replication targets.

17.1.1 Required Fields in the `Publication` Manifest

The following fields are required for a `Publication` object:

- `metadata.name` : Unique name for the Kubernetes `Publication` object.
- `spec.cluster.name` : Name of the PostgreSQL cluster.
- `spec.dbname` : Database name where the publication is created.
- `spec.name` : Publication name in PostgreSQL.
- `spec.target` : Specifies the tables or changes to include in the publication.

The `Publication` object must reference a specific `Cluster` , determining where the publication will be created. It is managed by the cluster's primary instance, ensuring the publication is created or updated as needed.

17.1.2 Reconciliation and Status

After creating a `Publication` , CloudNativePG manages it on the primary instance of the specified cluster. Following a successful reconciliation cycle, the `Publication` status will reflect the following:

- `applied: true` , indicates the configuration has been successfully applied.
- `observedGeneration` matches `metadata.generation` , confirming the applied configuration corresponds to the most recent changes.

If an error occurs during reconciliation, `status.applied` will be `false` , and an error message will be included in the `status.message` field.

17.1.3 Removing a publication

The `publicationReclaimPolicy` field controls the behavior when deleting a `Publication` object:

- `retain` (default): Leaves the publication in PostgreSQL for manual management.
- `delete` : Automatically removes the publication from PostgreSQL.

Consider the following example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Publication
metadata:
  name: freddie-publisher
spec:
  cluster:
    name: freddie
  dbname: app
  name: publisher
  target:
    allTables: true
  publicationReclaimPolicy: delete
```

In this case, deleting the `Publication` object also removes the `publisher` publication from the `app` database of the `freddie` cluster.

17.2 Subscriptions

In PostgreSQL's publish-and-subscribe replication model, a ***subscription***

represents the downstream component that consumes data changes. A subscription establishes the connection to a publisher's database and specifies the set of publications (one or more) it subscribes to. Subscriptions can be created on any supported PostgreSQL instance acting as the *subscriber*.

Important

Since schema definitions are not replicated, the subscriber must have the corresponding tables already defined before data replication begins.

CloudNativePG simplifies subscription management by enabling you to define them declaratively using the `Subscription` resource.

Note

Please refer to the *API Reference*

for the full list of attributes you can define for each `Subscription` object.

Suppose you want to replicate changes from the `publisher` publication on the `app` database of the `freddie` cluster (*publisher*) to the `app` database of the `king` cluster (*subscriber*). Here's an example of a `Subscription` manifest:

```
apiVersion: postgresql.cnpg.io/v1
kind: Subscription
metadata:
  name: freddie-to-king-subscription
spec:
  cluster:
    name: king
  dbname: app
  name: subscriber
  externalClusterName: freddie
  publicationName: publisher
```

In the above example:

- The subscription object is named `freddie-to-king-subscriber` (`metadata.name`).
- The subscription is created in the `app` database (`spec.dbname`) of the `king` cluster (`spec.cluster.name`), with name `subscriber` (`spec.name`).
- It connects to the `publisher` publication in the external `freddie` cluster, referenced by `spec.externalClusterName` .

To facilitate this setup, the `freddie` external cluster must be defined in the `king` cluster's configuration. Below is an example excerpt showing how to define the external cluster in the `king` manifest:

```
externalClusters:
  - name: freddie
    connectionParameters:
      host: freddie-rw.default.svc
      user: postgres
      dbname: app
```

Note

For more details on configuring the *externalClusters* section, see the *Bootstrap* of the documentation.

As you can see, a subscription can connect to any PostgreSQL database accessible over the network. This flexibility allows you to seamlessly migrate your data into Kubernetes with nearly zero downtime. It's an excellent option for transitioning from various environments, including popular cloud-based Database-as-a-Service (DBaaS) platforms.

17.2.1 Required Fields in the `Subscription` Manifest

The following fields are mandatory for defining a `Subscription` object:

- `metadata.name` : A unique name for the Kubernetes `Subscription` object within its namespace.
- `spec.cluster.name` : The name of the PostgreSQL cluster where the subscription will be created.
- `spec.dbname` : The name of the database in which the subscription will be created.
- `spec.name` : The name of the subscription as it will appear in PostgreSQL.
- `spec.externalClusterName` : The name of the external cluster, as defined in the `spec.cluster.name` cluster's configuration. This references the publisher database.
- `spec.publicationName` : The name of the publication in the publisher database to which the subscription will connect.

The `Subscription` object must reference a specific `Cluster` , determining where the subscription will be managed. CloudNativePG ensures that the subscription is created or updated on the primary instance of the specified cluster.

17.2.2 Reconciliation and Status

After creating a `Subscription` , CloudNativePG manages it on the primary instance of the specified cluster. Following a successful reconciliation cycle, the `Subscription` status will reflect the following:

- `applied: true` , indicates the configuration has been successfully applied.
- `observedGeneration` matches `metadata.generation` , confirming the applied configuration corresponds to the most recent changes.

If an error occurs during reconciliation, `status.applied` will be `false` , and an error message will be included in the `status.message` field.

17.2.3 Removing a Subscription

The `subscriptionReclaimPolicy` field controls the behavior when deleting a `Subscription` object:

- `retain` (default): Leaves the subscription in PostgreSQL for manual management.
- `delete` : Automatically removes the subscription from PostgreSQL.

Consider the following example:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Subscription
metadata:
  name: freddie-to-king-subscription
spec:
  cluster:
    name: king
  dbname: app
  name: subscriber
  externalClusterName: freddie
  publicationName: publisher
  subscriptionReclaimPolicy: delete
```

In this case, deleting the `Subscription` object also removes the `subscriber` subscription from the `app` database of the `king` cluster.

17.2.4 Resilience to Failovers

To ensure that your logical replication subscriptions remain operational after a failover of the publisher, configure CloudNativePG to synchronize logical decoding slots across the cluster. For detailed instructions, see *Logical Decoding Slot Synchronization* .

17.3 Limitations

Logical replication in PostgreSQL has some inherent limitations, as outlined in the *official documentation* . Notably, the following objects are not replicated:

- **Database schema and DDL commands**
- **Sequence data**
- **Large objects**

17.3.1 Addressing Schema Replication

The first limitation, related to schema replication, can be easily addressed using CloudNativePG's capabilities. For instance, you can leverage the `import` bootstrap feature to copy the schema of the tables you need to replicate. Alternatively, you can manually create the schema as you would for any PostgreSQL database.

17.3.2 Handling Sequences

While sequences are not automatically kept in sync through logical replication, CloudNativePG provides a solution to be used in live migrations. You can use the Backup {postgresql-cnpg-io-v1-Backup}

to synchronize sequence values, ensuring consistency between the publisher and subscriber databases.

17.4 Example of live migration and major Postgres upgrade with logical replication

To highlight the powerful capabilities of logical replication, this example demonstrates how to replicate data from a publisher database (`freddie`) running PostgreSQL 16 to a subscriber database (`king`) running the latest PostgreSQL version. This setup can be deployed in your Kubernetes cluster for evaluation and hands-on learning.

This example illustrates how logical replication facilitates live migrations and upgrades between PostgreSQL versions while ensuring data consistency. By combining logical replication with CloudNativePG, you can easily set up, manage, and evaluate such scenarios in a Kubernetes environment.

17.4.1 Step 1: Setting Up the Publisher (`freddie`)

The first step involves creating a `freddie` PostgreSQL cluster with version 16. The cluster contains a single instance and includes an `app` database initialized with a table, `n`, storing 10,000 numbers. A logical replication publication named `publisher` is also configured to include all tables in the database.

Here's the manifest for setting up the `freddie` cluster and its publication resource:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: freddie
spec:
  instances: 1

  imageName: ghcr.io/cloudnative-pg/postgresql:16

  storage:
    size: 1Gi

  bootstrap:
    initdb:
      postInitApplicationSQL:
        - CREATE TABLE n (i SERIAL PRIMARY KEY, m INTEGER)
        - INSERT INTO n (m) (SELECT generate_series(1, 10000))
        - ALTER TABLE n OWNER TO app

  managed:
    roles:
      - name: app
        login: true
        replication: true
- --
apiVersion: postgresql.cnpg.io/v1
kind: Publication
metadata:
  name: freddie-publisher
spec:
  cluster:
    name: freddie
  dbname: app
  name: publisher
  target:
    allTables: true
```

17.4.2 Step 2: Setting Up the Subscriber (`king`)

Next, create the `king` PostgreSQL cluster, running the latest version of PostgreSQL. This cluster initializes by importing the schema from the `app` database on the `freddie` cluster using the external cluster configuration. A `Subscription` resource, `freddie-to-king-subscription`, is then configured to consume changes published by the

`publisher` on `freddie`.

Below is the manifest for setting up the `king` cluster and its subscription:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: king
spec:
  instances: 1

  storage:
    size: 1Gi

  bootstrap:
    initdb:
      import:
        type: microservice
        schemaOnly: true
        databases:
          - app
        source:
          externalCluster: freddie

  externalClusters:
  - name: freddie
    connectionParameters:
      host: freddie-rw.default.svc
      user: app
      dbname: app
    password:
      name: freddie-app
      key: password
- --
apiVersion: postgresql.cnpg.io/v1
kind: Subscription
metadata:
  name: freddie-to-king-subscription
spec:
  cluster:
    name: king
  dbname: app
  name: subscriber
  externalClusterName: freddie
  publicationName: publisher
```

Once the `king` cluster is running, you can verify that the replication is working by connecting to the `app` database and counting the records in the `n` table. The following example uses the `psql` command provided by the `cnpg` plugin for simplicity:

```
kubectl cnpg psql king -- app -qAt -c SELECT count(*) FROM n
10000
```

This command should return `10000`, confirming that the data from the `freddie` cluster has been successfully replicated to the `king` cluster.

Using the `cnpg` plugin, you can also synchronize existing sequences to ensure consistency between the publisher and subscriber. The example below demonstrates how to synchronize a sequence for the `king` cluster:

```
kubectl cnpg subscription sync-sequences king --subscription=subscriber
SELECT setval("public"."n_i_seq", 10000);

10000
```

This command updates the sequence `n_i_seq` in the `king` cluster to match the current value, ensuring it is in sync with the source database.

CHAPTER 18

Backup

Note

This section covers **physical backups** in PostgreSQL. While PostgreSQL also supports logical backups using the *pg_dump* utility, these are **not suitable for business continuity** and are **not managed** by CloudNativePG. If you still wish to use *pg_dump* , refer to the **Troubleshooting / Emergency backup* section*

for guidance.

Important

Starting with version 1.26, native backup and recovery capabilities are being **progressively phased out** of the core operator and moved to official CNPG-I plugins. This transition aligns with CloudNativePG's shift towards a **backup-agnostic architecture**, enabled by its extensible interface—**CNPG-I**—which standardizes the management of **WAL archiving**, **physical base backups**, and corresponding **recovery processes**.

CloudNativePG currently supports **physical backups of PostgreSQL clusters** in two main ways:

- **Via `CNPG-I <https://github.com/cloudnative-pg/cnpg-i/>`_ plugins**: the CloudNativePG Community officially supports the **Barman Cloud Plugin**

for integration with object storage services.

- **Natively**, with support for:
  - Object storage via Barman Cloud

  (although deprecated from 1.26 in favor of the Barman Cloud Plugin)

  - Kubernetes Volume Snapshots , if supported by the underlying storage class

Before selecting a backup strategy with CloudNativePG, it's important to familiarize yourself with the foundational concepts covered in the *Main Concepts*

section. These include WAL archiving, hot and cold backups, performing backups from a standby, and more.

18.1 Main Concepts

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

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

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

CNPG-I provides a generic and extensible interface for managing WAL archiving (both archive and restore operations), as well as the base backup and corresponding restore processes.

18.1.1 WAL archive

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

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

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

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

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

When you *configure a WAL archive* , CloudNativePG provides out-of-the-box an PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode} <= 5 minutes for disaster recovery, even across regions.

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

18.1.2 Cold and Hot backups

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

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

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

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

18.2 Comparing Available Backup Options: Object Stores vs Volume Snapshots

CloudNativePG currently supports two main approaches for physical backups:

- **Object store–based backups**, via the **Barman Cloud Plugin** or the **deprecated native integration**
- **Volume Snapshots** , using the Kubernetes CSI interface and supported storage classes

Important

CNPG-I is designed to enable third parties to build and integrate their own backup plugins. Over time, we expect the ecosystem of supported backup solutions to grow.

18.2.1 Object Store–Based Backups

Backups to an object store (e.g. AWS S3, Azure Blob, GCS):

- Always require WAL archiving
- Support hot backups only
- Do not support incremental or differential copies
- Support retention policies

18.2.2 Volume Snapshots

Native volume snapshots:

- Do not require WAL archiving, though its use is still strongly recommended in production
- Support incremental and differential copies, depending on the capabilities of the underlying storage class
- Support both hot and cold backups
- Do not support retention policies

18.2.3 Choosing Between the Two

The best approach depends on your environment and operational requirements. Consider the following factors:

- **Object store availability**: Ensure your Kubernetes cluster can access a reliable object storage solution, including a stable networking layer.
- **Storage class capabilities**: Confirm that your storage class supports CSI volume snapshots with incremental/differential features.
- **Database size**: For very large databases (VLDBs), **volume snapshots are generally preferred** as they enable faster recovery due to copy-on-write technology—this significantly improves your *Recovery Time Objective (RTO)* .

- **Data mobility**: Object store–based backups may offer greater flexibility for replicating or storing backups across regions or environments.
- **Operational familiarity**: Choose the method that aligns best with your team's experience and confidence in managing storage.

18.2.4 Comparison Summary

Feature	Object Store	Volume Snapshots
WAL archiving	Required	Recommended^1^
Cold backup	NO	OK
Hot backup	OK	OK
Incremental copy	NO	OK^2^
Differential copy	NO	OK^2^
Backup from a standby	OK	OK
Snapshot recovery	NO^3^	OK
Retention policies	OK	NO
Point-in-Time Recovery (PITR)	OK	Requires WAL archive
Underlying technology	Barman Cloud	Kubernetes API

Important

On-demand backups do **not** include Kubernetes secrets for the PostgreSQL superuser or application user. You should ensure these secrets are included in your broader Kubernetes cluster backup strategy.

18.3 Backup Methods

CloudNativePG currently supports the following backup methods for scheduled and on-demand backups:

- `plugin` – Uses a CNPG-I plugin (requires `.spec.pluginConfiguration`)
- `volumeSnapshot` – Uses native Kubernetes volume snapshots
- `barmanObjectStore` – Uses Barman Cloud for object storage

(deprecated starting with v1.26 in favor of the `Barman Cloud Plugin <https://cloudnative-pg.io/plugin-barman-cloud/> `_ , but still the default for backward compatibility)

Specify the method using the `.spec.method` field (defaults to `barmanObjectStore`).

If your cluster is configured to support volume snapshots, you can enable scheduled snapshot backups like this:

```
spec:
  method: volumeSnapshot
```

To use the Barman Cloud Plugin as the backup method, set `method: plugin` and configure the plugin accordingly. You can find an example in the *Performing a Base Backup*

18.4 Backup from a Standby

Taking a base backup involves reading the entire on-disk data set of a PostgreSQL instance, which can introduce I/O contention and impact the performance of the active workload.

To reduce this impact, **CloudNativePG supports taking backups from a standby instance**, leveraging PostgreSQL's built-in capability to perform backups from read-only replicas.

By default, backups are performed on the **most up-to-date replica** in the cluster. If no replicas are available, the backup will fall back to the **primary instance**.

> **Note**
>
> The examples in this section are focused on backup target selection and do not take the backup method (*spec.method*) into account, as it is not relevant to the scope being discussed.

18.4.1 How It Works

When `prefer-standby` is the target (the default behavior), CloudNativePG will attempt to:

1. Identify the most synchronized standby node.
2. Run the backup process on that standby.
3. Fall back to the primary if no standbys are available.

This strategy minimizes interference with the primary's workload.

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

18.4.2 Forcing Backup on the Primary

To always run backups on the primary instance, explicitly set the backup target to `primary` in the cluster configuration:

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

> **Warning**
>
> Be cautious when using *primary* as the target for **cold backups using volume snapshots**, as this will require shutting down the primary instance temporarily—interrupting all write operations. The same caution applies to single-instance clusters, even if you haven't explicitly set the target.

18.4.3 Overriding the Cluster-Wide Target

You can override the cluster-level target on a per-backup basis, using either `Backup` or `ScheduledBackup` resources. Here's an example of an on-demand backup:

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

In this example, even if the cluster's default target is `prefer-standby` , the backup will be taken from the primary instance.

18.5 Retention Policies

CloudNativePG is evolving toward a **backup-agnostic architecture**, where backup responsibilities are delegated to external **CNPG-I plugins**. These plugins are expected to offer advanced and customizable data protection features, including sophisticated retention management, that go beyond the built-in capabilities and scope of CloudNativePG.

As part of this transition, the `spec.backup.retentionPolicy` field in the `Cluster` resource is **deprecated** and will be removed in a future release.

For more details on available retention features, refer to your chosen plugin's documentation. For example: Retention Policies .

Important

Users are encouraged to rely on the retention mechanisms provided by the backup plugin they are using. This ensures better flexibility and consistency with the backup method in use.

CHAPTER 19

WAL archiving

Write-Ahead Log (WAL) archiving in CloudNativePG is the process of continuously shipping WAL files to a designated object store from the PostgreSQL primary. These archives are essential for enabling Point-In-Time Recovery (PITR) and are a foundational component for both object store and volume snapshot-based backup strategies.

19.1 Plugin-Based Architecture

CloudNativePG supports WAL archiving through a **plugin-based mechanism**, defined via the *spec.pluginConfiguration*

section of the `Cluster` resource.

Only **one plugin at a time** can be responsible for WAL archiving. This is configured by setting the `isWALArchiver` field to `true` within the plugin configuration.

19.2 Supported Plugins

Currently, the **Barman Cloud Plugin** is the only officially supported WAL archiving plugin maintained by the CloudNativePG Community. For full documentation, configuration options, and best practices, see the Barman Cloud Plugin documentation .

19.3 Deprecation Notice: Native Barman Cloud

CloudNativePG still supports WAL archiving natively through the `.spec.backup.barmanObjectStore` field. While still functional, **this interface is deprecated** and will be removed in a future release.

> **Important**
>
> All new deployments are strongly encouraged to adopt the plugin-based architecture, which offers a more flexible and maintainable approach.

If you are currently using the native `.spec.backup.barmanObjectStore` approach, refer to the official guide for a smooth transition: Migrating from Built-in CloudNativePG Backup .

19.4 About the archive timeout

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 (PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode}).

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.

CHAPTER 20

Recovery

In PostgreSQL, **recovery** refers to the process of starting an instance from an existing physical backup. PostgreSQL's recovery system is robust and feature-rich, supporting **Point-In-Time Recovery (PITR)**—the ability to restore a cluster to any specific moment, from the earliest available backup to the latest archived WAL file.

> **Important**
>
> A valid WAL archive is required to perform PITR.

In CloudNativePG, recovery is **not performed in-place** on an existing cluster. Instead, it is used to **bootstrap a new cluster** from a physical backup.

> **Note**
>
> For more details on configuring the *bootstrap* stanza, refer to *Bootstrap* .

The `recovery` bootstrap mode allows you to initialize a cluster from a physical base backup and replay the associated WAL files to bring the system to a consistent and optionally point-in-time state.

CloudNativePG supports recovery via:

- A **pluggable backup and recovery interface (CNPG-I)**, enabling integration with external tools such as the Barman Cloud Plugin .
- **Native recovery from volume snapshots**, where supported by the underlying Kubernetes storage infrastructure.
- **Native recovery from object stores via Barman Cloud**, which is **deprecated** as of version 1.26 in favor of the plugin-based approach.

With the deprecation of native Barman Cloud support in version 1.26, this section now focuses on two supported recovery methods: using the **Barman Cloud Plugin** for recovery from object stores, and the **native interface** for recovery from volume snapshots.

Important

For legacy documentation, see Appendix B – Recovery from an Object Store .

20.1 Recovery from an Object Store with the Barman Cloud Plugin

This section outlines how to recover a PostgreSQL cluster from an object store using the recommended Barman Cloud Plugin.

Important

The object store must contain backup data produced by a CloudNativePG *Cluster* —either using the **deprecated native Barman Cloud integration** or the **Barman Cloud Plugin**.

Note

For full details, refer to the *"Recovery of a Postgres Cluster" section in the Barman Cloud Plugin documentation* .

Begin by defining the object store that holds both your base backups and WAL files. The Barman Cloud Plugin uses a custom `ObjectStore` resource for this purpose. The following example shows how to configure one for Azure Blob Storage:

```
apiVersion: barmancloud.cnpg.io/v1
kind: ObjectStore
metadata:
  name: cluster-example-backup
spec:
  configuration:
    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
```

Next, configure the `Cluster` resource to use the `ObjectStore` you defined. In the `bootstrap` section, specify the recovery source, and define an `externalCluster` entry that references the plugin:

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

(continues on next page)

(continued from previous page)

```
  superuserSecret:
    name: superuser-secret

  bootstrap:
    recovery:
      source: origin

  externalClusters:
    - name: origin
      plugin:
        name: barman-cloud.cloudnative-pg.io
        parameters:
          barmanObjectName: cluster-example-backup
          serverName: cluster-example
```

20.2 Recovery from `VolumeSnapshot` Objects

> **Warning**
>
> When creating replicas after recovering a primary instance from a *VolumeSnapshot* , the operator may fall back to using *pg_basebackup* to synchronize them. This process can be significantly slower—especially for large databases—because it involves a full base backup. This limitation will be addressed in the future with support for online backups and PVC cloning in the scale-up process.

CloudNativePG allows you to create a new cluster from a `VolumeSnapshot` of a `PersistentVolumeClaim` (PVC) that belongs to an existing `Cluster` . These snapshots are created using the declarative API for How to configure Volume Snapshot backups .

To complete the recovery process, the new cluster must also reference an external cluster that provides access to the WAL archive needed to reapply changes and finalize the recovery.

The following example shows a cluster being recovered using both a `VolumeSnapshot` for the base backup and a WAL archive accessed through the Barman Cloud Plugin:

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

  bootstrap:
    recovery:
      source: origin
      volumeSnapshots:
        storage:
          name: <snapshot name>
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io
```

(continues on next page)

(continued from previous page)

```
  externalClusters:
    - name: origin
      plugin:
        name: barman-cloud.cloudnative-pg.io
        parameters:
          barmanObjectName: cluster-example-backup
          serverName: cluster-example
```

In case the backed-up cluster was using a separate PVC to store the WAL files, the recovery must include that too:

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

  bootstrap:
    recovery:
      volumeSnapshots:
        storage:
          name: <snapshot name>
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io

        walStorage:
          name: <snapshot name>
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io
```

The previous example assumes that the application database and its owning user are named `app` by default. If the PostgreSQL cluster being restored uses different names, you must specify these names before exiting the recovery phase, as documented in *Configure the application database* .

> **Warning**
>
> If bootstrapping a replica-mode cluster from snapshots, to leverage snapshots for the standby instances and not just the primary, we recommend that you:

1. Start with a single instance replica cluster. The primary instance will be recovered using the snapshot, and available WALs from the source cluster. 2. Take a snapshot of the primary in the replica cluster. 3. Increase the number of instances in the replica cluster as desired.

20.3 Recovery from a `Backup` object

If a `Backup` resource is already available in the namespace in which you need to create the cluster, you can specify the name using `.spec.bootstrap.recovery.backup.name` , as in the following example:

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

(continues on next page)

(continued from previous page)

```
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    recovery:
      backup:
        name: backup-example

  storage:
    size: 1Gi
```

This bootstrap method allows you to specify just a reference to the backup that needs to be restored.

The previous example assumes that the application database and its owning user are named `app` by default. If the PostgreSQL cluster being restored uses different names, you must specify these names before exiting the recovery phase, as documented in *Configure the application database* .

20.4 Additional Considerations

Whether you recover from an object store, a volume snapshot, or an existing `Backup` resource, no changes to the database, including the catalog, are permitted until the `Cluster` is fully promoted to primary and accepts write operations. This restriction includes any role overrides, which are deferred until the `Cluster` transitions to primary. As a result, the following considerations apply:

- The application database name and user are copied from the backup being restored. The operator does not currently back up the underlying secrets, as this is part of the usual maintenance activity of the Kubernetes cluster.
- To preserve the original postgres user password, configure `enableSuperuserAccess` and supply a `superuserSecret` .

By default, recovery continues up to the latest available WAL on the default target timeline (`latest`). You can optionally specify a `recoveryTarget` to perform a point-in-time recovery (see *Point in time recovery (PITR)*).

Important

Consider using the *barmanObjectStore.wal.maxParallel* option to speed up WAL fetching from the archive by concurrently downloading the transaction logs from the recovery object store.

20.5 Point in time recovery (PITR)

Instead of replaying all the WALs up to the latest one, after extracting a base backup, you can ask PostgreSQL to stop replaying WALs at any given point in time. PostgreSQL uses this technique to achieve PITR. The presence of a WAL archive is mandatory.

Important

PITR requires you to specify a recovery target by using the options described in *Recovery targets* .

The operator generates the configuration parameters required for this feature to work if you specify a recovery target.

20.5.1 PITR from an object store

This example uses the same recovery object store in Azure defined earlier for the Barman Cloud plugin, containing both the base backups and the WAL archive. The recovery target is based on a requested timestamp.

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

  storage:
    size: 5Gi

  bootstrap:
    recovery:
      # Recovery object store containing WAL archive and base backups
      source: origin
      recoveryTarget:
        # Time base target for the recovery
        targetTime: "2023-08-11 11:14:21.00000+02"

  externalClusters:
    - name: origin
      plugin:
        name: barman-cloud.cloudnative-pg.io
        parameters:
          barmanObjectName: cluster-example-backup
          serverName: cluster-example
```

In this example, you had to specify only the `targetTime` in the form of a timestamp. You didn't have to specify the base backup from which to start the recovery.

The `backupID` option is the one that allows you to specify the base backup from which to initiate the recovery process. By default, this value is empty.

If you assign a value to it (in the form of a Barman backup ID), the operator uses that backup as the base for the recovery.

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

If you don't specify the backup ID, the operator detects the base backup for the recovery as follows:

- When you use `targetTime` or `targetLSN` , the operator selects the closest backup that was completed before that target.
- Otherwise, the operator selects the last available backup, in chronological order.

20.5.2 Point-in-Time Recovery (PITR) from `VolumeSnapshot` Objects

The following example demonstrates how to perform a **Point-in-Time Recovery (PITR)** using:

- A Kubernetes `VolumeSnapshot` of the `PGDATA` directory, which provides the base backup. This snapshot is specified in the `recovery.volumeSnapshots` section and is named `test-snapshot-1` .
- A recovery object store (in this case, MinIO) containing the archived WAL files. The object store is defined via a Barman Cloud Plugin `ObjectStore` resource (not shown here), and referenced using the `recovery.source` field, which points to an external cluster configuration.

The cluster will be restored to a specific point in time using the `recoveryTarget.targetTime` option.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-snapshot
spec:
  # ...
  bootstrap:
    recovery:
      source: origin
      volumeSnapshots:
        storage:
          name: test-snapshot-1
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io
      recoveryTarget:
        targetTime: "2023-07-06T08:00:39"
  externalClusters:
    - name: origin
      plugin:
        name: barman-cloud.cloudnative-pg.io
        parameters:
          barmanObjectName: minio-backup
          serverName: cluster-example
```

This setup enables CloudNativePG to restore the base data from a volume snapshot and apply WAL segments from the object store to reach the desired recovery target.

Note

If the backed-up cluster had *walStorage* enabled, you also must specify the volume snapshot containing the *PGWAL* directory, as mentioned in *Recovery from VolumeSnapshot objects* .

Warning

It's your responsibility to ensure that the end time of the base backup in the volume snapshot is before the recovery target timestamp.

Warning

If you added or removed a TablespaceConfiguration {postgresql-cnpg-io-v1-TablespaceConfiguration} in your cluster since the last base backup, replaying the WAL will fail. You need a base backup between the time of the tablespace change and the recovery target timestamp.

20.5.3 Recovery targets

Here are the recovery target criteria you can use:

targetTime : Time stamp up to which recovery proceeds, expressed in RFC 3339 format. (The precise stopping point is also influenced by the `exclusive` option.)

Warning

PostgreSQL recovery will stop when it encounters the first transaction that occurs after the specified time. If no such transaction exists after the target time, the recovery process will fail.

targetXID : Transaction ID up to which recovery proceeds. (The precise stopping point is also influenced by the `exclusive` option.) Keep in mind that while transaction IDs are assigned sequentially at transaction start, transactions can complete in a different numeric order. The transactions that are recovered are those that committed before (and optionally including) the specified one.

targetName : Named restore point (created with `pg_create_restore_point()`) to which recovery proceeds.

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

targetImmediate : Recovery ends as soon as a consistent state is reached, that is, as early as possible. When restoring from an online backup, this means the point where taking the backup ended.

Important

The operator can retrieve the closest backup when you specify either *targetTime* or *targetLSN* . However, this isn't possible for the remaining targets: *targetName* , *targetXID* , and *targetImmediate* . In such cases, it's mandatory to specify *backupID* .

This example uses a `targetName` -based recovery target:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
[...]
  bootstrap:
    recovery:
      source: origin
      recoveryTarget:
        backupID: 20220616T142236
        targetName: restore_point_1
[...]
```

You can choose only a single one among the targets in each `recoveryTarget` configuration.

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

By default, the previous parameters are considered to be inclusive, stopping just after the recovery target, matching the behavior in PostgreSQL .

You can request exclusive behavior, stopping right before the recovery target, by setting the `exclusive` parameter to `true` . The following example shows this behavior, relying on a blob container in Azure for both base backups and the WAL archive:

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

  storage:
    size: 5Gi

  bootstrap:
    recovery:
      source: origin
      recoveryTarget:
        backupID: 20220616T142236
        targetName: "maintenance-activity"
        exclusive: true

  externalClusters:
    - name: origin
      plugin:
        name: barman-cloud.cloudnative-pg.io
        parameters:
          barmanObjectName: cluster-example-backup
          serverName: cluster-example
```

20.6 Configure the application database

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

for more information about secrets.

Important

While the *Cluster* is in recovery mode, no changes to the database, including the catalog, are permitted. This restriction includes any role overrides, which are deferred until the *Cluster* transitions to primary. During this phase, users remain as defined in the source cluster.

The following example configures the `app` database with the owner `app` and the password stored in the provided secret `app-secret` , following the bootstrap from a live cluster.

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

(continues on next page)

(continued from previous page)

```
  bootstrap:
    recovery:
      database: app
      owner: app
      secret:
        name: app-secret
      [...]
```

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

1. If the `app` database does not exist, it will be created.
2. If the `app` user does not exist, it will be created.
3. If the `app` user is not the owner of the `app` database, ownership will be granted to the `app` user.
4. If the `username` value matches the `owner` value in the secret, the password for the application user (the `app` user in this case) will be updated to the `password` value in the secret.

20.7 How recovery works under the hood

You can use the data uploaded to the object storage to *bootstrap* a new cluster from an existing backup. The operator orchestrates the recovery process using the `barman-cloud-restore` tool (for the base backup) and the `barman-cloud-wal-restore` tool (for WAL files, including parallel support, if requested).

For details and instructions on the `recovery` bootstrap method, see Bootstrap from a backup (`recovery ) <Bootstrap from a backup (*recovery* )>` .

Important

If you're not familiar with how PostgreSQL PITR

works, we suggest that you configure the recovery cluster as the original one when it comes to `.spec.postgresql.parameters` . Once the new cluster is restored, you can then change the settings as desired.

The way it works is that the operator injects an init container in the first instance of the new cluster, and the init container starts recovering the backup from the object storage.

Important

The duration of the base backup copy in the new PVC depends on the size of the backup, as well as the speed of both the network and the storage.

When the base backup recovery process is complete, the operator starts the Postgres instance in recovery mode. In this phase, PostgreSQL is up, though not able to accept connections, and the pod is healthy according to the liveness probe. By way of the `restore_command` , PostgreSQL starts fetching WAL files from the archive. You can speed up this phase by setting the `maxParallel` option and enabling the parallel WAL restore capability.

This phase terminates when PostgreSQL reaches the target, either the end of the WAL or the required target in case of PITR. You can optionally specify a `recoveryTarget` to perform a PITR. If left unspecified, the recovery continues up to the latest available WAL on the default target timeline (`latest`).

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

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

20.8 Restoring into a Cluster with a Backup Section

When restoring a cluster, the manifest may include a `plugins` section with Barman Cloud plugin pointing to a *backup* object store resource. This enables the newly created cluster to begin archiving WAL files and taking backups immediately after recovery—provided backup policies are configured.

Avoid reusing the same `ObjectStore` configuration for both *backup* and *recovery* in the same cluster. If you must, ensure that each cluster uses a unique `serverName` to prevent accidental overwrites of backup or WAL archive data.

Warning

CloudNativePG includes a safety check to prevent a cluster from overwriting existing data in a shared storage bucket. If a conflict is detected, the cluster remains in the *Setting up primary* state, and the associated pods will fail with an error. The pod logs will display: *ERROR: WAL archive check failed for server recoveredCluster: Expected empty archive* .

Important

You can bypass this safety check by setting the *cnpg.io/skipEmptyWalArchiveCheck* annotation to *enabled* on the recovered cluster. However, this is strongly discouraged unless you are highly familiar with PostgreSQL's recovery process. Skipping the check incorrectly can lead to severe data loss. Use with caution and only in expert scenarios.

CHAPTER 21

Service Management

A PostgreSQL cluster should only be accessed via standard Kubernetes network services directly managed by CloudNativePG. For more details, refer to the Service .

CloudNativePG defines three types of services for each `Cluster` resource:

- `rw` : Points to the primary instance of the cluster (read/write).
- `ro` : Points to the replicas, where available (read-only).
- `r` : Points to any PostgreSQL instance in the cluster (read).

By default, CloudNativePG creates all the above services for a `Cluster` resource, with the following conventions:

- The name of the service follows this format: `<CLUSTER_NAME>-<SERVICE_NAME>` .
- All services are of type `ClusterIP` .

Important

Default service names are reserved for CloudNativePG usage.

While this setup covers most use cases for accessing PostgreSQL within the same Kubernetes cluster, CloudNativePG offers flexibility to:

- Disable the creation of the `ro` and/or `r` default services.
- Define your own services using the standard `Service` API provided by Kubernetes.

You can mix these two options.

A common scenario arises when using CloudNativePG in database-as-a-service (DBaaS) contexts, where access to the database from outside the Kubernetes cluster is required. In such cases, you can create your own service of type `LoadBalancer` , if available in your Kubernetes environment.

21.1 Disabling Default Services

You can disable any or all of the `ro` and `r` default services through the *managed.services.disabledDefaultServices* .

Important

The *rw* service is essential and cannot be disabled because CloudNativePG relies on it to ensure PostgreSQL replication.

For example, if you want to remove both the `ro` (read-only) and `r` (read) services, you can use this configuration:

```
#  <snip>

managed:
  services:
    disabledDefaultServices: ["ro", "r"]
```

21.2 Adding Your Own Services

Important

When defining your own services, you cannot use any of the default reserved service names that follow the convention <*CLUSTER_NAME*>-<*SERVICE_NAME*> . It is your responsibility to pick a unique name for the service in the Kubernetes namespace.

You can define a list of additional services through the *managed.services.additional*

by specifying the service type (e.g., `rw`) in the `selectorType` field and optionally the `updateStrategy` .

The `serviceTemplate` field gives you access to the standard Kubernetes API for the network `Service` resource, allowing you to define both the `metadata` and the `spec` sections as you like.

You must provide a `name` to the service and avoid defining the `selector` field, as it is managed by the operator.

Warning

Service templates give you unlimited possibilities in terms of configuring network access to your PostgreSQL database. This translates into greater responsibility on your end to ensure that services work as expected. CloudNativePG has no control over the service configuration, except honoring the selector.

The `updateStrategy` field allows you to control how the operator updates a service definition. By default, the operator uses the `patch` strategy, applying changes directly to the service. Alternatively, the `replace` strategy deletes the existing service and recreates it from the template.

Warning

The *replace* strategy will cause a service disruption with every change. However, it may be necessary for modifying certain parameters that can only be set during service creation.

For example, if you want to have a single `LoadBalancer` service for your PostgreSQL database primary, you can use the following excerpt:

```
#  <snip>

managed:
  services:
    additional:
      - selectorType: rw
        serviceTemplate:
          metadata:
            name: "mydb-lb"
            labels:
              test-label: "true"
            annotations:
              test-annotation: "true"
          spec:
            type: LoadBalancer
```

The above example also shows how to set metadata such as annotations and labels for the created service.

21.2.1 About Exposing Postgres Services

There are primarily three use cases for exposing your PostgreSQL service outside your Kubernetes cluster:

- Temporarily, for testing.
- Permanently, for **DBaaS purposes**.
- Prolonged period/permanently, for **legacy applications** that cannot be easily or sustainably containerized and need to reside in a virtual machine or physical machine outside Kubernetes. This use case is very similar to DBaaS.

Be aware that allowing access to a database from the public network could expose your database to potential attacks from malicious users.

> **Warning**
>
> Ensure you secure your database before granting external access, or make sure your Kubernetes cluster is only reachable from a private network.

CHAPTER 22

PostgreSQL Configuration

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

- `postgresql.conf` : main run-time configuration file of PostgreSQL
- `pg_hba.conf` : clients authentication file
- `pg_ident.conf` : map external users to internal users

Due to the concepts of declarative configuration and immutability of the PostgreSQL containers, users are not allowed to directly touch those files. Configuration is possible through the `postgresql` section of the `Cluster` resource definition by defining custom `postgresql.conf` , `pg_hba.conf` , and `pg_ident.conf` settings via the `parameters` , the `pg_hba` , and the `pg_ident` keys.

These settings are the same across all instances.

Warning

Please don't use the *ALTER SYSTEM* query to change the configuration of the PostgreSQL instances in an imperative way. Changing some of the options that are normally controlled by the operator might indeed lead to an unpredictable/unrecoverable state of the cluster. Moreover, *ALTER SYSTEM* changes are not replicated across the cluster. See Enabling ALTER SYSTEM below for details.

A reference for custom settings usage is included in the samples, see below:

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

  # Parameters and pg_hba configuration will be append
  # to the default ones to make the cluster work
```

(continues on next page)

(continued from previous page)

```
  postgresql:
    parameters:
      max_worker_processes: "60"
    pg_hba:
      # To access through TCP/IP you will need to get username
      # and password from the secret cluster-example-custom-app
      - host all all all md5

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Require 1Gi of space per instance using default storage class
  storage:
    size: 1Gi
```

22.1 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:

```
# ...
postgresql:
  parameters:
    shared_buffers: "1GB"
# ...
```

See also

Refer to the PostgreSQL documentation for more information on the available parameters , also known as GUC (Grand Unified Configuration). Please note that CloudNativePG accepts only strings for the PostgreSQL parameters.

The content of `custom.conf` is automatically generated and maintained by the operator by applying the following sections in this order:

- Global default parameters
- Default parameters that depend on the PostgreSQL major version
- User-provided parameters
- Fixed parameters

The **global default parameters** are:

```
archive_timeout = 5min
dynamic_shared_memory_type = posix
full_page_writes = on
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
shared_preload_libraries =
ssl_max_protocol_version = TLSv1.3
ssl_min_protocol_version = TLSv1.3
wal_keep_size = 512MB
wal_level = logical
wal_log_hints = on
wal_sender_timeout = 5s
wal_receiver_timeout = 5s
```

Warning

It is your duty to plan for WAL segments retention in your PostgreSQL cluster and properly configure either *wal_keep_size* or *wal_keep_segments* , depending on the server version, based on the expected and observed workloads.

Alternatively, if the only streaming replication clients are the replica instances running in the High Availability cluster, you can take advantage of the replication slots feature, which adds support for replication slots at the cluster level. You can enable the feature with the `replicationSlots.highAvailability` option (for more information, please refer to the *About the replication user* .)

Without replication slots nor continuous backups in place, configuring `wal_keep_size` or `wal_keep_segments` is the only way to protect standbys from falling out of sync. If a standby did fall out of sync it would produce error messages like: `"could not receive data from WAL stream: ERROR: requested WAL segment ********** has already been removed"` . This will require you to dedicate a part of your `PGDATA` , or the volume dedicated to storing WAL files, to keep older WAL segments for streaming replication purposes.

The following parameters are **fixed** and exclusively controlled by the operator:

```
archive_command = /controller/manager wal-archive %p
hot_standby = true
listen_addresses = *
port = 5432
restart_after_crash = false
ssl = on
ssl_ca_file = /controller/certificates/client-ca.crt
ssl_cert_file = /controller/certificates/server.crt
ssl_key_file = /controller/certificates/server.key
unix_socket_directories = /controller/run
```

Since the fixed parameters are added at the end, they can't be overridden by the user via the YAML configuration. Those parameters are required for correct WAL archiving and replication.

22.1.1 Write-Ahead Log Level

The wal_level

parameter in PostgreSQL determines the amount of information written to the Write-Ahead Log (WAL). It accepts the following values:

- `minimal` : Writes only the information required for crash recovery.
- `replica` : Adds sufficient information to support WAL archiving and streaming replication, including the ability to run read-only queries on standby instances.
- `logical` : Includes all information from `replica` , plus additional information required for logical decoding and replication.

By default, upstream PostgreSQL sets `wal_level` to `replica` . CloudNativePG, instead, sets `wal_level` to `logical` by default to enable logical replication out of the box. This makes it easier to support use cases such as migrations from external PostgreSQL servers.

If your cluster does not require logical replication, it is recommended to set `wal_level` to `replica` to reduce WAL volume and overhead.

Finally, CloudNativePG allows `wal_level` to be set to `minimal` only for single-instance clusters with WAL archiving disabled.

22.1.2 Replication Settings

The `primary_conninfo` , `restore_command` , and `recovery_target_timeline` parameters are automatically managed by the operator based on the instance's role within the cluster. These parameters are effectively applied only when the instance is operating as a replica.

```
primary_conninfo = host=<PRIMARY> user=postgres dbname=postgres
recovery_target_timeline = latest
```

The *STANDBY_TCP_USER_TIMEOUT* , if specified, sets the `tcp_user_timeout` parameter on all standby instances managed by the operator.

The `tcp_user_timeout` parameter determines how long transmitted data can remain unacknowledged before the TCP connection is forcibly closed. Adjusting this value allows you to fine-tune the responsiveness of standby instances to network disruptions. For more details, refer to the PostgreSQL documentation .

22.1.3 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* .

22.1.4 Shared Preload Libraries

The `shared_preload_libraries` option in PostgreSQL exists to specify one or more shared libraries to be pre-loaded at server start, in the form of a comma-separated list. Typically, it is used in PostgreSQL to load those extensions that need to be available to most database sessions in the whole system (e.g. `pg_stat_statements`).

In CloudNativePG the `shared_preload_libraries` option is empty by default. Although you can override the content of `shared_preload_libraries` , we recommend that only expert Postgres users take advantage of this option.

Important

In case a specified library is not found, the server fails to start, preventing CloudNativePG from any self-healing attempt and requiring manual intervention. Please make sure you always test both the extensions and the settings of *shared_preload_libraries* if you plan to directly manage its content.

CloudNativePG is able to automatically manage the content of the `shared_preload_libraries` option for some of the most used PostgreSQL extensions (see the *Managed extensions* section below for details).

Specifically, as soon as the operator notices that a configuration parameter requires one of the managed libraries, it will automatically add the needed library. The operator will also remove the library as soon as no actual parameter requires it.

Important

Please always keep in mind that removing libraries from *shared_preload_libraries* requires a restart of all instances in the cluster in order to be effective.

You can provide additional `shared_preload_libraries` via `.spec.postgresql.shared_preload_libraries` as a list of strings: the operator will merge them with the ones that it automatically manages.

22.1.5 Managed extensions

As anticipated in the previous section, CloudNativePG automatically manages the content in `shared_preload_libraries` for some well-known and supported extensions. The current list includes:

- `auto_explain`
- `pg_stat_statements`
- `pgaudit`
- `pg_failover_slots`

Some of these libraries also require additional objects in a database before using them, normally views and/or functions managed via the `CREATE EXTENSION` command to be run in a database (the `DROP EXTENSION` command typically removes those objects).

For such libraries, CloudNativePG automatically handles the creation and removal of the extension in all databases that accept a connection in the cluster, identified by the following query:

```
SELECT datname FROM pg_database WHERE datallowconn
```

Note

The above query also includes template databases like *template1* .

Important

With the introduction of *declarative extensions*

in the `Database` CRD, you can now manage extensions directly. As a result, the managed extensions feature may undergo significant changes in future versions of CloudNativePG, and some functionalities might be deprecated.

22.1.6 Enabling `auto_explain`

The auto_explain

extension provides a means for logging execution plans of slow statements automatically, without having to manually run `EXPLAIN` (helpful for tracking down un-optimized queries).

You can enable `auto_explain` by adding to the configuration a parameter that starts with `auto_explain.` as in the following example excerpt (which automatically logs execution plans of queries that take longer than 10 seconds to complete):

```
# ...
postgresql:
  parameters:
    auto_explain.log_min_duration: "10s"
# ...
```

Note

Enabling auto_explain can lead to performance issues. Please refer to the auto explain documentation

22.1.7 Enabling `pg_stat_statements`

The pg_stat_statements

extension is one of the most important capabilities available in PostgreSQL for real-time monitoring of queries.

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

```
# ...
postgresql:
  parameters:
    pg_stat_statements.max: "10000"
    pg_stat_statements.track: all
# ...
```

As explained previously, the operator will automatically add `pg_stat_statements` to `shared_preload_libraries` and run `CREATE EXTENSION IF NOT EXISTS pg_stat_statements` on each database, enabling you to run queries against the `pg_stat_statements` view.

22.1.8 Enabling `pgaudit`

The `pgaudit` extension provides detailed session and/or object audit logging via the standard PostgreSQL logging facility.

CloudNativePG has transparent and native support for PGAudit on PostgreSQL clusters. For further information, please refer to the *PGAudit Logs*

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

```
#
postgresql:

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

#


####  Enabling `pg_failover_slots`

The [`pg_failover_slots`](https://github.com/EnterpriseDB/pg_failover_slots)
extension by EDB ensures that logical replication slots can survive a
failover scenario. Failovers are normally implemented using physical
streaming replication, like in the case of CloudNativePG.

You can enable `pg_failover_slots` by adding to the configuration a parameter
that starts with `pg_failover_slots.`: as explained above, the operator will
transparently manage the `pg_failover_slots` entry in the
`shared_preload_libraries` option depending on this.

Please refer to [`the `pg_failover_slots` documentation`](https://www.enterprisedb.
↪com/docs/pg_extensions/pg_failover_slots)
for details on this extension.

Additionally, for each database that you intend to you use with `pg_failover_slots`
you need to add an entry in the `pg_hba` section that enables each replica to
connect to the primary.
For example, suppose that you want to use the `app` database with `pg_failover_slots`,
you need to add this entry in the `pg_hba` section:

```  yaml postgresql: pg_hba: - hostssl app streaming_replica all cert
```

22.2 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.

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

Since the first matching rule is used for authentication, the `pg_hba.conf` file generated by the operator can be seen as composed of four sections:

1. Fixed rules
2. User-defined rules
3. Optional LDAP section
4. Default rules

Fixed rules:
```

``` text local all all peer

hostssl postgres streaming_replica all cert map=cnpg_streaming_replica hostssl replication streaming_replica all cert map=cnpg_streaming_replica hostssl all cnpg_pooler_pgbouncer all cert map=cnpg_pooler_pgbouncer

```
Default rules:

``` text 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:

``` text local all all peer

hostssl postgres streaming_replica all cert map=cnpg_streaming_replica hostssl replication streaming_replica all cert map=cnpg_streaming_replica hostssl all cnpg_pooler_pgbouncer all cert map=cnpg_pooler_pgbouncer

host all all all scram-sha-256 # (or md5 for PostgreSQL version <= 13)

edb_notranlate_13 yaml 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:

``` yaml postgresql: ldap: server: 'openldap.default.svc.cluster.local'␣
↪bindSearchAuth: baseDN: 'ou=org,dc=example,dc=com' bindDN: 'cn=admin,dc=example,
↪dc=com' bindPassword: name: 'ldapBindPassword' key: 'data' searchAttribute: 'uid'
```

22.3 The `pg_ident` section

`pg_ident` is a list of PostgreSQL User Name Maps that CloudNativePG uses to generate and maintain the ident map file (known as `pg_ident.conf`) inside the data directory.

!!! Important See the PostgreSQL documentation for *more information on ``pg_ident.conf`* <https://www.postgresql.org/docs/current/auth-username-maps.html>`__.

The `pg_ident.conf` file written by the operator is made up of the following two sections:

1. Fixed rules
2. User-defined rules

Currently the only fixed rule, automatically generated by the operator, is:

``` text local postgres

```
The instance manager detects the user running the PostgreSQL instance and
automatically adds a rule to map it to the `postgres` user in the database.

If the `postgres` user is not properly configured inside the container, the
instance manager will allow any local user to connect and then log a warning
message like the following:

``` text Unable to identify the current user. Falling back to insecure mapping.
```

The resulting `pg_ident.conf` will look like this:

``` text local postgres

edb_notranlate_18 yaml postgresql: pg_ident: - “mymap /^(.*)@mydomain\.com$ \1”

```
Changing configuration

You can apply configuration changes by editing the `postgresql` section of
the `Cluster` resource.

After the change, the cluster instances will immediately reload the
configuration to apply the changes.
If the change involves a parameter requiring a restart, the operator will
perform a rolling upgrade.

Enabling `ALTER SYSTEM`

CloudNativePG strongly advocates employing the Cluster manifest as the
exclusive method for altering the configuration of a PostgreSQL cluster. This
approach guarantees coherence across the entire high-availability cluster and
aligns with best practices for Infrastructure-as-Code.

In CloudNativePG the default configuration disables the use of `ALTER SYSTEM`
on new Postgres clusters. This decision is rooted in the recognition of
potential risks associated with this command. To enable the use of `ALTER SYSTEM`,
you can explicitly set `.spec.postgresql.enableAlterSystem` to `true`.

!!! Warning
    Proceed with caution when utilizing `ALTER SYSTEM`. This command operates
```

(continues on next page)
```

(continued from previous page)

```
    directly on the connected instance and does not undergo replication.
    CloudNativePG assumes responsibility for certain fixed parameters and complete
    control over others, emphasizing the need for careful consideration.

Starting from PostgreSQL 17, the `.spec.postgresql.enableAlterSystem` setting
directly controls the [`allow_alter_system` GUC in PostgreSQL](https://www.postgresql.
↪org/docs/17/runtime-config-compatible.html#GUC-ALLOW-ALTER-SYSTEM)
— a feature directly contributed by CloudNativePG to PostgreSQL.

Prior to PostgreSQL 17, when `.spec.postgresql.enableAlterSystem` is set to
`false`, the `postgresql.auto.conf` file is made read-only. Consequently, any
attempt to execute the `ALTER SYSTEM` command will result in an error. The
error message might look like this:

```  output ERROR:  could not open file "postgresql.auto.conf": Permission denied
```

22.4 Dynamic Shared Memory settings

PostgreSQL supports a few implementations for dynamic shared memory management through the `dynamic_shared_memory_type <https://www.postgresql.org/docs/current/runtime-config-resource.html#GUC-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>`__.

22.4.1 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:

``` sh mount | grep shm

```
You should get something similar to the following output:

``` console shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=******)

edb_notranlate_22 yaml spec: ephemeralVolumesSizeLimit: shm: 1Gi
```

22.4.2 System V shared memory

In case your Kubernetes cluster has a high enough value for the `SHMMAX` and `SHMALL` parameters, you can also set:

``` yaml dynamic_shared_memory_type: “sysv”

```
You can check the `SHMMAX`/`SHMALL` from inside a PostgreSQL container, by running:
```

(continues on next page)

(continued from previous page)

```
``` sh ipcs -lm

edb_notranlate_25 console ------ 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:

`sh cat /proc/sys/kernel/shmall cat /proc/sys/kernel/shmmax`

22.5 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_alter_system`
- `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`
- `hba_file`
- `hot_standby`
- `ident_file`
- `jit_provider`
- `listen_addresses`
- `log_destination`
- `log_directory`
- `log_file_mode`
- `log_filename`
- `log_rotation_age`
```

- log_rotation_size
- log_truncate_on_rotation
- logging_collector
- port
- primary_conninfo
- primary_slot_name
- promote_trigger_file
- recovery_end_command
- recovery_min_apply_delay
- recovery_target
- recovery_target_action
- recovery_target_inclusive
- recovery_target_lsn
- recovery_target_name
- recovery_target_time
- recovery_target_timeline
- recovery_target_xid
- restart_after_crash
- restore_command
- shared_preload_libraries
- ssl
- ssl_ca_file
- ssl_cert_file
- ssl_crl_file
- ssl_dh_params_file
- ssl_ecdh_curve
- ssl_key_file
- ssl_passphrase_command
- ssl_passphrase_command_supports_reload
- ssl_prefer_server_ciphers
- stats_temp_directory
- synchronous_standby_names
- syslog_facility
- syslog_ident
- syslog_sequence_numbers
- syslog_split_messages

- unix_socket_directories
- unix_socket_group
- unix_socket_permissions

CHAPTER 23

PostgreSQL Role Management

From its inception, CloudNativePG has managed the creation of specific roles required in PostgreSQL instances:

- some reserved users, such as the `postgres` superuser, `streaming_replica` and `cnpg_pooler_pgbouncer` (when the PgBouncer `Pooler` is used)
- The application user, set as the low-privilege owner of the application database

This process is described in the *Bootstrap* section.

With the `managed` stanza in the cluster spec, CloudNativePG now provides full lifecycle management for roles specified in `.spec.managed.roles`.

This feature enables declarative management of existing roles, as well as the creation of new roles if they are not already present in the database. Role creation will occur *after* the database bootstrapping is complete.

An example manifest for a cluster with declarative role management can be found in the file.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-roles
spec:
  instances: 3
  storage:
    size: 1Gi

  managed:
    roles:
    - name: app
      createdb: true
      login: true
    - name: dante
      ensure: present
      comment: my database-side comment
      login: true
```

(continues on next page)

(continued from previous page)

```
      superuser: false
      createdb: true
      createrole: false
      inherit: false
      replication: false
      bypassrls: false
      connectionLimit: 4
      validUntil: "2053-04-12T15:04:05Z"
      inRoles:
        - pg_monitor
        - pg_signal_backend
      passwordSecret:
        name: cluster-example-dante
- --
apiVersion: v1
data:
  username: ZGFudGU=
  password: ZGFudGU=
kind: Secret
metadata:
  name: cluster-example-dante
type: kubernetes.io/basic-auth
```

Here is an excerpt from that file:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
spec:
  managed:
    roles:
    - name: dante
      ensure: present
      comment: Dante Alighieri
      login: true
      superuser: false
      inRoles:
        - pg_monitor
        - pg_signal_backend
```

The role specification in `.spec.managed.roles` adheres to the PostgreSQL structure and naming conventions . Please refer to the *API Reference* for the full list of attributes you can define for each role.

A few points are worth noting:

1. The `ensure` attribute is **not** part of PostgreSQL. It enables declarative role management to create and remove roles. The two possible values are `present` (the default) and `absent` .
2. The `inherit` attribute is true by default, following PostgreSQL conventions.
3. The `connectionLimit` attribute defaults to -1, in line with PostgreSQL conventions.
4. Role membership with `inRoles` defaults to no memberships.

Declarative role management ensures that PostgreSQL instances align with the spec. If a user modifies role attributes directly in the database, the CloudNativePG operator will revert those changes during the next reconciliation cycle.

23.1 Password management

The declarative role management feature includes reconciling of role passwords. Passwords are managed in fundamentally different ways in the Kubernetes world and in PostgreSQL, and as a result there are a few things to note.

Managed role configurations may optionally specify the name of a **Secret** where the username and password are stored (encoded in Base64 as is usual in Kubernetes). For example:

```
managed:
  roles:
  - name: dante
    ensure: present
    [… snipped …]
    passwordSecret:
      name: cluster-example-dante
```

This would assume the existence of a Secret called `cluster-example-dante`, containing a username and password. The username should match the role we are setting the password for. For example, :

```
apiVersion: v1
data:
  username: ZGFudGU=
  password: ZGFudGU=
kind: Secret
metadata:
  name: cluster-example-dante
  labels:
    cnpg.io/reload: "true"
type: kubernetes.io/basic-auth
```

If there is no `passwordSecret` specified for a role, the instance manager will not try to CREATE / ALTER the role with a password. This keeps with PostgreSQL conventions, where ALTER will not update passwords unless directed to with `WITH PASSWORD`.

If a role was initially created with a password, and we would like to set the password to NULL in PostgreSQL, this necessitates being explicit on the part of the user of CloudNativePG. To distinguish "no password provided in spec" from "set the password to NULL", the field `DisablePassword` should be used.

Imagine we decided we would like to have no password on the `dante` role in the database. In such case we would specify the following:

```
managed:
  roles:
  - name: dante
    ensure: present
    [… snipped …]
    disablePassword: true
```

NOTE: it is considered an error to set both `passwordSecret` and `disablePassword` on a given role. This configuration will be rejected by the validation webhook.

23.1.1 Password expiry, VALID UNTIL

The `VALID UNTIL` role attribute in PostgreSQL controls password expiry. Roles created without `VALID UNTIL` specified get NULL by default in PostgreSQL, meaning that their password will never expire.

PostgreSQL uses a timestamp type for `VALID UNTIL` , which includes support for the value `'infinity'` indicating that the password never expires. Please see the PostgreSQL documentation

for reference.

With declarative role management, the `validUntil` attribute for managed roles controls password expiry. `validUntil` can only take:

- a Kubernetes timestamp, or
- be omitted (defaulting to `null`)

In the first case, the given `validUntil` timestamp will be set in the database as the `VALID UNTIL` attribute of the role.

In the second case (omitted `validUntil`) the operator will ensure password never expires, mirroring the behavior of PostgreSQL. Specifically:

- in case of new role, it will omit the `VALID UNTIL` clause in the role creation statement
- in case of existing role, it will set `VALID UNTIL` to `infinity` if `VALID UNTIL` was not set to `NULL` in the database (this is due to PostgreSQL not allowing `VALID UNTIL NULL` in the `ALTER ROLE` SQL statement)

> **Warning**
>
> New roles created without *passwordSecret* will have a *NULL* password inside PostgreSQL.

23.1.2 Password hashed

You can also provide pre-encrypted passwords by specifying the password in MD5/SCRAM-SHA-256 hash format:

```
kind: Secret
type: kubernetes.io/basic-auth
metadata:
  name: cluster-example-cavalcanti
  labels:
    cnpg.io/reload: "true"
apiVersion: v1
stringData:
  username: cavalcanti
  password: SCRAM-SHA-256$<iteration count>:<salt>$<StoredKey>:<ServerKey>
```

23.2 Unrealizable role configurations

In PostgreSQL, in some cases, commands cannot be honored by the database and will be rejected. Please refer to the PostgreSQL documentation on error codes

for details.

Role operations can produce such fundamental errors. Two examples:

- We ask PostgreSQL to create the role `petrarca` as a member of the role (group) `poets` , but `poets` does not exist.

- We ask PostgreSQL to drop the role `dante` , but the role `dante` is the owner of the database `inferno` .

These fundamental errors cannot be fixed by the database, nor the CloudNativePG operator, without clarification from the human administrator. The two examples above could be fixed by creating the role `poets` or dropping the database `inferno` respectively, but they might have originated due to human error, and in such case, the "fix" proposed might be the wrong thing to do.

CloudNativePG will record when such fundamental errors occur, and will display them in the cluster Status. Which segues into…

23.3 Status of managed roles

The Cluster status includes a section for the managed roles' status, as shown below:

```
status:
  […snipped…]
  managedRolesStatus:
    byStatus:
      not-managed:
      - app
      pending-reconciliation:
      - dante
      - petrarca
      reconciled:
      - ariosto
      reserved:
      - postgres
      - streaming_replica
    cannotReconcile:
      dante:
      - could not perform DELETE on role dante: owner of database inferno
      petrarca:
      - could not perform UPDATE_MEMBERSHIPS on role petrarca: role "poets" does not␣
↪exist
```

Note the special sub-section `cannotReconcile` for operations the database (and CloudNativePG) cannot honor, and which require human intervention.

This section covers roles reserved for operator use and those that are **not** under declarative management, providing a comprehensive view of the roles in the database instances.

The *Kubectl Plugin* also shows the status of managed roles in its `status` sub-command:

```
Managed roles status
Status                  Roles
- -----                   -----
pending-reconciliation  petrarca
reconciled              app,dante
reserved                postgres,streaming_replica

Irreconcilable roles
Role      Errors
- ---       ------
petrarca  could not perform UPDATE_MEMBERSHIPS on role petrarca: role "poets" does␣
↪not exist
```

Important

In terms of backward compatibility, declarative role management is designed to ignore roles that exist in the database but are not included in the spec. The lifecycle of these roles will continue to be managed within PostgreSQL, allowing CloudNativePG users to adopt this feature at their convenience.

CHAPTER 24

PostgreSQL Database Management

CloudNativePG simplifies PostgreSQL database provisioning by automatically creating an application database named `app` by default. This default behavior is explained in the Bootstrap an empty cluster (`initdb ) <Bootstrap an empty cluster (*initdb* )>`

section.

For more advanced use cases, CloudNativePG introduces **declarative database management**, which empowers users to define and control the lifecycle of PostgreSQL databases using the `Database` Custom Resource Definition (CRD). This method seamlessly integrates with Kubernetes, providing a scalable, automated, and consistent approach to managing PostgreSQL databases.

24.1 Key Concepts

24.1.1 Scope of Management

Important

CloudNativePG manages **global objects** in PostgreSQL clusters, including databases, roles, and tablespaces. However, it does **not** manage database content beyond extensions and schemas (e.g., tables). To manage database content, use specialized tools or rely on the applications themselves.

24.1.2 Declarative `Database` Manifest

The following example demonstrates how a `Database` resource interacts with a `Cluster` :

```
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: cluster-example-one
```

(continues on next page)

(continued from previous page)

```
spec:
  name: one
  owner: app
  cluster:
    name: cluster-example
  extensions:
  - name: bloom
    ensure: present
```

When applied, this manifest creates a `Database` object called `cluster-example-one` requesting a database named `one` , owned by the `app` role, in the `cluster-example` PostgreSQL cluster.

Note

Please refer to the *API Reference*

the full list of attributes you can define for each `Database` object.

24.1.3 Required Fields in the `Database` Manifest

- `metadata.name` : Unique name of the Kubernetes object within its namespace.
- `spec.name` : Name of the database as it will appear in PostgreSQL.
- `spec.owner` : PostgreSQL role that owns the database.
- `spec.cluster.name` : Name of the target PostgreSQL cluster.

The `Database` object must reference a specific `Cluster` , determining where the database will be created. It is managed by the cluster's primary instance, ensuring the database is created or updated as needed.

Note

The distinction between *metadata.name* and *spec.name* allows multiple *Database* resources to reference databases with the same name across different CloudNativePG clusters in the same Kubernetes namespace.

24.2 Reserved Database Names

PostgreSQL automatically creates databases such as `postgres` , `template0` , and `template1` . These names are reserved and cannot be used for new `Database` objects in CloudNativePG.

Important

Creating a *Database* with *spec.name* set to *postgres* , *template0* , or *template1* is not allowed.

24.3 Reconciliation and Status

Once a `Database` object is reconciled successfully:

- `status.applied` will be set to `true` .

- `status.observedGeneration` will match the `metadata.generation` of the last applied configuration.

Example of a reconciled `Database` object:

```
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  generation: 1
  name: cluster-example-one
spec:
  cluster:
    name: cluster-example
  name: one
  owner: app
status:
  observedGeneration: 1
  applied: true
```

If an error occurs during reconciliation, `status.applied` will be `false` , and an error message will be included in the `status.message` field.

24.4 Deleting a Database

CloudNativePG supports two methods for database deletion:

1. Using the `delete` reclaim policy
2. Declaratively setting the database's `ensure` field to `absent`

24.4.1 Deleting via `delete` Reclaim Policy

The `databaseReclaimPolicy` field determines the behavior when a `Database` object is deleted:

- `retain` (default): The database remains in PostgreSQL for manual management.
- `delete` : The database is automatically removed from PostgreSQL.

Example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: cluster-example-two
spec:
  databaseReclaimPolicy: delete
  name: two
  owner: app
  cluster:
    name: cluster-example
```

Deleting this `Database` object will automatically remove the `two` database from the `cluster-example` cluster.

24.4.2 Declaratively Setting `ensure: absent`

To remove a database, set the `ensure` field to `absent` like in the following example:.

```
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: cluster-example-database-to-drop
spec:
  cluster:
    name: cluster-example
  name: database-to-drop
  owner: app
  ensure: absent
```

This manifest ensures that the `database-to-drop` database is removed from the `cluster-example` cluster.

24.5 Managing Extensions in a Database

> **Note**
>
> While extensions are database-scoped rather than global objects, CloudNativePG provides a declarative interface for managing them. This approach is necessary because installing certain extensions may require superuser privileges, which CloudNativePG recommends disabling by default. By leveraging this API, users can efficiently manage extensions in a scalable and controlled manner without requiring elevated privileges.

CloudNativePG simplifies and automates the management of PostgreSQL extensions within the target database.

To enable this feature, define the `spec.extensions` field with a list of extension specifications, as shown in the following example:

```
#  ...

spec:
  extensions:
  - name: bloom
    ensure: present

#  ...
```

Each extension entry supports the following properties:

- `name` *(mandatory)*: The name of the extension.
- `ensure` : Specifies whether the extension should be present or absent in the database:
  - `present` : Ensures that the extension is installed (default).
  - `absent` : Ensures that the extension is removed.
- `version` : The specific version of the extension to install or upgrade to.
- `schema` : The schema in which the extension should be installed.

> **Note**
>
> CloudNativePG manages extensions using the following PostgreSQL's SQL commands: CREATE EXTENSION , DROP EXTENSION , ALTER EXTENSION

(limited to `UPDATE TO` and `SET SCHEMA`).

The operator reconciles only the extensions explicitly listed in `spec.extensions` . Any existing extensions not specified in this list remain unchanged.

> **Warning**
>
> Before the introduction of declarative extension management, CloudNativePG did not offer a straightforward way to create extensions through configuration. To address this, the *Managed extensions*

feature was introduced, enabling the automated and transparent management of key extensions like `pg_stat_statements` . Currently, it is your responsibility to ensure there are no conflicts between extension support in the `Database` CRD and the managed extensions feature.

24.6 Managing Schemas in a Database

> **Note**
>
> Schema management in PostgreSQL is an exception to CloudNativePG's primary focus on managing global objects. Since schemas exist within a database, they are typically managed as part of the application development process. However, CloudNativePG provides a declarative interface for schema management, primarily to complete the support of extensions deployment within schemas.

CloudNativePG simplifies and automates the management of PostgreSQL schemas within the target database.

To enable this feature, define the `spec.schemas` field with a list of schema specifications, as shown in the following example:

```
#  ...

spec:
  schemas:
  - name: app
    owner: app

#  ...
```

Each schema entry supports the following properties:

- `name` *(mandatory)*: The name of the schema.
- `owner` : The owner of the schema.
- `ensure` : Specifies whether the schema should be present or absent in the database:
  - `present` : Ensures that the schema is installed (default).
  - `absent` : Ensures that the schema is removed.

> **Note**
>
> CloudNativePG manages schemas using the following PostgreSQL's SQL commands: CREATE SCHEMA , DROP SCHEMA , ALTER SCHEMA .

24.7 Managing Foreign Data Wrappers (FDWs) In a Database

> **Note**
>
> Foreign Data Wrappers (FDWs) are database-scoped objects that typically require superuser privileges to create or modify. CloudNativePG provides a declarative API for managing FDWs, enabling users to define and maintain them in a controlled, Kubernetes-native way without directly executing SQL commands or escalating privileges.

CloudNativePG enables seamless and automated management of PostgreSQL foreign data wrappers in the target database using declarative configuration.

To enable this feature, define the `spec.fdws` field with a list of FDW specifications, as shown in the following example:

```
#  ...

spec:
  fdws:
  - name: postgres_fdw
    usage:
      - name: app
      - type: grant

#  ...
```

Each FDW entry supports the following properties:

- `name` : The name of the foreign data wrapper **(mandatory)**.
- `ensure` : Indicates whether the FDW should be `present` or `absent` in the database (default is `present`).
- `handler` : The name of the handler function used by the FDW. If not specified, the default handler defined by the FDW extension (if any) will be used.
- `validator` : The name of the validator function used by the FDW. If not specified, the default validator defined by the FDW extension (if any) will be used.
- `owner` : The owner of the FDW **(must be a superuser)**.
- `usage` : The list of `USAGE` permissions of the FDW, with the following fields:
  - `name` : The name of the role to which the usage permission should be granted or from which it should be revoked.
  - `type` : The type of the usage permission. Supports `grant` and `revoke` .
- `options` : A map of FDW-specific options to manage, where each key is the name of an option. Each option supports the following fields:
  - `value` : The string value of the option.
  - `ensure` : Indicates whether the option should be `present` or `absent` .

> **Note**
>
> Both *handler* and *validator* are optional, and if not specified, the default handler and validator defined by the FDW extension (if any) will be used. Setting *handler* or *validator* to *"-"* will remove the handler or validator from the FDW respectively. This follows the PostgreSQL convention, where "-" denotes the absence of a handler or validator.

> **Warning**
>
> PostgreSQL restricts ownership of foreign data wrappers to **roles with superuser privileges only**. Attempting to assign ownership to a non-superuser (e.g., an app role) will be ignored or rejected, as PostgreSQL does not allow non-superuser ownership of foreign data wrappers.

The operator reconciles only the FDWs explicitly listed in `spec.fdws`. Any existing FDWs not declared in this list are left untouched.

> **Note**
>
> CloudNativePG manages FDWs using PostgreSQL's native SQL commands: CREATE FOREIGN DATA WRAPPER , ALTER FOREIGN DATA WRAPPER , and DROP FOREIGN DATA WRAPPER . The *ALTER* command supports option updates.

24.8 Limitations and Caveats

24.8.1 Renaming a database

While CloudNativePG adheres to PostgreSQL's CREATE DATABASE and ALTER DATABASE

commands, **renaming databases is not supported**. Attempting to modify `spec.name` in an existing `Database` object will result in rejection by Kubernetes.

24.8.2 Creating vs. Altering a Database

- For new databases, CloudNativePG uses the `CREATE DATABASE` statement.
- For existing databases, `ALTER DATABASE` is used to apply changes.

It is important to note that there are some differences between these two Postgres commands: in particular, the options accepted by `ALTER` are a subset of those accepted by `CREATE` .

> **Warning**
>
> Some fields, such as encoding and collation settings, are immutable in PostgreSQL. Attempts to modify these fields on existing databases will be ignored.

24.8.3 Replica Clusters

Database objects declared on replica clusters cannot be enforced, as replicas lack write privileges. These objects will remain in a pending state until the replica is promoted.

24.8.4 Conflict Resolution

If two `Database` objects in the same namespace manage the same PostgreSQL database (i.e., identical `spec.name` and `spec.cluster.name`), the second object will be rejected.

Example status message:

```
status:
  applied: false
  message: reconciliation error: database "one" is already managed by Database object
↪"cluster-example-one"
```

24.8.5 Postgres Version Differences

CloudNativePG adheres to PostgreSQL's capabilities. For example, features like `ICU_RULES` introduced in PostgreSQL 16 are unavailable in earlier versions. Errors from PostgreSQL will be reflected in the `Database` object's `status` .

24.8.6 Manual Changes

CloudNativePG does not overwrite manual changes to databases. Once reconciled, a `Database` object will not be reapplied unless its `metadata.generation` changes, giving flexibility for direct PostgreSQL modifications.

CHAPTER 25

Tablespaces

A tablespace is a robust and widely embraced feature in database management systems. It offers a powerful means to enhance the vertical scalability of a database by decoupling the physical and logical modeling of data. Essentially, it serves as a technique for physical database modeling, enabling the efficient distribution of I/O operations across multiple volumes on distinct storage. It thereby optimizes performance through parallel on-disk read/write operations.

In the context of the database industry, tablespaces play a strategic role, particularly when paired with table partitioning, a logical database modeling technique. They prove instrumental in managing large-scale databases and are also used for tasks such as separating tables from indexes or executing temporary operations.

Tablespaces in PostgreSQL have been playing a pivotal role since 2005 (version 8.0), while declarative partitioning was introduced in 2017 (version 10). Consequently, tablespaces are seamlessly integrated into all supported releases of PostgreSQL. Quoting from the *PostgreSQL documentation on tablespaces* :

> By using tablespaces, an administrator can control the disk layout of a > PostgreSQL installation. This is useful in at least two ways. > > - First, if the partition or volume on which the cluster was initialized runs > out of space and cannot be extended, a tablespace can be created on a > different partition and used until the system can be reconfigured. > - Second, tablespaces allow an administrator to use knowledge of the usage > pattern of database objects to optimize performance.

25.1 Declarative tablespaces

CloudNativePG provides support for PostgreSQL tablespaces through *declarative tablespaces*, operating at two distinct levels:

- Kubernetes, managing persistent volume claims, identically to how PGDATA and WAL volumes are handled
- PostgreSQL, managing the `TABLESPACE` global objects in the PostgreSQL instance

Being a part of the Kubernetes ecosystem, CloudNativePG's declarative tablespaces are implemented by leveraging persistent volume claims (and persistent volumes). Each tablespace defined in the cluster is housed in its own persistent volume. CloudNativePG takes care of generating the PVCs. It mounts the required volumes in the instance pods in normalized locations and ensures replicas are ready to support tablespaces before activating them in the primary.

You can set up tablespaces when creating the cluster or add them later, provided the storage is available when requested. Currently, you can't remove them. However, this limitation will be addressed in a future minor or patch version of

CloudNativePG.

25.2 Using declarative tablespaces

Using declarative tablespaces is straightforward. You can find a full example in

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

  storage:
    size: 1Gi

  tablespaces:
    - name: atablespace
      storage:
        size: 1Gi
        storageClass: standard
      temporary: true
    - name: another_tablespace
      storage:
        size: 2Gi
        storageClass: standard
      temporary: true
    - name: tablespacea1
      storage:
        size: 2Gi
        storageClass: standard
```

.

To use them, use the new `tablespaces` stanza on a new or existing `Cluster` resource:

```
spec:
  instances: 3

  # ...

  tablespaces:
    - name: tbs1
      storage:
        size: 1Gi
    - name: tbs2
      storage:
        size: 2Gi
    - name: tbs3
      storage:
        size: 2Gi
```

Each tablespace has its own storage section where you can configure the size and the storage class of the generated PVC. The administrator can thus plan to use different storage classes for different kinds of workloads, as explained in

Storage classes and tablespaces .

CloudNativePG creates the persistent volume claims for each instance in the high-availability Postgres cluster. It mounts them in each pod when they have been provisioned. Then, it ensures that the `tbs1` , `tbs2` , and `tbs3` tablespaces are created on the primary PostgreSQL instance using the `CREATE TABLESPACE` command. This process is quick, and you see this reflected in Postgres:

```
app=# SELECT oid, spcname FROM pg_tablespace;
  oid  |      spcname
- ------+--------------------
  1663 | pg_default
  1664 | pg_global
 16387 | tbs1
 16388 | tbs2
 16389 | tbs3
(5 rows)
```

You can start using them right away:

```
app=# CREATE TABLE fibonacci(num INTEGER) TABLESPACE tbs1;
CREATE TABLE
```

The cluster status has a section for tablespaces:

```
status:

  <- snipped ->
  tablespacesStatus:
  - name: atablespace
    state: reconciled
  - name: another_tablespace
    state: reconciled
  - name: tablespacea1
    state: reconciled
```

25.3 Storage classes and tablespaces

You can use different storage classes for your tablespaces, just as you can for PGDATA and WAL volumes. This is a convenient way of optimizing your resources, balancing performance and costs of your storage based on data access usage and expectations.

This example helps to explain the feature:

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

  storage:
    size: 10Gi
  walStorage:
    size: 10Gi
```

(continues on next page)

(continued from previous page)

```
  tablespaces:
    - name: current
      storage:
        size: 100Gi
        storageClass: fastest
    - name: this_year
      storage:
        size: 500Gi
        storageClass: balanced
```

The `yardbirds` cluster example requests 4 persistent volume claims using 3 different storage classes:

- Default storage class – Used by the `PGDATA` and WAL volumes.
- `fastest` – Used by the `current` tablespace to store the most active and demanding set of data in the database.
- `balanced` – Used by the `this_year` tablespace to store older partitions of data that are rarely accessed by users and where performance expectations aren't the highest.

You can then take advantage of horizontal table partitioning and create the current month's table (for example, facts for December 2023) in the `current` tablespace:

```
CREATE TABLE facts_202312 PARTITION OF facts
    FOR VALUES FROM (2023-12-01) TO (2024-01-01)
    TABLESPACE current;
```

Important

This example assumes you're familiar with PostgreSQL declarative partitioning .

25.4 Tablespace ownership

By default, unless otherwise specified, tablespaces are owned by the `app` application user, as defined in `.spec.bootstrap.initdb.owner` . See *Bootstrap a new cluster* for details. This default behavior works in most microservice database use cases.

You can set the owner of a tablespace in the `owner` stanza, for example the `postgres` user, like in the following excerpt:

```
# ...
tablespaces:
  - name: clapton
    owner:
      name: postgres
    storage:
      size: 1Gi
```

Important

If you change the ownership of a tablespace, make sure that you're using an existing role. Otherwise, the status of the cluster reports the issue and stops reconciling tablespaces until fixed. It's your responsibility to monitor the status and the log and to promptly intervene by fixing the issue.

If you define a tablespace with an owner that doesn't exist, CloudNativePG can't create the tablespace and reflects this in the cluster status:

```
spec:
  instances: 3

  # ...

  tablespaces:
    - name: tbs1
      storage:
        size: 1Gi
    - name: tbs2
      storage:
        size: 2Gi
    - name: tbs3
      owner:
        name: badhombre
      storage:
        size: 2Gi
        status:

  <- snipped ->
  tablespacesStatus:
  - name: tbs1
    status: reconciled
  - name: tbs2
    status: reconciled
  - error: while creating tablespace tbs3: ERROR: role "badhombre" does
      not exist (SQLSTATE 42704)
    name: tbs3
    status: pending
```

25.5 Backup and recovery

CloudNativePG handles backup of tablespaces (and the relative tablespace map) both on object stores and volume snapshots.

Warning

By default, backups are taken from replica nodes. A backup taken immediately after creating tablespaces in a cluster can result in an incomplete view of the tablespaces from the replica and thus an incomplete backup. The lag will be resolved in a maximum of 5 minutes, with the next reconciliation.

Warning

When you add or remove a tablespace in an existing cluster, recovery from WAL will fail until you take a new base backup.

Once a cluster with tablespaces has a base backup, you can restore a new cluster from it. When it comes to the recovery

side, it's your responsibility to ensure that the `Cluster` definition of the recovered database contains the exact list of tablespaces.

25.6 Replica clusters

Replica clusters must have the same tablespace definition as their origin. The reason is that tablespace management commands like `CREATE TABLESPACE` are WAL logged and are replayed by any physical replication client (streaming or by way of WAL shipping).

It's your responsibility to ensure that replica clusters have the same list of tablespaces, with the same name. Storage class and size might vary.

For example:

```
spec:

  # ...
  bootstrap:
    recovery:
      # ... your selected recovery method

  tablespaces:
    - name: tbs1
      storage:
        size: 1Gi
    - name: tbs2
      storage:
        size: 2Gi
    - name: tbs3
      storage:
        size: 2Gi
```

25.7 Temporary tablespaces

PostgreSQL allows you to define one or more temporary tablespaces to create temporary objects (temporary tables and indexes on temporary tables) when a `CREATE` command doesn't explicitly specify a tablespace, and to create temporary files for purposes such as sorting large data sets. When no temporary tablespace is specified, PostgreSQL uses the default tablespace of a database, which is currently the main `PGDATA` volume.

When you specify more than one temporary tablespace, PostgreSQL randomly picks one the first time a temporary object needs to be created in a transaction. Then it sequentially iterates through the list.

Temporary tablespaces also work like regular tablespaces with regard to backups.

CloudNativePG provides the `.spec.tablespaces[*].temporary` option to determine whether to add a tablespace to the `temp_tablespaces` PostgreSQL parameter and thus become eligible to store temporary data that doesn't have an explicit tablespace assignment.

```
spec:
  [...]
  tablespaces:
    - name: atablespace
      storage:
```

(continues on next page)

(continued from previous page)

```
      size: 1Gi
    temporary: true
```

They can be created at initialization time or added later, requiring a rolling update. The `temporary: true/false` option adds or removes the tablespace name to or from the list of tablespaces in the `temp_tablespaces` option. This change doesn't require a restart of PostgreSQL.

Although temporary tablespaces can also work as regular tablespaces (meaning that users can also host regular data on them while using them for temporary operations), we recommend that you don't mix the two workloads.

See the *PostgreSQL documentation on `temp_tablespaces* <https://www.postgresql.org/docs/current/runtime-config-client.html#GUC-TEMP-TABLESPACES>`_

for details.

25.8 kubectl plugin support

The *kubectl status* plugin includes a section dedicated to tablespaces that offers a convenient overview, including tablespace status, owner, temporary flag, and any errors:

```
[...]

Tablespaces status
Tablespace          Owner  Status      Temporary  Error
- ---------          -----  ------      ---------  -----
atablespace         app    reconciled  true
another_tablespace  app    reconciled  true
tablespacea1        app    reconciled  false

Instances status
[...]
```

25.9 Limitations

Currently, you can't remove tablespaces from an existing CloudNativePG cluster.

CHAPTER 26

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.

26.1 Available options

The operator looks for the following environment variables to be defined in the `ConfigMap` /`Secret` :

Name	Description
CERTIFICATE_DURATION	Determines the lifetime of the generated certificates in days. Default is 90.
CLUSTERS_ROLLOUT_DELAY	The duration (in seconds) to wait between the roll-outs of different clusters during an operator upgrade. This setting controls the timing of upgrades across clusters, spreading them out to reduce system impact. The default value is *0* which means no delay between PostgreSQL cluster upgrades.
CREATE_ANY_SERVICE	When set to *true*, will create *-any* service for the cluster. Default is *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*)
EXPIRING_CHECK_THRESHOLD	Determines the threshold, in days, for identifying a certificate as expiring. Default is 7.
INCLUDE_PLUGINS	A comma-separated list of plugins to be always included in the Cluster's reconciliation.
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
INSTANCES_ROLLOUT_DELAY	The duration (in seconds) to wait between roll-outs of individual PostgreSQL instances within the same cluster during an operator upgrade. The default value is *0*, meaning no delay between upgrades of instances in the same PostgreSQL cluster.
KUBERNETES_CLUSTER_DOMAIN	Defines the domain suffix for service FQDNs within the Kubernetes cluster. If left unset, it defaults to "cluster.local".
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
OPERATOR_IMAGE_NAME	The name of the operator image used to bootstrap Pods. Defaults to the image specified during installation.
POSTGRES_IMAGE_NAME	The name of the PostgreSQL image used by default for new clusters. Defaults to the version specified in the operator.
PULL_SECRET_NAME	Name of an additional pull secret to be defined in the operator's namespace and to be used to download images
STANDBY_TCP_USER_TIMEOUT	Defines the ``TCP_USER_TIMEOUT` socket option <https://www.postgresql.org/docs/current/runtime-config-connection.html#GUC-TCP-USER-TIMEOUT>`__ for replication connections from standby instances to the primary. Default is 0 (system's default).
DRAIN_TAINTS	Specifies the taint keys that should be interpreted as indicators of node drain. By default, it includes the taints commonly applied by Using the `cnpg plugin for *kubectl*`<*Using the `cnpg* plugin for *kubectl*>` , Cluster Autoscaler , and Karpenter : *node.kubernetes.io/unschedulable*, *ToBeDeletedByClusterAutoscaler*, *karpenter.sh/disrupted*, *karpenter.sh/disruption*.
PULL_SECRET_NAME	Name of an additional pull secret to be defined in the operator's namespace and to be used to download images

continues on next page

Table 1 – continued from previous page

Name	Description
STANDBY_TCP_USER_TIMEOUT	Defines the ``TCP_USER_TIMEOUT` socket option <https://www.postgresql.org/docs/current/runtime-config-connection.html#GUC-TCP-USER-TIMEOUT>`__ for replication connections from standby instances to the primary. Default is 0 (system's default).
DRAIN_TAINTS	Specifies the taint keys that should be interpreted as indicators of node drain. By default, it includes the taints commonly applied by Using the `cnpg plugin for *kubectl*`<*Using the `cnpg* plugin for *kubectl*>` , Cluster Autoscaler , and Karpenter : *node.kubernetes.io/unschedulable*, *ToBeDeletedByClusterAutoscaler*, *karpenter.sh/disrupted*, *karpenter.sh/disruption*.

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.

26.2 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, by enabling *in-place updates for the instancemanager* , and by spreading upgrades.

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cnpg-controller-manager-config
  namespace: cnpg-system
data:
  CLUSTERS_ROLLOUT_DELAY: 60
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: true
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload, app
  INSTANCES_ROLLOUT_DELAY: 10
```

26.3 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 instancemanager* , and by spreading upgrades.

```
apiVersion: v1
kind: Secret
metadata:
  name: cnpg-controller-manager-config
  namespace: cnpg-system
```

(continues on next page)

(continued from previous page)

```
type: Opaque
stringData:
  CLUSTERS_ROLLOUT_DELAY: 60
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: true
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload, app
  INSTANCES_ROLLOUT_DELAY: 10
```

26.4 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.

26.5 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 program's command line, with arguments separated by NULL bytes.
- `/debug/pprof/profile` . Responds with the pprof-formatted cpu profile. Profiling lasts for duration specified in seconds GET parameter, or for 30 seconds if not specified.
- `/debug/pprof/symbol` . Looks up the program counters listed in the request, responding with a table mapping program counters to function names.
- `/debug/pprof/trace` . Responds with the execution trace in binary form. Tracing lasts for duration specified in seconds GET parameter, or for 1 second if not specified.

To enable the operator you need to edit the operator deployment add the flag `--pprof-server=true` .

You can do this by executing these commands:

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

Then on the edit page scroll down the container args and add `--pprof-server=true` , as in this 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 roll-out, and the new pod will have the PPROF server enabled.

Once the pod is running you can exec inside the container by doing:

```
kubectl exec -ti -n cnpg-system <pod name> -- bash
```

Once inside execute:

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

CHAPTER 27

Instance pod configuration

27.1 Projected volumes

CloudNativePG supports mounting custom files inside the Postgres pods through `.spec.projectedVolumeTemplate`. This ability is useful for several Postgres features and extensions that require additional data files. In CloudNativePG, the `.spec.projectedVolumeTemplate` field is a projected volume template in Kubernetes that allows you to mount arbitrary data under the `/projected` folder in Postgres pods.

This simple example shows how to mount an existing TLS secret (named `sample-secret`) as files into Postgres pods. The values for the secret keys `tls.crt` and `tls.key` in `sample-secret` are mounted as files into the paths `/projected/certificate/tls.crt` and `/projected/certificate/tls.key` in the Postgres pod.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-projected-volumes
spec:
  instances: 3
  projectedVolumeTemplate:
    sources:
      - secret:
          name: sample-secret
          items:
            - key: tls.crt
              path: certificate/tls.crt
            - key: tls.key
              path: certificate/tls.key
  storage:
    size: 1Gi
```

You can find a complete example that uses a projected volume template to mount the secret and ConfigMap in the deployment manifest.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-projected-volume
spec:
  instances: 3
  projectedVolumeTemplate:
    sources:
      - secret:
          name: sample-secret
          items:
            - key: tls.crt
              path: certificate/tls.crt
            - key: tls.key
              path: certificate/tls.key
      - configMap:
          name: sample-configmap
          items:
            - key: key1
              path: config/key1
            - key: key2
              path: config/key2
  storage:
    size: 1Gi

- --
apiVersion: v1
data:
  tls.crt:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNQekNDQWVXZ0F3SUJBZ0lRSHVDY2lKcDVkZis0dHZBcDBrTk9rekFLQmo
  tls.key:␣
↪LS0tLS1CRUdJTiBFQyBQUklWQVRFIEtFWS0tLS0tCk1IY0NBUUVFSUtucWxVQjFqU2sxWW5VZzAyb0tGbXlRdDJLUEZwaFc0K1l
kind: Secret
metadata:
  name: sample-secret
type: kubernetes.io/tls
- --
apiVersion: v1
data:
  key1: value1
  key2: value2
  key3: value3
kind: ConfigMap
metadata:
  name: sample-configmap
```

27.2 Ephemeral volumes

CloudNativePG relies on ephemeral volumes

for part of the internal activities. Ephemeral volumes exist for the sole duration of a pod's life, without persisting across pod restarts.

27.2.1 Volume Claim Template for Temporary Storage

The operator uses by default an `emptyDir` volume, which can be customized by using the `.spec.ephemeralVolumesSizeLimit field`. This can be overridden by specifying a volume claim template in the `.spec.ephemeralVolumeSource` field.

In the following example, a `1Gi` ephemeral volume is set.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-ephemeral-volume-source
spec:
  instances: 3
  ephemeralVolumeSource:
    volumeClaimTemplate:
      spec:
        accessModes: ["ReadWriteOnce"]
        # example storageClassName, replace with one existing in your Kubernetes␣
↪cluster
        storageClassName: "scratch-storage-class"
        resources:
          requests:
            storage: 1Gi
```

Both `.spec.emphemeralVolumeSource` and `.spec.ephemeralVolumesSizeLimit.temporaryData` cannot be specified simultaneously.

27.2.2 Volume for shared memory

This volume is used as shared memory space for Postgres and as an ephemeral type but stored in memory. You can configure an upper bound on the size using the `.spec.ephemeralVolumesSizeLimit.shm` field in the cluster spec. Use this field only in case of PostgreSQL running with `posix shared memory dynamic allocation <PostgreSQL Configuration>` .

27.3 Environment variables

You can customize some system behavior using environment variables. One example is the `LDAPCONF` variable, which can point to a custom LDAP configuration file. Another example is the `TZ` environment variable, which represents the timezone used by the PostgreSQL container.

CloudNativePG allows you to set custom environment variables using the `env` and the `envFrom` stanza of the cluster specification.

This example defines a PostgreSQL cluster using the `Australia/Sydney` timezone as the default cluster-level timezone:

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

  env:
```

(continues on next page)

(continued from previous page)

```
  - name: TZ
    value: Australia/Sydney

  storage:
    size: 1Gi
```

The `envFrom` stanza can refer to ConfigMaps or secrets to use their content as environment variables:

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

  envFrom:
  - configMapRef:
      name: config-map-name
  - secretRef:
      name: secret-name

  storage:
    size: 1Gi
```

The operator doesn't allow setting the following environment variables:

- `POD_NAME`
- `NAMESPACE`
- Any environment variable whose name starts with `PG` .

Any change in the `env` or in the `envFrom` section triggers a rolling update of the PostgreSQL pods.

If the `env` or the `envFrom` section refers to a secret or a ConfigMap, the operator doesn't detect any changes in them and doesn't trigger a rollout. The kubelet uses the same behavior with pods, and you must trigger the pod rollout manually.

CHAPTER 28

Storage

Storage is the most critical component in a database workload. Storage must always be available, scale, perform well, and guarantee consistency and durability. The same expectations and requirements that apply to traditional environments, such as virtual machines and bare metal, are also valid in container contexts managed by Kubernetes.

Important

When it comes to dynamically provisioned storage, Kubernetes has its own specifics. These include *storage classes*, *persistent volumes*, and *Persistent Volume Claims (PVCs)*. You need to own these concepts, on top of all the valuable knowledge you've built over the years in terms of storage for database workloads on VMs and physical servers.

There are two primary methods of access to storage:

- **Network** – Either directly or indirectly. (Think of an NFS volume locally mounted on a host running Kubernetes.)
- **Local** – Directly attached to the node where a pod is running. This also includes directly attached disks on bare metal installations of Kubernetes.

Network storage, which is the most common usage pattern in Kubernetes, presents the same issues of throughput and latency that you can experience in a traditional environment. These issues can be accentuated in a shared environment, where I/O contention with several applications increases the variability of performance results.

Local storage enables shared-nothing architectures, which is more suitable for high transactional and very large database (VLDB) workloads, as it guarantees higher and more predictable performance.

Warning

Before you deploy a PostgreSQL cluster with CloudNativePG, ensure that the storage you're using is recommended for database workloads. We recommend clearly setting performance expectations by first benchmarking the storage using tools such as fio

and then the database using pgbench .

> **Note**
>
> CloudNativePG doesn't use *StatefulSet* for managing data persistence. Rather, it manages PVCs directly. If you want to know more, see *Custom Pod Controller* .

28.1 Backup and recovery

Since CloudNativePG supports volume snapshots for both backup and recovery, we recommend that you also consider this aspect when you choose your storage solution, especially if you manage very large databases.

> **Important**
>
> See the Kubernetes documentation for a list of all the supported container storage interface (CSI) drivers

that provide snapshot capabilities.

28.2 Benchmarking CloudNativePG

Before deploying the database in production, we recommend that you benchmark CloudNativePG in a controlled Kubernetes environment. Follow the guidelines in *Benchmarking* .

Briefly, we recommend operating at two levels:

- Measuring the performance of the underlying storage using fio, with relevant metrics for database workloads such as throughput for sequential reads, sequential writes, random reads, and random writes
- Measuring the performance of the database using pgbench, the default benchmarking tool distributed with PostgreSQL

> **Important**
>
> You must measure both the storage and database performance before putting the database into production. These results are extremely valuable not just in the planning phase (for example, capacity planning). They are also valuable in the production lifecycle, particularly in emergency situations when you don't have time to run this kind of test. Databases change and evolve over time, and so does the distribution of data, potentially affecting performance. Knowing the theoretical maximum throughput of sequential reads or writes is extremely useful in those situations. This is true especially in shared-nothing contexts, where results don't vary due to the influence of external workloads.

Know your system: benchmark it.

28.3 Encryption at rest

Encryption at rest is possible with CloudNativePG. The operator delegates that to the underlying storage class. See the storage class for information about this important security feature.

28.4 Persistent Volume Claim (PVC)

The operator creates a PVC for each PostgreSQL instance, with the goal of storing the `PGDATA` . It then mounts it into each pod.

Additionally, it supports creating clusters with:

- A separate PVC on which to store PostgreSQL WAL, as explained in *Volume for WAL*
- Additional separate volumes reserved for PostgreSQL tablespaces, as explained in TablespaceState {postgresql-cnpg-io-v1-TablespaceState}

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

28.5 Configuration via a storage class

Important

CloudNativePG was designed to work interchangeably with all storage classes. As usual, we recommend properly benchmarking the storage class in a controlled environment before deploying to production.

The easiest way to configure the storage for a PostgreSQL class is to request storage of a certain size, like in the following example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-class
spec:
  instances: 3
  storage:
    size: 1Gi
```

Using the previous configuration, the generated PVCs are satisfied by the default storage class. If the target Kubernetes cluster has no default storage class, or even if you need your PVCs to be satisfied by a known storage class, you can set it into the custom resource:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-class
spec:
  instances: 3
  storage:
    storageClass: standard
    size: 1Gi
```

28.6 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:

```yaml
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
```

28.7 Volume for WAL

By default, PostgreSQL stores all its data in the so-called `PGDATA` (a directory). One of the core directories inside `PGDATA` is `pg_wal` , which contains the log of transactional changes that occurred in the database, in the form of segment files. (`pg_wal` is historically known as `pg_xlog` in PostgreSQL.)

Note

Normally, each segment is 16MB in size, but you can configure the size using the *walSegmentSize* option. This option is applied at cluster initialization time, as described in Bootstrap an empty cluster (`initdb ) <Bootstrap an empty cluster (*initdb* )>` .

In most cases, having `pg_wal` on the same volume where `PGDATA` resides is fine. However, having WALs stored in a separate volume has a few benefits:

- **I/O performance** – By storing WAL files on different storage from `PGDATA` , PostgreSQL can exploit parallel I/O for WAL operations (normally sequential writes) and for data files (tables and indexes for example), thus improving vertical scalability.
- **More reliability** – By reserving dedicated disk space to WAL files, you can be sure that exhausting space on the `PGDATA` volume never interferes with WAL writing. This behavior ensures that your PostgreSQL primary is correctly shut down.
- **Finer control** – You can define the amount of space dedicated to both `PGDATA` and `pg_wal` , fine tune WAL configuration

and checkpoints, and even use a different storage class for cost optimization.

- **Better I/O monitoring** – You can constantly monitor the load and disk usage on both `PGDATA` and `pg_wal` . You can also set alerts that notify you in case, for example, `PGDATA` requires resizing.

See also

See Reliability and the Write-Ahead Log

in the PostgreSQL documentation for more information.

You can add a separate volume for WAL using the `.spec.walStorage` option. It follows the same rules described for the `storage` field and provisions a dedicated PVC. For example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: separate-pgwal-volume
spec:
  instances: 3
  storage:
    size: 1Gi
  walStorage:
    size: 1Gi
```

Important

Removing *walStorage* isn't supported. Once added, a separate volume for WALs can't be removed from an existing Postgres cluster.

28.8 Volumes for tablespaces

CloudNativePG supports declarative tablespaces. You can add one or more volumes, each dedicated to a single PostgreSQL tablespace. See TablespaceState {postgresql-cnpg-io-v1-TablespaceState} for details.

28.9 Volume expansion

Kubernetes exposes an API allowing *expanding PVCs*

that's enabled by default. However, it needs to be supported by the underlying `StorageClass` .

To check if a certain `StorageClass` supports volume expansion, you can read the `allowVolumeExpansion` field for your storage class:

```
$ kubectl get storageclass -o jsonpath={$.allowVolumeExpansion} premium-storage
true
```

28.9.1 Using the volume expansion Kubernetes feature

Given the storage class supports volume expansion, you can change the size requirement of the `Cluster` , and the operator applies the change to every PVC.

If the `StorageClass` supports *online volume resizing* , the change is immediately applied to the pods. If the underlying storage class doesn't support that, you must delete the pod to trigger the resize.

The best way to proceed is to delete one pod at a time, starting from replicas and waiting for each pod to be back up.

28.9.2 Re-creating storage

If the storage class doesn't support volume expansion, you can still regenerate your cluster on different PVCs. Allocate new PVCs with increased storage and then move the database there. This operation is feasible only when the cluster contains more than one node.

While you do that, you need to prevent the operator from changing the existing PVC by disabling the `resizeInUseVolumes` flag, like in the following example:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-pvc-template
spec:
  instances: 3

  storage:
    storageClass: standard
    size: 1Gi
    resizeInUseVolumes: False
```

To move the entire cluster to a different storage area, you need to re-create all the PVCs and all the pods. Suppose you have a cluster with three replicas, like in the following example:

```
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
cluster-example-1   1/1     Running   0          2m37s
cluster-example-2   1/1     Running   0          2m22s
cluster-example-3   1/1     Running   0          2m10s
```

To re-create the cluster using different PVCs, you can edit the cluster definition to disable `resizeInUseVolumes`. Then re-create every instance in a different PVC.

For example, re-create the storage for `cluster-example-3`:

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

Important

If you created a dedicated WAL volume, both PVCs must be deleted during this process. The same procedure applies if you want to regenerate the WAL volume PVC. You can do this by also disabling *resizeInUseVolumes* for the *.spec.walStorage* section.

For example, if a PVC dedicated to WAL storage is present:

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

Having done that, the operator orchestrates creating another replica with a resized PVC:

```
$ kubectl get pods
NAME                          READY   STATUS      RESTARTS   AGE
cluster-example-1             1/1     Running     0          5m58s
cluster-example-2             1/1     Running     0          5m43s
cluster-example-4-join-v2     0/1     Completed   0          17s
cluster-example-4             1/1     Running     0          10s
```

28.10 Static provisioning of persistent volumes

CloudNativePG was designed to work with dynamic volume provisioning. This capability allows storage volumes to be created on demand when requested by users by way of storage classes and PVC templates. See *Re-creating storage* .

However, in some cases, Kubernetes administrators prefer to manually create storage volumes and then create the related `PersistentVolume` objects for their representation inside the Kubernetes cluster. This is also known as *pre-provisioning* of volumes.

Important

We recommend that you avoid pre-provisioning volumes, as it has an effect on the high availability and self-healing capabilities of the operator. It breaks the fully declarative model on which CloudNativePG was built. To use a pre-provisioned volume in CloudNativePG:

1. Manually create the volume outside Kubernetes.
2. Create the `PersistentVolume` object to match this volume using the correct parameters as required by the actual CSI driver (that is, `volumeHandle` , `fsType` , `storageClassName` , and so on).
3. Create the Postgres `Cluster` using, for each storage section, a coherent *pvcTemplate*

section that can help Kubernetes match the `PersistentVolume` and enable CloudNativePG to create the needed `PersistentVolumeClaim` .

Warning

With static provisioning, it's your responsibility to ensure that Postgres pods can be correctly scheduled by Kubernetes where a pre-provisioned volume exists. (The scheduling configuration is based on the affinity rules of your cluster.) Make sure you check for any pods stuck in *Pending* after you deploy the cluster. If the condition persists, investigate why it's happening.

28.11 Block storage considerations (Ceph/Longhorn)

Most block storage solutions in Kubernetes, such as Longhorn and Ceph, recommend having multiple replicas of a volume to enhance resiliency. This approach works well for workloads that lack built-in resiliency.

However, CloudNativePG integrates this resiliency directly into the Postgres `Cluster` through the number of instances and the persistent volumes attached to them, as explained in *Synchronizing the state* .

As a result, defining additional replicas at the storage level can lead to write amplification, unnecessarily increasing disk I/O and space usage.

For CloudNativePG usage, consider reducing the number of replicas at the block storage level to one, while ensuring that no single point of failure (SPoF) exists at the storage level for the entire `Cluster` resource. This typically means ensuring that a single storage host—and ultimately, a physical disk—does not host blocks from different instances of the same `Cluster` , in alignment with the broader *shared-nothing architecture* principle.

In Longhorn, you can mitigate this risk by enabling strict-local data locality when creating a custom storage class. Detailed instructions for creating a volume with strict-local data locality are available here . This setting ensures that a pod's data volume resides on the same node as the pod itself.

Additionally, your Postgres `Cluster` should have *pod anti-affinity rules*

in place to ensure that the operator deploys pods across different nodes, allowing Longhorn to place the data volumes on the corresponding hosts. If needed, you can manually relocate volumes in Longhorn by temporarily setting the volume replica count to 2, reducing it afterward, and then removing the old replica. If a host becomes corrupted, you can use the Backup {postgresql-cnpg-io-v1-Backup}

the affected instance. CloudNativePG will then recreate the instance on another host and replicate the data.

In Ceph, this can be configured through CRUSH rules. The documentation for configuring CRUSH rules is available here . These rules aim to ensure one volume per pod per node. You can also relocate volumes by importing them into a different pool.

CHAPTER 29

Labels and annotations

Resources in Kubernetes are organized in a flat structure, with no hierarchical information or relationship between them. However, such resources and objects can be linked together and put in relationship through *labels* and *annotations*.

> **Note**
>
> For more information, see the Kubernetes documentation on annotations and labels .

In brief:

- An annotation is used to assign additional non-identifying information to resources with the goal of facilitating integration with external tools.
- A label is used to group objects and query them through the Kubernetes native selector capability.

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

> **Note**
>
> Label and annotation inheritance is the technique adopted by CloudNativePG instead of alternative approaches such as pod templates.

29.1 Predefined labels

CloudNativePG manages the following predefined labels:

`cnpg.io/backupDate` : The date of the backup in ISO 8601 format (`YYYYMMDD`). This label is available only on `VolumeSnapshot` resources.

`cnpg.io/backupName` : Backup identifier. This label is available only on `VolumeSnapshot` resources.

`cnpg.io/backupMonth` : The year/month when a backup was taken. This label is available only on `VolumeSnapshot` resources.

`cnpg.io/backupTimeline` : The timeline of the instance when a backup was taken. This label is available only on `VolumeSnapshot` resources.

`cnpg.io/backupYear` : The year a backup was taken. This label is available only on `VolumeSnapshot` resources.

`cnpg.io/cluster` : Name of the cluster.

`cnpg.io/immediateBackup` : Applied to a `Backup` resource if the backup is the first one created from a `ScheduledBackup` object having `immediate` set to `true` .

`cnpg.io/instanceName` : Name of the PostgreSQL instance (replaces the old and deprecated `postgresql` label).

`cnpg.io/jobRole` : Role of the job (that is, `import` , `initdb` , `join` , ...)

`cnpg.io/onlineBackup` : Whether the backup is online (hot) or taken when Postgres is down (cold). This label is available only on `VolumeSnapshot` resources.

`cnpg.io/podRole` : Distinguishes pods dedicated to pooler deployment from those used for database instances.

`cnpg.io/poolerName` : Name of the PgBouncer pooler.

`cnpg.io/pvcRole` : Purpose of the PVC, such as `PG_DATA` or `PG_WAL` .

`cnpg.io/reload` : Available on `ConfigMap` and `Secret` resources. When set to `true` , a change in the resource is automatically reloaded by the operator.

`cnpg.io/userType` : Specifies the type of PostgreSQL user associated with the `Secret` , either `superuser` (Postgres superuser access) or `app` (application-level user in CloudNativePG terminology), and is limited to the default users created by CloudNativePG (typically `postgres` and `app`).

`role` - **deprecated** : Whether the instance running in a pod is a `primary` or a `replica` . This label is deprecated, you should use `cnpg.io/instanceRole` instead.

`cnpg.io/scheduled-backup` : When available, name of the `ScheduledBackup` resource that created a given `Backup` object.

`cnpg.io/instanceRole` : Whether the instance running in a pod is a `primary` or a `replica` .

29.2 Predefined annotations

CloudNativePG manages the following predefined annotations:

`container.apparmor.security.beta.kubernetes.io/*` : Name of the AppArmor profile to apply to the named container. See *Restricting Pod access using AppArmor*

for details.

`cnpg.io/backupEndTime` : The time a backup ended. This annotation is available only on `VolumeSnapshot` resources.

`cnpg.io/backupEndWAL` : The WAL at the conclusion of a backup. This annotation is available only on `VolumeSnapshot` resources.

`cnpg.io/backupStartTime` : The time a backup started.

`cnpg.io/backupStartWAL` : The WAL at the start of a backup. This annotation is available only on `VolumeSnapshot` resources.

`cnpg.io/coredumpFilter` : Filter to control the coredump of Postgres processes, expressed with a bitmask. By default it's set to `0x31` to exclude shared memory segments from the dump. See *PostgreSQL core dumps*

for more information.

`cnpg.io/clusterManifest` : Manifest of the `Cluster` owning this resource (such as a PVC). This label replaces the old, deprecated `cnpg.io/hibernateClusterManifest` label.

`cnpg.io/fencedInstances` : List of the instances that need to be fenced, expressed in JSON format. The whole cluster is fenced if the list contains the `*` element.

`cnpg.io/forceLegacyBackup` : Applied to a `Cluster` resource for testing purposes only, to simulate the behavior of `barman-cloud-backup` prior to version 3.4 (Jan 2023) when the `--name` option wasn't available.

`cnpg.io/hash` : The hash value of the resource.

`cnpg.io/hibernation` : Applied to a `Cluster` resource to control the *declarative hibernation feature* . Allowed values are `on` and `off` .

`cnpg.io/managedSecrets` : Pull secrets managed by the operator and automatically set in the `ServiceAccount` resources for each Postgres cluster.

`cnpg.io/nodeSerial` : On a pod resource, identifies the serial number of the instance within the Postgres cluster.

`cnpg.io/operatorVersion` : Version of the operator.

`cnpg.io/pgControldata` : Output of the `pg_controldata` command. This annotation replaces the old, deprecated `cnpg.io/hibernatePgControlData` annotation.

`cnpg.io/podEnvHash` : Deprecated, as the `cnpg.io/podSpec` annotation now also contains the pod environment.

`cnpg.io/podPatch` : Annotation can be applied on a `Cluster` resource.

When set to JSON-patch formatted patch, the patch will be applied on the instance Pods.

□□ **WARNING:** This feature may introduce discrepancies between the operator's expectations and Kubernetes behavior. Use with caution and only as a last resort.

IMPORTANT: adding or changing this annotation won't trigger a rolling deployment of the generated Pods. The latter can be triggered manually by the user with `kubectl cnpg restart` .

`cnpg.io/podSpec` : Snapshot of the `spec` of the pod generated by the operator. This annotation replaces the old, deprecated `cnpg.io/podEnvHash` annotation.

`cnpg.io/poolerSpecHash` : Hash of the pooler resource.

`cnpg.io/pvcStatus` : Current status of the PVC: `initializing` , `ready` , or `detached` .

`cnpg.io/reconcilePodSpec` : Annotation can be applied to a `Cluster` or `Pooler` to prevent restarts.

When set to `disabled` on a `Cluster` , the operator prevents instances from restarting due to changes in the PodSpec. This includes changes to:

```
- Topology or affinity - Scheduler - Volumes or containers
```

When set to `disabled` on a `Pooler` , the operator restricts any modifications to the deployment specification, except for changes to `spec.instances` .

`cnpg.io/reconciliationLoop` : When set to `disabled` on a `Cluster` , the operator prevents the reconciliation loop from running.

`cnpg.io/reloadedAt` : Contains the latest cluster `reload` time. `reload` is triggered by the user through a plugin.

`cnpg.io/skipEmptyWalArchiveCheck` : When set to `enabled` on a `Cluster` resource, the operator disables the check that ensures that the WAL archive is empty before writing data. Use at your own risk.

`cnpg.io/skipWalArchiving` : When set to `enabled` on a `Cluster` resource, the operator disables WAL archiving. This will set `archive_mode` to `off` and require a restart of all PostgreSQL instances. Use at your own risk.

`cnpg.io/snapshotStartTime` : The time a snapshot started.

`cnpg.io/snapshotEndTime` : The time a snapshot was marked as ready to use.

`cnpg.io/validation` : When set to `disabled` on a CloudNativePG-managed custom resource, the validation webhook allows all changes without restriction.

□□ **WARNING:** Disabling validation may permit unsafe or destructive operations. Use this setting with caution and at your own risk.

`cnpg.io/volumeSnapshotDeadline` : Applied to `Backup` and `ScheduledBackup` resources, allows you to control how long the operator should retry recoverable errors before considering the volume snapshot backup failed. In minutes, defaulting to 10.

`kubectl.kubernetes.io/restartedAt` : When available, the time of last requested restart of a Postgres cluster.

29.3 Prerequisites

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

The following continues from that example and limits it to the following:

- Annotations: `categories`
- Labels: `app` , `environment` , and `workload`

> **Note**
>
> Feel free to select the names that most suit your context for both annotations and labels. You can also use wildcards in naming and adopt strategies like using *mycompany/** for all labels or setting annotations starting with *mycompany/* to be inherited.

29.4 Defining cluster's metadata

When defining the cluster, before any resource is deployed, you can set the metadata as follows:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
  annotations:
    categories: database
  labels:
    environment: production
    workload: database
    app: sso
spec:
     # ... <snip>
```

Once the cluster is deployed, you can verify, for example, that the labels were correctly set in the pods:

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

29.5 Current limitations

Currently, CloudNativePG doesn't automatically propagate labels or annotations deletions. Therefore, when an annotation or label is removed from a cluster that was previously propagated to the underlying pods, the operator doesn't remove it on the associated resources.

CHAPTER 30

Monitoring

Important

Installing Prometheus and Grafana is beyond the scope of this project. We assume they are correctly installed in your system. However, for experimentation we provide instructions in *Part 4 of the Quickstart* .

30.1 Monitoring Instances

For each PostgreSQL instance, the operator provides an exporter of metrics for Prometheus via HTTP or HTTPS, 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

CloudNativePG, by default, installs a set of *Default set of metrics*

in a `ConfigMap` named `default-monitoring` .

Note

You can inspect the exported metrics by following the instructions in the *How to inspect the exported metrics*

section below.

All monitoring queries that are performed on PostgreSQL are:

- atomic (one transaction per query)
- executed with the `pg_monitor` role

- executed with `application_name` set to `cnpg_metrics_exporter`
- executed as user `postgres`

Please refer to the "Predefined Roles" section in PostgreSQL documentation

for details on the `pg_monitor` role.

Queries, by default, are run against the *main database*, as defined by the specified `bootstrap` method of the `Cluster` resource, according to the following logic:

- using `initdb` : queries will be run by default against the specified database in `initdb.database` , or `app` if not specified
- using `recovery` : queries will be run by default against the specified database in `recovery.database` , or `postgres` if not specified
- using `pg_basebackup` : queries will be run by default against the specified database in `pg_basebackup.database` , or `postgres` if not specified

The default database can always be overridden for a given user-defined metric, by specifying a list of one or more databases in the `target_databases` option.

See also

If you are interested in evaluating the integration of CloudNativePG with Prometheus and Grafana, you can find a quick setup guide in *Part 4 of the quickstart*

30.1.1 Monitoring with the Prometheus operator

A specific PostgreSQL cluster can be monitored using the Prometheus Operator's resource PodMonitor .

A `PodMonitor` that correctly points to the 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, define it as follows and adjust 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

Ensure you modify the example above with a unique name, as well as the correct cluster's namespace and labels (e.g., *cluster-example*).

Important

The *postgresql* label, used in previous versions of this document, is deprecated and will be removed in the future. Please use the *cnpg.io/cluster* label instead to select the instances.

30.1.2 Enabling TLS on the Metrics Port

To enable TLS communication on the metrics port, configure the `.spec.monitoring.tls.enabled` setting to `true` . This setup ensures that the metrics exporter uses the same server certificate used by PostgreSQL to secure communication on port 5432.

Important

Changing the *.spec.monitoring.tls.enabled* setting will trigger a rolling restart of the Cluster.

If the `PodMonitor` is managed by the operator (`.spec.monitoring.enablePodMonitor` set to `true`), it will automatically contain the necessary configurations to access the metrics via TLS.

To manually deploy a `PodMonitor` suitable for reading metrics via TLS, define it as follows and adjust as needed:

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cluster-example
spec:
  selector:
    matchLabels:
      "cnpg.io/cluster": cluster-example
  podMetricsEndpoints:
  - port: metrics
    scheme: https
    tlsConfig:
      ca:
        secret:
          name: cluster-example-ca
          key: ca.crt
      serverName: cluster-example-rw
```

Important

Ensure you modify the example above with a unique name, as well as the correct Cluster's namespace and labels (e.g., *cluster-example*).

Important

The *serverName* field in the metrics endpoint must match one of the names defined in the server certificate. If the default certificate is in use, the *serverName* value should be in the format *<cluster-name>-rw* .

30.1.3 Predefined set of metrics

Every PostgreSQL instance exporter automatically exposes a set of predefined metrics, which can be classified in two major categories:

- PostgreSQL related metrics, starting with `cnpg_collector_*` , including:
  - number of WAL files and total size on disk
  - number of `.ready` and `.done` files in the archive status folder
  - requested minimum and maximum number of synchronous replicas, as well as the expected and actually observed values
  - number of distinct nodes accommodating the instances
  - timestamps indicating last failed and last available backup, as well as the first point of recoverability for the cluster
  - flag indicating if replica cluster mode is enabled or disabled
  - flag indicating if a manual switchover is required
  - flag indicating if fencing is enabled or disabled
- Go runtime related metrics, starting with `go_*`

Below is a sample of the metrics returned by the `localhost:9187/metrics` endpoint of an instance. As you can see, the Prometheus format is self-documenting:

```
#  HELP cnpg_collector_collection_duration_seconds Collection time duration in seconds

#  TYPE cnpg_collector_collection_duration_seconds gauge

cnpg_collector_collection_duration_seconds{collector="Collect.up"} 0.0031393

#  HELP cnpg_collector_collections_total Total number of times PostgreSQL was␣
↪accessed for metrics.

#  TYPE cnpg_collector_collections_total counter

cnpg_collector_collections_total 2

#  HELP cnpg_collector_fencing_on 1 if the instance is fenced, 0 otherwise

#  TYPE cnpg_collector_fencing_on gauge

cnpg_collector_fencing_on 0

#  HELP cnpg_collector_nodes_used NodesUsed represents the count of distinct nodes␣
↪accommodating the instances. A value of -1 suggests that the metric is not␣
↪available. A value of 1 suggests that all instances are hosted on a single node,␣
↪implying the absence of High Availability (HA). Ideally this value should match the␣
```

(continues on next page)

(continued from previous page)

```
↪number of instances in the cluster.

#  TYPE cnpg_collector_nodes_used gauge

cnpg_collector_nodes_used 3

#  HELP cnpg_collector_last_collection_error 1 if the last collection ended with␣
↪error, 0 otherwise.

#  TYPE cnpg_collector_last_collection_error gauge

cnpg_collector_last_collection_error 0

#  HELP cnpg_collector_manual_switchover_required 1 if a manual switchover is␣
↪required, 0 otherwise

#  TYPE cnpg_collector_manual_switchover_required gauge

cnpg_collector_manual_switchover_required 0

#  HELP cnpg_collector_pg_wal Total size in bytes of WAL segments in the /var/lib/
↪postgresql/data/pgdata/pg_wal directory  computed as (wal_segment_size * count)

#  TYPE cnpg_collector_pg_wal gauge

cnpg_collector_pg_wal{value="count"} 9
cnpg_collector_pg_wal{value="slots_max"} NaN
cnpg_collector_pg_wal{value="keep"} 32
cnpg_collector_pg_wal{value="max"} 64
cnpg_collector_pg_wal{value="min"} 5
cnpg_collector_pg_wal{value="size"} 1.50994944e+08
cnpg_collector_pg_wal{value="volume_max"} 128
cnpg_collector_pg_wal{value="volume_size"} 2.147483648e+09

#  HELP cnpg_collector_pg_wal_archive_status Number of WAL segments in the /var/lib/
↪postgresql/data/pgdata/pg_wal/archive_status directory (ready, done)

#  TYPE cnpg_collector_pg_wal_archive_status gauge

cnpg_collector_pg_wal_archive_status{value="done"} 6
cnpg_collector_pg_wal_archive_status{value="ready"} 0

#  HELP cnpg_collector_replica_mode 1 if the cluster is in replica mode, 0 otherwise

#  TYPE cnpg_collector_replica_mode gauge

cnpg_collector_replica_mode 0

#  HELP cnpg_collector_sync_replicas Number of requested synchronous replicas␣
↪(synchronous_standby_names)

#  TYPE cnpg_collector_sync_replicas gauge
```

(continues on next page)

(continued from previous page)

```
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="17.5"} 17.5

#  HELP cnpg_collector_last_failed_backup_timestamp The last failed backup as a unix␣
↪timestamp (Deprecated)

#  TYPE cnpg_collector_last_failed_backup_timestamp gauge

cnpg_collector_last_failed_backup_timestamp 0

#  HELP cnpg_collector_last_available_backup_timestamp The last available backup as a␣
↪unix timestamp (Deprecated)

#  TYPE cnpg_collector_last_available_backup_timestamp gauge

cnpg_collector_last_available_backup_timestamp 1.63238406e+09

#  HELP cnpg_collector_first_recoverability_point The first point of recoverability␣
↪for the cluster as a unix timestamp (Deprecated)

#  TYPE cnpg_collector_first_recoverability_point gauge

cnpg_collector_first_recoverability_point 1.63238406e+09

#  HELP cnpg_collector_lo_pages Estimated number of pages in the pg_largeobject table

#  TYPE cnpg_collector_lo_pages gauge

cnpg_collector_lo_pages{datname="app"} 0
cnpg_collector_lo_pages{datname="postgres"} 78

#  HELP cnpg_collector_wal_buffers_full Number of times WAL data was written to disk␣
↪because WAL buffers became full. Only available on PG 14+

#  TYPE cnpg_collector_wal_buffers_full gauge

cnpg_collector_wal_buffers_full{stats_reset="2023-06-19T10:51:27.473259Z"} 6472
```

(continues on next page)

(continued from previous page)

```
# HELP cnpg_collector_wal_bytes Total amount of WAL generated in bytes. Only␣
↪available on PG 14+

# TYPE cnpg_collector_wal_bytes gauge

cnpg_collector_wal_bytes{stats_reset="2023-06-19T10:51:27.473259Z"} 1.0035147e+07

# HELP cnpg_collector_wal_fpi Total number of WAL full page images generated. Only␣
↪available on PG 14+

# TYPE cnpg_collector_wal_fpi gauge

cnpg_collector_wal_fpi{stats_reset="2023-06-19T10:51:27.473259Z"} 1474

# HELP cnpg_collector_wal_records Total number of WAL records generated. Only␣
↪available on PG 14+

# TYPE cnpg_collector_wal_records gauge

cnpg_collector_wal_records{stats_reset="2023-06-19T10:51:27.473259Z"} 26178

# HELP cnpg_collector_wal_sync Number of times WAL files were synced to disk via␣
↪issue_xlog_fsync request (if fsync is on and wal_sync_method is either fdatasync,␣
↪fsync or fsync_writethrough, otherwise zero). Only available on PG 14+

# TYPE cnpg_collector_wal_sync gauge

cnpg_collector_wal_sync{stats_reset="2023-06-19T10:51:27.473259Z"} 37

# HELP cnpg_collector_wal_sync_time Total amount of time spent syncing WAL files to␣
↪disk via issue_xlog_fsync request, in milliseconds (if track_wal_io_timing is␣
↪enabled, fsync is on, and wal_sync_method is either fdatasync, fsync or fsync_
↪writethrough, otherwise zero). Only available on PG 14+

# TYPE cnpg_collector_wal_sync_time gauge

cnpg_collector_wal_sync_time{stats_reset="2023-06-19T10:51:27.473259Z"} 0

# HELP cnpg_collector_wal_write Number of times WAL buffers were written out to disk␣
↪via XLogWrite request. Only available on PG 14+

# TYPE cnpg_collector_wal_write gauge

cnpg_collector_wal_write{stats_reset="2023-06-19T10:51:27.473259Z"} 7243

# HELP cnpg_collector_wal_write_time Total amount of time spent writing WAL buffers␣
↪to disk via XLogWrite request, in milliseconds (if track_wal_io_timing is enabled,␣
↪otherwise zero). This includes the sync time when wal_sync_method is either open_
↪datasync or open_sync. Only available on PG 14+

# TYPE cnpg_collector_wal_write_time gauge
```

(continues on next page)

(continued from previous page)

```
cnpg_collector_wal_write_time{stats_reset="2023-06-19T10:51:27.473259Z"} 0

#  HELP cnpg_last_error 1 if the last collection ended with error, 0 otherwise.

#  TYPE cnpg_last_error gauge

cnpg_last_error 0

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

#  TYPE go_gc_duration_seconds summary

go_gc_duration_seconds{quantile="0"} 5.01e-05
go_gc_duration_seconds{quantile="0.25"} 7.27e-05
go_gc_duration_seconds{quantile="0.5"} 0.0001748
go_gc_duration_seconds{quantile="0.75"} 0.0002959
go_gc_duration_seconds{quantile="1"} 0.0012776
go_gc_duration_seconds_sum 0.0035741
go_gc_duration_seconds_count 13

#  HELP go_goroutines Number of goroutines that currently exist.

#  TYPE go_goroutines gauge

go_goroutines 25

#  HELP go_info Information about the Go environment.

#  TYPE go_info gauge

go_info{version="go1.20.5"} 1

#  HELP go_memstats_alloc_bytes Number of bytes allocated and still in use.

#  TYPE go_memstats_alloc_bytes gauge

go_memstats_alloc_bytes 4.493744e+06

#  HELP go_memstats_alloc_bytes_total Total number of bytes allocated, even if freed.

#  TYPE go_memstats_alloc_bytes_total counter

go_memstats_alloc_bytes_total 2.1698216e+07

#  HELP go_memstats_buck_hash_sys_bytes Number of bytes used by the profiling bucket␣
↪hash table.

#  TYPE go_memstats_buck_hash_sys_bytes gauge

go_memstats_buck_hash_sys_bytes 1.456234e+06
```

(continues on next page)

(continued from previous page)

```
# 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

# 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.
```

(continues on next page)

(continued from previous page)

```
#  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

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.
```

(continues on next page)

(continued from previous page)

```
#  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

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* .

Warning

The metrics *cnpg_collector_last_failed_backup_timestamp* , *cnpg_collector_last_available_backup_timestamp* , and *cnpg_collector_first_recoverability_point* have been deprecated starting from version 1.26. These metrics will continue to function with native backup solutions such as in-core Barman Cloud (deprecated) and volume snapshots. Note that for these cases, *cnpg_collector_first_recoverability_point* and *cnpg_collector_last_available_backup_timestamp* will remain zero until the first backup is completed to the object store. This is separate from WAL archiving.

30.1.4 User defined metrics

This feature is currently in *beta* state and the format is inspired by the queries.yaml file (release 0.12)

of the PostgreSQL Prometheus Exporter.

Custom metrics can be defined by users by referring to the created `Configmap` /`Secret` in a `Cluster` definition under the `.spec.monitoring.customQueriesConfigMap` or `customQueriesSecret` section as in the following example:

```
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 **in the same namespace as the Cluster** resource.

Note

If you want ConfigMaps and Secrets to be **automatically** reloaded by instances, you can add a label with key *cnpg.io/reload* to it, otherwise you will have to reload the instances using the *kubectl cnpg reload* subcommand.

Important

When a user defined metric overwrites an already existing metric the instance manager prints a json warning log, containing the message:*Query with the same name already found. Overwriting the existing one.* and a key *queryName* containing the overwritten query name.

30.1.5 Example of a user defined metric

Here you can see an example of a `ConfigMap` containing a single custom query, referenced by the `Cluster` example above:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-monitoring
  namespace: test
  labels:
```

(continues on next page)

(continued from previous page)

```
    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*).

30.1.6 Example of a user defined metric with predicate query

The `predicate_query` option allows the user to execute the `query` to collect the metrics only under the specified conditions. To do so the user needs to provide a predicate query that returns at most one row with a single `boolean` column.

The predicate query is executed in the same transaction as the main query and against the same databases.

```
some_query: |
  predicate_query: |
    SELECT
      some_bool as predicate
    FROM some_table
  query: |
    SELECT
     count(*) as rows
    FROM some_table
  metrics:
    - rows:
        usage: "GAUGE"
        description: "number of rows"
```

30.1.7 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"
```

(continues on next page)

(continued from previous page)

```
    - rows:
        usage: "GAUGE"
        description: "number of rows"
  target_databases:
    - "*"
    - "template1"
```

The above example will produce the following metrics (provided the databases exist):

```
cnpg_some_query_rows{datname="albert"} 2
cnpg_some_query_rows{datname="bb"} 5
cnpg_some_query_rows{datname="freddie"} 10
cnpg_some_query_rows{datname="template1"} 7
cnpg_some_query_rows{datname="postgres"} 42
```

30.1.8 Structure of a user defined metric

Every custom query has the following basic structure:

```
<MetricName>:
      query: "<SQLQuery>"
      metrics:
        - <ColumnName>:
            usage: "<MetricType>"
            description: "<MetricDescription>"
```

Here is a short description of all the available fields:

- `<MetricName>` : the name of the Prometheus metric
  - `name` : override `<MetricName>` , if defined
  - `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 *Example of a user defined metric running on multiple databases*

to enable auto discovery. Overwrites the default database if provided. - `predicate_query` : a SQL query that returns at most one row and one `boolean` column to run on the target database. The system evaluates the predicate and if `true` executes the `query` . - `metrics` : section containing a list of all exported columns, defined as follows: - `<ColumnName>` : the name of the column returned by the query - `name` : override the `ColumnName` of the column in the metric, if defined - `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.

30.1.9 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
```

30.1.10 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.

30.1.11 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.

30.2 Monitoring the CloudNativePG operator

The operator internally exposes Prometheus metrics via HTTP on port 8080, named `metrics` .

Note

You can inspect the exported metrics by following the instructions in the *How to inspect the exported metrics*

section below.

Currently, the operator exposes default `kubebuilder` metrics. See kubebuilder documentation

for more details.

30.2.1 Monitoring the operator with Prometheus

The operator can be monitored using the Prometheus Operator by defining a PodMonitor

pointing to the operator pod(s), as follows (note it's applied in the same namespace as the operator):

```
kubectl -n cnpg-system apply -f - <<EOF
- --
apiVersion: monitoring.coreos.com/v1
```

(continues on next page)

(continued from previous page)

```
kind: PodMonitor
metadata:
  name: cnpg-controller-manager
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: cloudnative-pg
  podMetricsEndpoints:
    - port: metrics
EOF
```

30.3 How to inspect the exported metrics

In this section we provide basic instructions on how to inspect the metrics exported by a specific PostgreSQL instance manager (primary or replica) or the operator.

> **Note**
>
> In the examples below we assume we are working in the default namespace, and with the operator installed in the *cnpg-system* namespace. Please adapt to your use case.

30.3.1 Using port forwarding

The simplest way to inspect the metrics is to port-forward the metrics ports of the pods involved.

For example, to inspect the metrics on the `-1` instance of `cluster-example` , we port-forward the 9187 port:

```
kubectl port-forward cluster-example-1 9187:9187
```

With port-forwarding active, the metrics can be inspected easily, for example on a web browser, using HTTP or HTTPS depending on the configuration, with address: `localhost:9187/metrics` .

The operator pod also exports metrics, on port 8080. Similarly to instances, we port-forward the operator pod, which is located in the operator namespace:

```
kubectl -n cnpg-system port-forward pod/<CONTROLLER-MANAGER-POD> 8080:8080
```

With port forwarding active, the metrics are easily viewable on a browser at localhost:8080/metrics .

30.3.2 Using curl

Create the `curl` pod with the following command:

```
kubectl apply -f - <<EOF
- --
apiVersion: v1
kind: Pod
metadata:
  name: curl
spec:
  containers:
```

(continues on next page)

(continued from previous page)

```
  - name: curl
    image: curlimages/curl:8.2.1
    command: [sleep, 3600]
EOF
```

To inspect the metrics exported by an instance, you need to connect to port 9187 of the target pod. You will need to know the pod's IP address, which you can find easily by running `kubectl get pod -o wide`. The following generic command will run `curl` on the desired 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
```

If you enabled TLS metrics, run instead:

```
kubectl exec -ti curl -- curl -sk https://${POD_IP}:9187/metrics
```

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.

When you're done inspecting metrics, please remember to delete the `curl` pod:

```
kubectl delete -f curl.yaml
```

30.4 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.
- `prometheusrule.yaml` : a `PrometheusRule` with alerts for CloudNativePG. NOTE: this does not include inter-operation with notification services. Please refer to the Prometheus documentation .
- `podmonitor.yaml` : a `PodMonitor` for the CloudNativePG Operator deployment.

In addition, we provide the "raw" sources for the Prometheus alert rules in the `alerts.yaml` file.

A Grafana dashboard for CloudNativePG clusters and operator, is kept in the dedicated repository cloudnative-pg/grafana-dashboards

as a dashboard JSON configuration: grafana-dashboard.json . The file can be downloaded, and imported into Grafana (menus: Dashboard > New > Import).

For a general reference on the settings available on `kube-prometheus-stack`, you can execute `helm show values prometheus-community/kube-prometheus-stack`. Please refer to the kube-prometheus-stack

page for more detail.

CHAPTER 31

Logging

CloudNativePG outputs logs in JSON format directly to standard output, including PostgreSQL logs, without persisting them to storage for security reasons. This design facilitates seamless integration with most Kubernetes-compatible log management tools, including command line ones like stern .

Important

Long-term storage and management of logs are outside the scope of the operator and should be handled at the Kubernetes infrastructure level. For more information, see the *Kubernetes Logging Architecture*

documentation.

Each log entry includes the following fields:

- `level` – The log level (e.g., `info` , `notice`).
- `ts` – The timestamp.
- `logger` – The type of log (e.g., `postgres` , `pg_controldata`).
- `msg` – The log message, or the keyword `record` if the message is in JSON format.
- `record` – The actual record, with a structure that varies depending on the `logger` type.
- `logging_pod` – The name of the pod where the log was generated.

Note

If your log ingestion system requires custom field names, you can rename the *level* and *ts* fields using the *log-field-level* and *log-field-timestamp* flags in the operator controller. This can be configured by editing the *Deployment* definition of the *cloudnative-pg* operator.

31.1 Cluster Logs

You can configure the log level for the instance pods in the cluster specification using the `logLevel` option. Available log levels are: `error` , `warning` , `info` (default), `debug` , and `trace` .

Important

Currently, the log level can only be set at the time the instance starts. Changes to the log level in the cluster specification after the cluster has started will only apply to new pods, not existing ones.

31.2 Operator Logs

The logs produced by the operator pod can be configured with log levels, same as instance pods: `error` , `warning` , `info` (default), `debug` , and `trace` .

The log level for the operator can be configured by editing the `Deployment` definition of the operator and setting the `--log-level` command line argument to the desired value.

31.3 PostgreSQL Logs

Each PostgreSQL log entry is a JSON object with the `logger` key set to `postgres` . The structure of the log entries is as follows:


```
{
  "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": "",
```


(continues on next page)

(continued from previous page)

```
    "location": "",
    "application_name": "",
    "backend_type": "startup"
  },
  "logging_pod": "cluster-example-1",
}
```

Note

Internally, the operator uses PostgreSQL's CSV log format. For more details, refer to the *PostgreSQL documentation on CSV log format* .

31.4 PGAudit Logs

CloudNativePG offers seamless and native support for PGAudit on PostgreSQL clusters.

To enable PGAudit, add the necessary `pgaudit` parameters in the `postgresql` section of the cluster configuration.

Important

The PGAudit library must be added to *shared_preload_libraries* . CloudNativePG automatically manages this based on the presence of *pgaudit.** parameters in the PostgreSQL configuration. The operator handles both the addition and removal of the library from *shared_preload_libraries* .

Additionally, the operator manages the creation and removal of the PGAudit extension across all databases within the cluster.

Important

CloudNativePG executes the *CREATE EXTENSION* and *DROP EXTENSION* commands in all databases within the cluster that accept connections.

The following example demonstrates a PostgreSQL `Cluster` deployment with PGAudit enabled and configured:

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

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

(continues on next page)

(continued from previous page)

```
  storage:
    size: 1Gi
```

The audit CSV log entries generated by PGAudit are parsed and routed to standard output in JSON format, similar to all other logs:

- `.logger` is set to `pgaudit` .
- `.msg` is set to `record` .
- `.record` contains the entire parsed record as a JSON object. This structure resembles that of `logging_collector` logs, with the exception of `.record.audit` , which contains the PGAudit CSV message formatted as a JSON object.

This example shows sample log entries:

```
{
  "level": "info",
  "ts": 1627394507.8814096,
  "logger": "pgaudit",
  "msg": "record",
  "record": {
    "log_time": "2021-07-27 14:01:47.881 UTC",
    "user_name": "postgres",
    "database_name": "postgres",
    "process_id": "203",
    "connection_from": "[local]",
    "session_id": "610011cb.cb",
    "session_line_num": "1",
    "command_tag": "SELECT",
    "session_start_time": "2021-07-27 14:01:47 UTC",
    "virtual_transaction_id": "3/336",
    "transaction_id": "0",
    "error_severity": "LOG",
    "sql_state_code": "00000",
    "backend_type": "client backend",
    "audit": {
      "audit_type": "SESSION",
      "statement_id": "1",
      "substatement_id": "1",
      "class": "READ",
      "command": "SELECT FOR KEY SHARE",
      "statement": "SELECT pg_current_wal_lsn()",
      "parameter": "<none>"
    }
  },
  "logging_pod": "cluster-example-1",
}
```

See the PGAudit documentation for more details about each field in a record.

31.5 Other Logs

All logs generated by the operator and its instances are in JSON format, with the `logger` field indicating the process that produced them. The possible `logger` values are as follows:

- `barman-cloud-wal-archive` : logs from `barman-cloud-wal-archive`
- `barman-cloud-wal-restore` : logs from `barman-cloud-wal-restore`
- `initdb` : logs from running `initdb`
- `pg_basebackup` : logs from running `pg_basebackup`
- `pg_controldata` : logs from running `pg_controldata`
- `pg_ctl` : logs from running any `pg_ctl` subcommand
- `pg_rewind` : logs from running `pg_rewind`
- `pgaudit` : logs from the PGAudit extension
- `postgres` : logs from the `postgres` instance (with `msg` distinct from `record`)
- `wal-archive` : logs from the `wal-archive` subcommand of the instance manager
- `wal-restore` : logs from the `wal-restore` subcommand of the instance manager
- `instance-manager` : from the *PostgreSQL instance manager*

With the exception of `postgres` , which follows a specific structure, all other `logger` values contain the `msg` field with the escaped message that is logged.

CHAPTER 32

Certificates

CloudNativePG was designed to natively support TLS certificates. To set up a cluster, the operator requires:

- A server certification authority (CA) certificate
- A server TLS certificate signed by the server CA
- A client CA certificate
- A streaming replication client certificate generated by the client CA

> **Note**
>
> You can find all the secrets used by the cluster and their expiration dates in the cluster's status.

CloudNativePG is very flexible when it comes to TLS certificates. It primarily operates in two modes:

1. *Operator-Managed Mode* – Certificates are internally managed by the operator in a fully automated way and signed using a CA created by CloudNativePG.
2. *User-provided certificates mode* – Certificates are generated outside the operator and imported in the cluster definition as secrets. CloudNativePG integrates itself with cert-manager

(See *Cert-manager example* .)

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

> **Note**
>
> The operator and instances verify server certificates against the CA only, disregarding the DNS name. This approach is due to the typical absence of DNS names in user-provided certificates for the <*cluster*>-*rw* service used for communication within the cluster.

32.1 Operator-Managed Mode

By default, the operator automatically generates a single Certificate Authority (CA) to issue both client and server certificates. These certificates are managed continuously by the operator, with automatic renewal 7 days before expiration (within a 90-day validity period).

> **Note**
>
> You can adjust this default behavior by configuring the *CERTIFICATE_DURATION* and *EXPIRING_CHECK_THRESHOLD* environment variables. For detailed guidance, refer to the *Operator configuration* .

> **Important**
>
> Certificate renewal does not cause any downtime for the PostgreSQL server, as a simple reload operation is sufficient. However, any user-managed certificates not controlled by CloudNativePG must be re-issued following the renewal process.

When generating certificates, the operator assumes that the Kubernetes cluster's DNS zone is set to `cluster.local` by default. This behavior can be customized by setting the `KUBERNETES_CLUSTER_DOMAIN` environment variable. A convenient alternative is to use the *operator's configuration capability* .

32.1.1 Server certificates

32.1.2 Server CA secret

The operator generates a self-signed CA and stores it in a generic secret containing the following keys:

- `ca.crt` – CA certificate used to validate the server certificate, used as `sslrootcert` in clients' connection strings.
- `ca.key` – The key used to sign the server SSL certificate automatically.

32.1.3 Server TLS secret

The operator uses the generated self-signed CA to sign a server TLS certificate. It's stored in a secret of type `kubernetes.io/tls` and configured to be used as `ssl_cert_file` and `ssl_key_file` by the instances. This approach enables clients to verify their identity and connect securely.

32.1.4 Server alternative DNS names

In addition to the default ones, you can specify DNS server alternative names as part of the generated server TLS secret.

32.1.5 Client certificates

32.1.6 Client CA secret

By default, the same self-signed CA as the server CA is used. The public part is passed as `ssl_ca_file` to all the instances so it can verify client certificates it signed. The private key is stored in the same secret and used to sign client certificates generated by the `kubectl cnpg` plugin.

32.1.7 Client `streaming_replica` certificate

The operator uses the generated self-signed CA to sign a client certificate for the user `streaming_replica` , storing it in a secret of type `kubernetes.io/tls` . To allow secure connection to the primary instance, this certificate is passed as `sslcert` and `sslkey` in the replicas' connection strings.

32.2 User-provided certificates mode

32.2.1 Server certificates

If required, you can also provide the two server certificates, generating them using a separate component such as cert-manager . To use a custom server TLS certificate for a cluster, you must specify the following parameters:

- `serverTLSSecret` – The name of a secret of type `kubernetes.io/tls` containing the server TLS certificate. It must contain both the standard `tls.crt` and `tls.key` keys.
- `serverCASecret` – The name of a secret containing the `ca.crt` key.

Note

The operator still creates and manages the two secrets related to client certificates.

Note

The operator and instances verify server certificates against the CA only, disregarding the DNS name. This approach is due to the typical absence of DNS names in user-provided certificates for the *<cluster>-rw* service used for communication within the cluster.

Note

If you want ConfigMaps and secrets to be reloaded by instances, you can add a label with the key *cnpg.io/reload* to it. Otherwise you must reload the instances using the *kubectl cnpg reload* subcommand.

32.2.2 Example

Given the following files:

- `server-ca.crt` – The certificate of the CA that signed the server TLS certificate.
- `server.crt` – The certificate of the server TLS certificate.
- `server.key` – The private key of the server TLS certificate.

Create a secret containing the CA certificate:

```
kubectl create secret generic my-postgresql-server-ca \
  --from-file=ca.crt=./server-ca.crt
```

Create a secret with the TLS certificate:

```
kubectl create secret tls my-postgresql-server \
  --cert=./server.crt --key=./server.key
```

Create a PostgreSQL cluster referencing those secrets:

```
kubectl apply -f - <<EOF
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    serverCASecret: my-postgresql-server-ca
    serverTLSSecret: my-postgresql-server
  storage:
    storageClass: standard
    size: 1Gi
EOF
```

The new cluster uses the provided server certificates for TLS connections.

32.2.3 Cert-manager example

This simple example shows how to use cert-manager

to set up a self-signed CA and generate the needed TLS server certificate:

```
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-server-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-server-cert
spec:
  secretName: my-postgres-server-cert
  usages:
    - server auth
  dnsNames:
    - cluster-example-lb.internal.mydomain.net
    - cluster-example-rw
    - cluster-example-rw.default
    - cluster-example-rw.default.svc
    - cluster-example-r
```

(continues on next page)

(continued from previous page)

```
    - cluster-example-r.default
    - cluster-example-r.default.svc
    - cluster-example-ro
    - cluster-example-ro.default
    - cluster-example-ro.default.svc
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
```

Cert-manager creates a secret named `my-postgres-server-cert` . It contains all the needed files and can be referenced from a cluster as follows:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    serverTLSSecret: my-postgres-server-cert
    serverCASecret: my-postgres-server-cert
  storage:
    size: 1Gi
```

You can find a complete example using cert-manager to manage both server and client CA and certificates in the deployment manifest.

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: server-ca
spec:
  isCA: true
  commonName: my-selfsigned-server-ca
  secretName: server-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
```

(continues on next page)

(continued from previous page)

```
kind: Issuer
metadata:
  name: server-ca-issuer
spec:
  ca:
    secretName: server-ca-key-pair
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-server-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-server-cert
spec:
  secretName: my-postgres-server-cert
  usages:
    - server auth
  dnsNames:
    - cluster-example-lb.internal.mydomain.net
    - cluster-example-rw
    - cluster-example-rw.default
    - cluster-example-rw.default.svc
    - cluster-example-r
    - cluster-example-r.default
    - cluster-example-r.default.svc
    - cluster-example-ro
    - cluster-example-ro.default
    - cluster-example-ro.default.svc
  issuerRef:
    name: server-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: client-ca
spec:
  isCA: true
  commonName: my-selfsigned-client-ca
  secretName: client-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
```

(continues on next page)

(continued from previous page)

```
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: client-ca-issuer
spec:
  ca:
    secretName: client-ca-key-pair
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-client-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-client-cert
spec:
  secretName: my-postgres-client-cert
  usages:
    - client auth
  commonName: streaming_replica
  issuerRef:
    name: client-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    serverTLSSecret: my-postgres-server-cert
    serverCASecret: my-postgres-server-cert
    clientCASecret: my-postgres-client-cert
    replicationTLSSecret: my-postgres-client-cert
  storage:
    size: 1Gi
```

32.2.4 Client certificate

If required, you can also provide the two client certificates, generating them using a separate component such as cert-manager or HashiCorp vault . To use a custom CA to verify client certificates for a cluster, you must specify the following parameters:

- `replicationTLSSecret` – The name of a secret of type `kubernetes.io/tls` containing the client certificate for user `streaming_replica` . It must contain both the standard `tls.crt` and `tls.key` keys.

- `clientCASecret` – The name of a secret containing the `ca.crt` key of the CA to use to verify client certificate.

> **Note**
>
> The operator still creates and manages the two secrets related to server certificates.

> **Note**
>
> As the cluster isn't in control of the client CA secret key, you can no longer generate client certificates using *kubectl cnpg certificate* .

> **Note**
>
> If you want ConfigMaps and secrets to be automatically reloaded by instances, you can add a label with the key *cnpg.io/reload* to it. Otherwise, you must reload the instances using the *kubectl cnpg reload* subcommand.

32.2.5 Customizing the `streaming_replica` client certificate

In some environments, it may not be possible to generate a certificate with the common name `streaming_replica` due to company policies or other security concerns, such as a CA shared across multiple clusters. In such cases, the user mapping feature can be used to allow authentication as the `streaming_replica` user with certificates containing different common names.

To configure this setup, add a `pg_ident.conf` entry for the predefined map named `cnpg_streaming_replica` .

For example, to enable `streaming_replica` authentication using a certificate with the common name `streaming-replica.cnpg.svc.cluster.local` , add the following to your cluster definition:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  postgresql:
    pg_ident:
    - cnpg_streaming_replica streaming-replica.cnpg.svc.cluster.local streaming_
↪replica
```

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

32.2.6 Cert-manager example

This simple example shows how to use cert-manager

to set up a self-signed CA and generate the needed TLS server certificate:

```
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
```

(continues on next page)

(continued from previous page)

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

Cert-manager creates a secret named `my-postgres-client-cert` that contains all the needed files. You can reference it from a cluster as follows:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  certificates:
    clientCASecret: my-postgres-client-cert
    replicationTLSSecret: my-postgres-client-cert
  storage:
    size: 1Gi
```

You can find a complete example using cert-manager to manage both server and client CA and certificates in the deployment manifest.

```
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
- --
apiVersion: cert-manager.io/v1
kind: Certificate
```

(continues on next page)

(continued from previous page)

```
metadata:
  name: server-ca
spec:
  isCA: true
  commonName: my-selfsigned-server-ca
  secretName: server-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: server-ca-issuer
spec:
  ca:
    secretName: server-ca-key-pair
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-server-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-server-cert
spec:
  secretName: my-postgres-server-cert
  usages:
    - server auth
  dnsNames:
    - cluster-example-lb.internal.mydomain.net
    - cluster-example-rw
    - cluster-example-rw.default
    - cluster-example-rw.default.svc
    - cluster-example-r
    - cluster-example-r.default
    - cluster-example-r.default.svc
    - cluster-example-ro
    - cluster-example-ro.default
    - cluster-example-ro.default.svc
  issuerRef:
    name: server-ca-issuer
    kind: Issuer
    group: cert-manager.io
```

(continues on next page)

(continued from previous page)

```
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: client-ca
spec:
  isCA: true
  commonName: my-selfsigned-client-ca
  secretName: client-ca-key-pair
  privateKey:
    algorithm: ECDSA
    size: 256
  issuerRef:
    name: selfsigned-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: client-ca-issuer
spec:
  ca:
    secretName: client-ca-key-pair
- --
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-client-cert
  labels:
    cnpg.io/reload: ""
- --
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: my-postgres-client-cert
spec:
  secretName: my-postgres-client-cert
  usages:
    - client auth
  commonName: streaming_replica
  issuerRef:
    name: client-ca-issuer
    kind: Issuer
    group: cert-manager.io
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
```

(continues on next page)

(continued from previous page)

```
  certificates:
    serverTLSSecret: my-postgres-server-cert
    serverCASecret: my-postgres-server-cert
    clientCASecret: my-postgres-client-cert
    replicationTLSSecret: my-postgres-client-cert
  storage:
    size: 1Gi
```

CHAPTER 33

Client TLS/SSL connections

See also

See *Certificates*

for more details on how CloudNativePG supports TLS certificates.

The CloudNativePG operator was designed to work with TLS/SSL for both encryption in transit and authentication on the server and client sides. Clusters created using the CNPG operator come with a certification authority (CA) to create and sign TLS client certificates. Using the cnpg plugin for kubectl, you can issue a new TLS client certificate for authenticating a user instead of using passwords.

These instructions for authenticating using TLS/SSL certificates assume you installed a cluster using the

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  postgresql:
    pg_hba:
      - hostssl app all all cert

  storage:
    size: 1Gi
```

manifest. According to the convention-over-configuration paradigm, that file creates an `app` database that's owned by a user called app. (You can change this convention by way of the `initdb` configuration in the `bootstrap` section.)

33.1 Issuing a new certificate

> **See also**
>
> See the *Certificates in the CloudNativePG plugin*

content for details on how to use the plugin for kubectl.

You can create a certificate for the app user in the `cluster-example` PostgreSQL cluster as follows:

```
kubectl cnpg certificate cluster-app \
  --cnpg-cluster cluster-example \
  --cnpg-user app
```

You can now verify the certificate:

```
kubectl get secret cluster-app \
  -o jsonpath="{.data[tls\.crt]}" \
  | base64 -d | openssl x509 -text -noout \
  | head -n 11
```

Output:

```
Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      5d:e1:72:8a:39:9f:ce:51:19:9d:21:ff:1e:4b:24:5d
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: OU = default, CN = cluster-example
    Validity
      Not Before: Mar 22 10:22:14 2021 GMT
      Not After : Mar 22 10:22:14 2022 GMT
    Subject: CN = app
```

As you can see, TLS client certificates by default are created with 90 days of validity, and with a simple CN that corresponds to the username in PostgreSQL. You can specify the validity and threshold values using the `EXPIRE_CHECK_THRESHOLD` and `CERTIFICATE_DURATION` parameters. This is necessary to leverage the `cert` authentication method for `hostssl` entries in `pg_hba.conf` .

33.2 Testing the connection via a TLS certificate

Next, test this client certificate by configuring a demo client application that connects to your CloudNativePG cluster.

The following manifest, called `cert-test.yaml` , creates a demo pod with a test application in the same namespace where your database cluster is running:

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cert-test
spec:
  replicas: 1
```

(continues on next page)

(continued from previous page)

```
  selector:
    matchLabels:
      app: webtest
  template:
    metadata:
      labels:
        app: webtest
    spec:
      containers:
        - image: ghcr.io/cloudnative-pg/webtest:1.7.0
          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 mounts secrets managed by the CloudNativePG operator, including:

- `sslcert` – The TLS client public certificate.
- `sslkey` – The TLS client certificate private key.
- `sslrootcert` – The TLS CA certificate that signed the certificate on the server to use to verify the identity of the instances.

They're used to create the default resources that psql (and other libpq-based applications, like pgbench) requires to establish a TLS-encrypted connection to the Postgres database.

By default, psql searches for certificates in the `~/.postgresql` directory of the current user, but you can use the `sslkey`, `sslcert`, and `sslrootcert` options to point libpq to the actual location of the cryptographic material. The content of these files is gathered from the secrets that were previously created by using the cnpg plugin for kubectl.

Deploy the application:

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

Then use the created pod as the PostgreSQL client to validate the SSL connection and authentication using the TLS certificates you just created.

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

You can verify that the connection works. To do so, execute an interactive `bash` inside the pod's container to run psql using the necessary options. The PostgreSQL server is exposed through the read-write Kubernetes service. Point the psql command to connect to this service:

```
kubectl exec -it cert-test -- bash -c "psql
sslkey=/etc/secrets/app/tls.key sslcert=/etc/secrets/app/tls.crt
sslrootcert=/etc/secrets/ca/ca.crt host=cluster-example-rw.default.svc dbname=app
user=app sslmode=verify-full -c select version();"
```

Output:

```
                                          version
- ------------------------------------------------------------------------------------
↪-
- -----------------
PostgreSQL 17.5 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20191121 (Red Hat
8. 3.1-5), 64-bit
(1 row)
```

33.3 About TLS protocol versions

By default, the operator sets both ssl_min_protocol_version

and ssl_max_protocol_version

to `TLSv1.3`.

This assumes that the PostgreSQL operand images include an OpenSSL library that supports the `TLSv1.3` version. If not, or if your client applications need a lower version number, you need to manually configure it in the PostgreSQL configuration as any other Postgres GUC.

CHAPTER 34

Connecting from an application

Applications are supposed to work with the services created by CloudNativePG in the same Kubernetes cluster.

For more information on services and how to manage them, please refer to the *Service Management* section.

Hint

It is highly recommended using those services in your applications, and avoiding connecting directly to a specific PostgreSQL instance, as the latter can change during the cluster lifetime.

You can use these services in your applications through:

- DNS resolution
- environment variables

For the credentials to connect to PostgreSQL, you can use the secrets generated by the operator.

See also

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

34.1 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.

34.2 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

34.3 Secrets

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

- `[cluster name]-app` (unless you have provided an existing secret through `.spec.bootstrap.initdb.secret.name`)
- `[cluster name]-superuser` (if `.spec.enableSuperuserAccess` is set to `true` and you have not specified a different secret using `.spec.superuserSecret`)

Each secret contain the following:

- username
- password
- hostname to the RW service
- port number
- database name
- a working .pgpass file
- uri
- jdbc-uri
- fqdn-uri
- fqdn-jdbc-uri

The FQDN to be used in the URIs is calculated using the Kubernetes cluster domain specified in the `KUBERNETES_CLUSTER_DOMAIN` configuration parameter. See *the operator configuration documentation* for more information about that.

The `-app` credentials are the ones that should be used by applications connecting to the PostgreSQL cluster, and correspond to the user *owning* the database.

The `-superuser` ones are supposed to be used only for administrative purposes, and correspond to the `postgres` user.

Important

Superuser access over the network is disabled by default.

CHAPTER 35

Connection pooling

CloudNativePG provides native support for connection pooling with PgBouncer , one of the most popular open source connection poolers for PostgreSQL, through the `Pooler` custom resource definition (CRD).

In brief, a pooler in CloudNativePG is a deployment of PgBouncer pods that sits between your applications and a PostgreSQL service, for example, the `rw` service. It creates a separate, scalable, configurable, and highly available database access layer.

35.1 Architecture

The following diagram highlights how introducing a database access layer based on PgBouncer changes the architecture of CloudNativePG. Instead of directly connecting to the PostgreSQL primary service, applications can connect to the equivalent service for PgBouncer. This ability enables reuse of existing connections for faster performance and better resource management on the PostgreSQL side.

35.2 Quick start

This example helps to show how CloudNativePG implements a PgBouncer pooler:

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

  instances: 3
  type: rw
  pgbouncer:
    poolMode: session
    parameters:
```

(continues on next page)

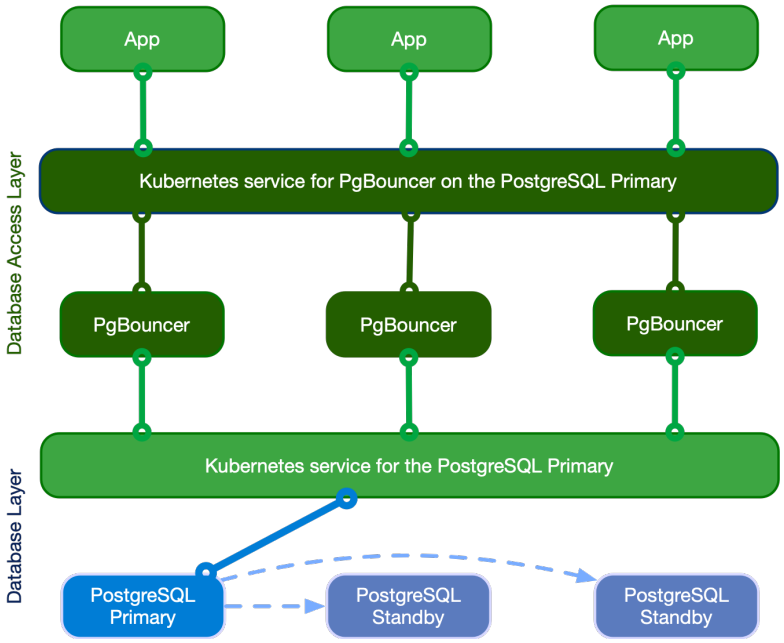


Fig. 1: Applications writing to the single primary via PgBouncer

(continued from previous page)

```
    max_client_conn: "1000"
    default_pool_size: "10"
```

Important

The pooler name can't be the same as any cluster name in the same namespace.

This example creates a `Pooler` resource called `pooler-example-rw` that's strictly associated with the Postgres `Cluster` resource called `cluster-example` . It points to the primary, identified by the read/write service (`rw` , therefore `cluster-example-rw`).

The `Pooler` resource must live in the same namespace as the Postgres cluster. It consists of a Kubernetes deployment of 3 pods running the latest stable image of PgBouncer , configured with the session

and accepting up to 1000 connections each. The default pool size is 10 user/database pairs toward PostgreSQL.

Important

The *Pooler* resource sets only the * fallback database in PgBouncer. This setting means that that all parameters in the connection strings passed from the client are relayed to the PostgreSQL server. For details, see ["Section [databases]" in the PgBouncer documentation](https://www.pgbouncer.org/config.html#section-databases).

CloudNativePG also creates a secret with the same name as the pooler containing the configuration files used with PgBouncer.

See also

For details, see PgBouncerSpec {postgresql-cnpg-io-v1-PgBouncerSpec}

in the API reference.

35.3 Pooler resource lifecycle

`Pooler` resources aren't cluster-managed resources. You create poolers manually when they're needed. You can also deploy multiple poolers per PostgreSQL cluster.

What's important is that the life cycles of the `Cluster` and the `Pooler` resources are currently independent. Deleting the cluster doesn't imply the deletion of the pooler, and vice versa.

Important

Once you know how a pooler works, you have full freedom in terms of possible architectures. You can have clusters without poolers, clusters with a single pooler, or clusters with several poolers, that is, one per application.

Important

When the operator is upgraded, the pooler pods will undergo a rolling upgrade. This is necessary to ensure that the instance manager within the pooler pods is also upgraded.

35.4 Security

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

Specifically, PgBouncer reuses the certificates of the PostgreSQL server. It also uses TLS client certificate authentication to connect to the PostgreSQL server to run the `auth_query` for clients' password authentication (see Using authentication).

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

35.4.1 Certificates

By default, a PgBouncer pooler uses the same certificates that are used by the cluster. However, if you provide those certificates, the pooler accepts secrets with the following formats:

1. Basic Auth
2. TLS
3. Opaque

In the Opaque case, it looks for the following specific keys that need to be used:

- tls.crt
- tls.key

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

35.5 Authentication

Password-based authentication is the only supported method for clients of PgBouncer in CloudNativePG.

Internally, the implementation relies on PgBouncer's `auth_user` and `auth_query` options. Specifically, the operator:

- Creates a standard user called `cnpg_pooler_pgbouncer` in the PostgreSQL server
- Creates the lookup function in the `postgres` database and grants execution privileges to the cnpg_pooler_pgbouncer user (PoLA)
- Issues a TLS certificate for this user
- Sets `cnpg_pooler_pgbouncer` as the `auth_user`
- Configures PgBouncer to use the TLS certificate to authenticate `cnpg_pooler_pgbouncer` against the PostgreSQL server
- Removes all the above when it detects that a cluster doesn't have any pooler associated to it

Important

If you specify your own secrets, the operator doesn't automatically integrate the pooler.

To manually integrate the pooler, if you specified your own secrets, you must run the following queries from inside your cluster.

First, you must create the role:

```
CREATE ROLE cnpg_pooler_pgbouncer WITH LOGIN;
```

Then, for each application database, grant the permission for `cnpg_pooler_pgbouncer` to connect to it:

```
GRANT CONNECT ON DATABASE { database name here } TO cnpg_pooler_pgbouncer;
```

Finally, as a *superuser* connect in each application database, and then create the authentication function inside each of the application databases:

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

REVOKE ALL ON FUNCTION public.user_search(text)
  FROM public;

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

Important

Given that *user_search* is a *SECURITY DEFINER* function, you need to create it through a role with *SUPERUSER* privileges, such as the *postgres* user.

35.6 Pod templates

You can take advantage of pod templates specification in the `template` section of a `Pooler` resource. For details, see PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec} in the API reference.

Using templates, you can configure pods as you like, including fine control over affinity and anti-affinity rules for pods and nodes. By default, containers use images from `ghcr.io/cloudnative-pg/pgbouncer` .

This example shows `Pooler` specifying `PodAntiAffinity` `:

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example
  instances: 3
  type: rw

  template:
    metadata:
      labels:
        app: pooler
    spec:
      containers: []
      affinity:
        podAntiAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
          - labelSelector:
              matchExpressions:
              - key: app
                operator: In
                values:
                - pooler
            topologyKey: "kubernetes.io/hostname"
```

Note

Explicitly set *.spec.template.spec.containers* to *[]* when not modified, as it's a required field for a *PodSpec* . If *.spec.template.spec.containers* isn't set, the Kubernetes api-server returns the following error when trying to apply the manifest:*error validating "pooler.yaml": error validating data: ValidationError(Pooler.spec.template.spec): missing required field "containers"*

This example sets resources and changes the used image:

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

(continues on next page)

(continued from previous page)

```
  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
```

35.7 Service Template

Sometimes, your pooler will require some different labels, annotations, or even change the type of the service, you can achieve that by using the `serviceTemplate` field:

```
apiVersion: postgresql.cnpg.io/v1
kind: Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-example
  instances: 3
  type: rw
  serviceTemplate:
    metadata:
      labels:
        app: pooler
    spec:
      type: LoadBalancer
  pgbouncer:
    poolMode: session
    parameters:
      max_client_conn: "1000"
      default_pool_size: "10"
```

The operator by default adds a `ServicePort` with the following data:

```
ports:
- name: pgbouncer
  port: 5432
  protocol: TCP
```

(continues on next page)

(continued from previous page)

```
  targetPort: pgbouncer
```

Warning

Specifying a *ServicePort* with the name *pgbouncer* or the port *5432* will prevent the default *ServicePort* from being added. This because *ServicePort* entries with the same *name* or *port* are not allowed on Kubernetes and result in errors.

35.8 High availability (HA)

Because of Kubernetes' deployments, you can configure your pooler to run on a single instance or over multiple pods. The exposed service makes sure that your clients are randomly distributed over the available pods running PgBouncer, which then manages and reuses connections toward the underlying server (if using the `rw` service) or servers (if using the `ro` service with multiple replicas).

Warning

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

35.9 PgBouncer configuration options

The operator manages most of the configuration options for PgBouncer , allowing you to modify only a subset of them.

Warning

You are responsible for correctly setting the value of each option, as the operator doesn't validate them.

These are the PgBouncer options you can customize, with links to the PgBouncer documentation for each parameter. Unless stated otherwise, the default values are the ones directly set by PgBouncer.

- application_name_add_host
- autodb_idle_timeout
- cancel_wait_timeout
- client_idle_timeout
- client_login_timeout
- default_pool_size
- disable_pqexec
- dns_max_ttl
- dns_nxdomain_ttl
- idle_transaction_timeout
- ignore_startup_parameters : to be appended to `extra_float_digits,options` - required by CloudNativePG

- listen_backlog
- 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 ClusterMonitoringTLSConfiguration {postgresql-cnpg-io-v1-ClusterMonitoringTLSConfiguration} section below
- max_client_conn
- max_db_connections
- max_packet_size
- max_prepared_statements
- max_user_connections
- min_pool_size
- pkt_buf
- query_timeout
- query_wait_timeout
- reserve_pool_size
- reserve_pool_timeout
- sbuf_loopcnt
- 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
- server_tls_ciphers
- server_tls_protocols
- stats_period
- suspend_timeout
- tcp_defer_accept
- tcp_keepalive
- tcp_keepcnt
- tcp_keepidle

- tcp_keepintvl
- tcp_user_timeout
- tcp_socket_buffer
- track_extra_parameters
- verbose

Customizations of the PgBouncer configuration are written declaratively in the `.spec.pgbouncer.parameters` map.

The operator reacts to the changes in the pooler specification, and every PgBouncer instance reloads the updated configuration without disrupting the service.

Warning

Every PgBouncer pod has the same configuration, aligned with the parameters in the specification. A mistake in these parameters might disrupt the operability of the whole pooler. The operator doesn't validate the value of any option.

35.10 Monitoring

The PgBouncer implementation of the `Pooler` comes with a default Prometheus exporter. It makes available several metrics having the `cnpg_pgbouncer_` prefix by running:

- `SHOW LISTS` (prefix: `cnpg_pgbouncer_lists`)
- `SHOW POOLS` (prefix: `cnpg_pgbouncer_pools`)
- `SHOW STATS` (prefix: `cnpg_pgbouncer_stats`)

Like the CloudNativePG instance, the exporter runs on port `9127` of each pod running PgBouncer and also provides metrics related to the Go runtime (with the prefix `go_*`).

Note

You can inspect the exported metrics on a pod running PgBouncer. For instructions, see *How to inspect the exported metrics* . Make sure that you use the correct IP and the *9127* port.

This example shows the output for `cnpg_pgbouncer` metrics:

```
#  HELP cnpg_pgbouncer_collection_duration_seconds Collection time duration in seconds

#  TYPE cnpg_pgbouncer_collection_duration_seconds gauge

cnpg_pgbouncer_collection_duration_seconds{collector="Collect.up"} 0.002338805

#  HELP cnpg_pgbouncer_collection_errors_total Total errors occurred accessing␣
↪PostgreSQL for metrics.

#  TYPE cnpg_pgbouncer_collection_errors_total counter

cnpg_pgbouncer_collection_errors_total{collector="sql: Scan error on column index 16,␣
```

(continues on next page)

(continued from previous page)

```
↪name \"load_balance_hosts\": converting NULL to int is unsupported"} 5

#  HELP cnpg_pgbouncer_collections_total Total number of times PostgreSQL was␣
↪accessed for metrics.

#  TYPE cnpg_pgbouncer_collections_total counter

cnpg_pgbouncer_collections_total 5

#  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
```

(continues on next page)

(continued from previous page)

```
#  HELP cnpg_pgbouncer_lists_free_servers Count of free servers.

#  TYPE cnpg_pgbouncer_lists_free_servers gauge

cnpg_pgbouncer_lists_free_servers 0

#  HELP cnpg_pgbouncer_lists_login_clients Count of clients in login state.

#  TYPE cnpg_pgbouncer_lists_login_clients gauge

cnpg_pgbouncer_lists_login_clients 0

#  HELP cnpg_pgbouncer_lists_pools Count of pools.

#  TYPE cnpg_pgbouncer_lists_pools gauge

cnpg_pgbouncer_lists_pools 1

#  HELP cnpg_pgbouncer_lists_used_clients Count of used clients.

#  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_active_cancel_req Client connections that have␣
↪forwarded query cancellations to the server and are waiting for the server response.

#  TYPE cnpg_pgbouncer_pools_cl_active_cancel_req gauge

cnpg_pgbouncer_pools_cl_active_cancel_req{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_cl_cancel_req Client connections that have not forwarded␣
↪query cancellations to the server yet.
```

(continues on next page)

(continued from previous page)

```
#  TYPE cnpg_pgbouncer_pools_cl_cancel_req gauge

cnpg_pgbouncer_pools_cl_cancel_req{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_cl_waiting Client connections that have sent queries but␣
↪have not yet got a server connection.

#  TYPE cnpg_pgbouncer_pools_cl_waiting gauge

cnpg_pgbouncer_pools_cl_waiting{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_cl_waiting_cancel_req Client connections that have not␣
↪forwarded query cancellations to the server yet.

#  TYPE cnpg_pgbouncer_pools_cl_waiting_cancel_req gauge

cnpg_pgbouncer_pools_cl_waiting_cancel_req{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_load_balance_hosts Number of hosts not load balancing␣
↪between hosts

#  TYPE cnpg_pgbouncer_pools_load_balance_hosts gauge

cnpg_pgbouncer_pools_load_balance_hosts{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_maxwait How long the first (oldest) client in the queue␣
↪has waited, in seconds. If this starts increasing, then the current pool of servers␣
↪does not handle requests quickly enough. The reason may be either an overloaded␣
↪server or just too small of a pool_size setting.

#  TYPE cnpg_pgbouncer_pools_maxwait gauge

cnpg_pgbouncer_pools_maxwait{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_maxwait_us Microsecond part of the maximum waiting time.

#  TYPE cnpg_pgbouncer_pools_maxwait_us gauge

cnpg_pgbouncer_pools_maxwait_us{database="pgbouncer",user="pgbouncer"} 0

#  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
```

(continues on next page)

(continued from previous page)

```
cnpg_pgbouncer_pools_sv_active{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_sv_active_cancel Server connections that are currently␣
↪forwarding a cancel request

#  TYPE cnpg_pgbouncer_pools_sv_active_cancel gauge

cnpg_pgbouncer_pools_sv_active_cancel{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_sv_idle Server connections that are unused and␣
↪immediately usable for client queries.

#  TYPE cnpg_pgbouncer_pools_sv_idle gauge

cnpg_pgbouncer_pools_sv_idle{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_sv_login Server connections currently in the process of␣
↪logging in.

#  TYPE cnpg_pgbouncer_pools_sv_login gauge

cnpg_pgbouncer_pools_sv_login{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_sv_tested Server connections that are currently running␣
↪either server_reset_query or server_check_query.

#  TYPE cnpg_pgbouncer_pools_sv_tested gauge

cnpg_pgbouncer_pools_sv_tested{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_sv_used Server connections that have been idle for more␣
↪than server_check_delay, so they need server_check_query to run on them before they␣
↪can be used again.

#  TYPE cnpg_pgbouncer_pools_sv_used gauge

cnpg_pgbouncer_pools_sv_used{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_pools_sv_wait_cancels Servers that normally could become idle,␣
↪but are waiting to do so until all in-flight cancel requests have completed that␣
↪were sent to cancel a query on this server.

#  TYPE cnpg_pgbouncer_pools_sv_wait_cancels gauge

cnpg_pgbouncer_pools_sv_wait_cancels{database="pgbouncer",user="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_avg_bind_count Average number of prepared statements␣
↪readied for execution by clients and forwarded to PostgreSQL by pgbouncer.

#  TYPE cnpg_pgbouncer_stats_avg_bind_count gauge

cnpg_pgbouncer_stats_avg_bind_count{database="pgbouncer"} 0
```

(continues on next page)

(continued from previous page)

```
# HELP cnpg_pgbouncer_stats_avg_client_parse_count Average number of prepared␣
↪statements created by clients.

# TYPE cnpg_pgbouncer_stats_avg_client_parse_count gauge

cnpg_pgbouncer_stats_avg_client_parse_count{database="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"} 0

# 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_server_parse_count Average number of prepared␣
↪statements created by pgbouncer on a server.

# TYPE cnpg_pgbouncer_stats_avg_server_parse_count gauge

cnpg_pgbouncer_stats_avg_server_parse_count{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
```

(continues on next page)

(continued from previous page)

```
cnpg_pgbouncer_stats_avg_xact_count{database="pgbouncer"} 0

#  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_bind_count Total number of prepared statements␣
↪readied for execution by clients and forwarded to PostgreSQL by pgbouncer

#  TYPE cnpg_pgbouncer_stats_total_bind_count gauge

cnpg_pgbouncer_stats_total_bind_count{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_total_client_parse_count Total number of prepared␣
↪statements created by clients.

#  TYPE cnpg_pgbouncer_stats_total_client_parse_count gauge

cnpg_pgbouncer_stats_total_client_parse_count{database="pgbouncer"} 0

#  HELP cnpg_pgbouncer_stats_total_query_count Total number of SQL queries pooled by␣
↪pgbouncer.

#  TYPE cnpg_pgbouncer_stats_total_query_count gauge

cnpg_pgbouncer_stats_total_query_count{database="pgbouncer"} 15

#  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
```

(continues on next page)

(continued from previous page)

```
#  HELP cnpg_pgbouncer_stats_total_server_parse_count Total number of prepared␣
↪statements created by pgbouncer on a server.

#  TYPE cnpg_pgbouncer_stats_total_server_parse_count gauge

cnpg_pgbouncer_stats_total_server_parse_count{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"} 15

#  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
```

Note

For a better understanding of the metrics please refer to the PgBouncer documentation.

As for clusters, a specific pooler can be monitored using the Prometheus operator's resource PodMonitor . A `PodMonitor` correctly pointing to a pooler can be created by the operator by setting `.spec.monitoring.enablePodMonitor` to `true` in the `Pooler` resource. The default is `false` .

Important

Any change to *PodMonitor* created automatically is overridden by the operator at the next reconciliation cycle. If you need to customize it, you can do so as shown in the following example.

To deploy a `PodMonitor` for a specific pooler manually, you can define it as follows and change it as needed:

```
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: <POOLER_NAME>
spec:
  selector:
```

(continues on next page)

(continued from previous page)

```
    matchLabels:
      cnpg.io/poolerName: <POOLER_NAME>
  podMetricsEndpoints:
  - port: metrics
```

35.11 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"
  }
}
```

35.12 Pausing connections

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

1. Closes all active connections toward the PostgreSQL server, after waiting for the queries to complete
2. Pauses any new connection coming from the client

When the `paused` option is reset to `false` , the operator invokes the `RESUME` command in PgBouncer, reopening the taps toward the PostgreSQL service defined in the `Pooler` resource.

See also

For more information, see PAUSE .

Important

In future versions, the switchover operation will be fully integrated with the PgBouncer pooler and take advantage of the *PAUSE /RESUME* features to reduce the perceived downtime by client applications. Currently, you can achieve the same results by setting the *paused* attribute to *true* , issuing the switchover command through the Backup {postgresql-cnpg-io-v1-Backup} , and then restoring the *paused* attribute to *false* .

35.13 Limitations

35.13.1 Single PostgreSQL cluster

The current implementation of the pooler is designed to work as part of a specific CloudNativePG cluster (a service). It isn't currently possible to create a pooler that spans multiple clusters.

35.13.2 Controlled configurability

CloudNativePG transparently manages several configuration options that are used for the PgBouncer layer to communicate with PostgreSQL. Such options aren't configurable from outside and include TLS certificates, authentication settings, the `databases` section, and the `users` section. Also, considering the specific use case for the single PostgreSQL cluster, the adopted criteria is to explicitly list the options that can be configured by users.

> **Note**
>
> The adopted solution likely addresses the majority of use cases. It leaves room for the future implementation of a separate operator for PgBouncer to complete the gamma with more advanced and customized scenarios.

CHAPTER 36

Replica clusters

A replica cluster is a CloudNativePG `Cluster` resource designed to replicate data from another PostgreSQL instance, ideally also managed by CloudNativePG.

Typically, a replica cluster is deployed in a different Kubernetes cluster in another region. These clusters can be configured to perform cascading replication and can rely on object stores for data replication from the source, as detailed further down.

There are primarily two use cases for replica clusters:

1. **Disaster Recovery and High Availability**: Enhance disaster recovery and, to some extent, high availability of a CloudNativePG cluster across different Kubernetes clusters, typically located in different regions. In CloudNativePG terms, this is known as a *From Replica Clusters to Distributed Topology* .
2. **Read-Only Workloads**: Create standalone replicas of a PostgreSQL cluster for purposes such as reporting or Online Analytical Processing (OLAP). These replicas are primarily for read-only workloads. In CloudNativePG terms, this is referred to as a *Standalone Replica Clusters* .

For example, the diagram below — taken from the *Architecture*

— illustrates a distributed PostgreSQL topology spanning two Kubernetes clusters, with a symmetric replica cluster primarily serving disaster recovery purposes.

36.1 Basic Concepts

CloudNativePG builds on the PostgreSQL replication framework, allowing you to create and synchronize a PostgreSQL cluster from an existing source cluster using the replica cluster feature — described in this section. The source can be a primary cluster or another replica cluster (cascading replication).

36.1.1 About PostgreSQL Roles

A replica cluster operates in continuous recovery mode, meaning no changes to the database, including the catalog and global objects like roles or databases, are permitted. These changes are deferred until the `Cluster` transitions to primary. During this phase, global objects such as roles remain as defined in the source cluster. CloudNativePG applies any local redefinitions once the cluster is promoted.

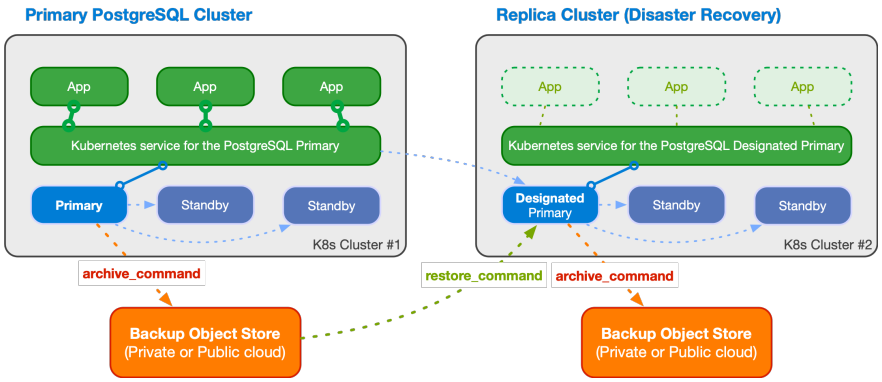


Fig. 1: An example of multi-cluster deployment with a primary and a replica cluster

If you are not planning to promote the cluster (e.g., for read-only workloads) or if you intend to detach completely from the source cluster once the replica cluster is promoted, you don't need to take any action. This is normally the case of the *Standalone Replica Clusters* .

If you are planning to promote the cluster at some point, CloudNativePG will manage the following roles and passwords when transitioning from replica cluster to primary:

- the application user
- the superuser (if you are using it)
- any role defined using the *declarative interface*

If your intention is to seamlessly ensure that the above roles and passwords don't change, you need to define the necessary secrets for the above in each `Cluster` . This is normally the case of the *From Replica Clusters to Distributed Topology* .

36.1.2 Bootstrapping a Replica Cluster

The first step is to bootstrap the replica cluster using one of the following methods:

- **Streaming replication** via `pg_basebackup`
- **Recovery from a volume snapshot**
- **Recovery from a Barman Cloud backup** in an object store

For detailed instructions on cloning a PostgreSQL server using `pg_basebackup` (streaming) or recovery (volume snapshot or object store), refer to the *Bootstrap* .

36.1.3 Configuring Replication

Once the base backup for the replica cluster is available, you need to define how changes will be replicated from the origin using PostgreSQL continuous recovery. There are three main options:

1. **Streaming Replication**: Set up streaming replication between the replica cluster and the source. This method requires configuring network connections and implementing appropriate administrative and security measures to ensure seamless data transfer.
2. **WAL Archive**: Use the WAL (Write-Ahead Logging) archive stored in an object store. WAL files are regularly transferred from the source cluster to the object store, from where a CNPG-I plugin like Barman Cloud

retrieves them for the replica cluster via the `restore_command` .

3. **Hybrid Approach**: Combine both streaming replication and WAL archive methods. PostgreSQL can manage and switch between these two approaches as needed to ensure data consistency and availability.

36.1.4 Defining an External Cluster

When configuring the external cluster, you have the following options:

- **``plugin`` section**:
  - Enables bootstrapping the replica cluster using a CNPG-I

plugin that support the restore_job

and the wal protocols. - CloudNativePG supports the Barman Cloud Plugin

to allow bootstrapping the replica cluster from an object store.

- **``connectionParameters`` section**:
  - Enables bootstrapping the replica cluster via streaming replication using the `pg_basebackup` section.
  - CloudNativePG automatically sets the `primary_conninfo` option in the designated primary instance, initiating a WAL receiver process to connect to the source cluster and receive data.

You still have access to the **``barmanObjectStore`` section**, although deprecated:

- Enables use of the WAL archive, with CloudNativePG automatically setting the `restore_command` in the designated primary instance.
- Allows bootstrapping the replica cluster from an object store using the `recovery` section if volume snapshots are not feasible.

36.1.5 Backup and Symmetric Architectures

The replica cluster can perform backups to a reserved object store from the designated primary, supporting symmetric architectures in a distributed environment. This architectural choice is crucial as it ensures the cluster is prepared for promotion during a controlled data center switchover or a failover following an unexpected event.

36.1.6 Distributed Architecture Flexibility

You have the flexibility to design your preferred distributed architecture for a PostgreSQL database, choosing from:

- **Private Cloud**: Spanning multiple Kubernetes clusters in different data centers.
- **Public Cloud**: Spanning multiple Kubernetes clusters in different regions.
- **Hybrid Cloud**: Combining private and public clouds.
- **Multi-Cloud**: Spanning multiple Kubernetes clusters across different regions and Cloud Service Providers.

36.2 Setting Up a Replica Cluster

To set up a replica cluster from a source cluster, follow these steps to create a cluster YAML file and configure it accordingly:

1. **Define External Clusters**:
   - In the `externalClusters` section, specify the replica cluster.
   - For a distributed PostgreSQL topology aimed at disaster recovery (DR) and high availability (HA), this section should be defined for every PostgreSQL cluster in the distributed database.
2. **Bootstrap the Replica Cluster**:

- **Streaming Bootstrap**: Use the `pg_basebackup` section for bootstrapping via streaming replication.
- **Snapshot/Object Store Bootstrap**: Use the `recovery` section to bootstrap from a volume snapshot or an object store.

3. **Continuous Recovery Strategy**: Define this in the `.spec.replica` stanza:
   - **Distributed Topology**: Configure using the `primary`, `source`, and `self` fields along with the distributed topology defined in `externalClusters`. This allows CloudNativePG to declaratively control the demotion of a primary cluster and the subsequent promotion of a replica cluster using a promotion token.
   - **Standalone Replica Cluster**: Enable continuous recovery using the `enabled` option and set the `source` field to point to an `externalClusters` name. This configuration is suitable for creating replicas primarily intended for read-only workloads.

Both the Distributed Topology and the Standalone Replica Cluster strategies for continuous recovery are thoroughly explained below.

36.3 Distributed Topology

36.3.1 Planning for a Distributed PostgreSQL Database

As Dwight Eisenhower famously said, "Planning is everything", and this holds true for designing PostgreSQL architectures in Kubernetes.

First, conceptualize your distributed topology on paper, and then translate it into a CloudNativePG API configuration. This configuration primarily involves:

- The `externalClusters` section, which must be included in every `Cluster` definition within your distributed PostgreSQL setup.
- The `.spec.replica` stanza, specifically the `primary`, `source`, and (optionally) `self` fields.

For example, suppose you want to deploy a PostgreSQL cluster distributed across two Kubernetes clusters located in Southern Europe and Central Europe.

In this scenario, assume you have CloudNativePG installed in the Southern Europe Kubernetes cluster, with a PostgreSQL `Cluster` named `cluster-eu-south` acting as the primary. This cluster has continuous backup configured with a local object store. This object store is also accessible by the PostgreSQL `Cluster` named `cluster-eu-central`, installed in the Central European Kubernetes cluster. Initially, `cluster-eu-central` functions as a replica cluster. Following a symmetric approach, it also has a local object store for continuous backup, which needs to be read by `cluster-eu-south`.

In this example, recovery is performed solely through WAL shipping, without any streaming replication between the two clusters. However, you can configure the setup to use streaming replication alone or adopt a hybrid approach—streaming replication with WAL shipping as a fallback—as described in the *"Configuring replication"* section.

Here's how you would configure the `externalClusters` section for both `Cluster` resources, relying on Barman Cloud Plugin for the object store:

```
#  Distributed topology configuration

externalClusters:
  - name: cluster-eu-south
    plugin:
      name: barman-cloud.cloudnative-pg.io
      parameters:
```

(continues on next page)

(continued from previous page)

```
        barmanObjectName: cluster-eu-south
        serverName: cluster-eu-south
  - name: cluster-eu-central
    plugin:
      name: barman-cloud.cloudnative-pg.io
      parameters:
        barmanObjectName: cluster-eu-central
        serverName: cluster-eu-central
```

The `.spec.replica` stanza for the `cluster-eu-south` PostgreSQL primary `Cluster` should be configured as follows:

```
replica:
  primary: cluster-eu-south
  source: cluster-eu-central
```

Meanwhile, the `.spec.replica` stanza for the `cluster-eu-central` PostgreSQL replica `Cluster` should be configured as:

```
replica:
  primary: cluster-eu-south
  source: cluster-eu-south
```

In this configuration, when the `primary` field matches the name of the `Cluster` resource (or `.spec.replica.self` if a different one is used), the current cluster is considered the primary in the distributed topology. Otherwise, it is set as a replica from the `source` (in this case, using the Barman object store).

This setup allows you to efficiently manage a distributed PostgreSQL architecture across multiple Kubernetes clusters, ensuring both high availability and disaster recovery through controlled switchover of a primary PostgreSQL cluster using declarative configuration.

Controlled switchover in a distributed topology is a two-step process involving:

- Demotion of a primary cluster to a replica cluster
- Promotion of a replica cluster to a primary cluster

These processes are described in the next sections.

Important

Before you proceed, ensure you review the *About PostgreSQL Roles*

above and use identical role definitions, including secrets, in all `Cluster` objects participating in the distributed topology.

36.3.2 Demoting a Primary to a Replica Cluster

CloudNativePG provides the functionality to demote a primary cluster to a replica cluster. This action is typically planned when transitioning the primary role from one data center to another. The process involves demoting the current primary cluster (e.g., `cluster-eu-south`) to a replica cluster and subsequently promoting the designated replica cluster (e.g., `cluster-eu-central`) to primary when fully synchronized.

Provided you have defined an external cluster in the current primary `Cluster` resource that points to the replica cluster that's been selected to become the new primary, all you need to do is change the `primary` field as follows:

```
replica:
  primary: cluster-eu-central
  source: cluster-eu-central
```

When the primary PostgreSQL cluster is demoted, write operations are no longer possible. CloudNativePG then:

1. Archives the WAL file containing the shutdown checkpoint as a `.partial` file in the WAL archive.
2. Generates a `demotionToken` in the status, a base64-encoded JSON structure containing relevant information from `pg_controldata` such as the system identifier, the timestamp, timeline ID, REDO location, and REDO WAL file of the latest checkpoint.

The first step is necessary to demote/promote using solely the WAL archive to feed the continuous recovery process (without streaming replication).

The second step, generation of the `.status.demotionToken` , will ensure a smooth demotion/promotion process, without any data loss and without rebuilding the former primary.

At this stage, the former primary has transitioned to a replica cluster, awaiting WAL data from the new global primary: `cluster-eu-central` .

To proceed with promoting the other cluster, you need to retrieve the `demotionToken` from `cluster-eu-south` using the following command:

```
kubectl get cluster cluster-eu-south \
  -o jsonpath={.status.demotionToken}
```

You can obtain the `demotionToken` using the `cnpg` plugin by checking the cluster's status. The token is listed under the `Demotion token` section.

> **Note**
>
> The *demotionToken* obtained from *cluster-eu-south* will serve as the *promotionToken* for *cluster-eu-central* .

You can verify the role change using the `cnpg` plugin, checking the status of the cluster:

```
kubectl cnpg status cluster-eu-south
```

36.3.3 Promoting a Replica to a Primary Cluster

To promote a PostgreSQL replica cluster (e.g., `cluster-eu-central`) to a primary cluster and make the designated primary an actual primary instance, you need to perform the following steps simultaneously:

1. Set the `.spec.replica.primary` to the name of the current replica cluster to be promoted (e.g., `cluster-eu-central`).
2. Set the `.spec.replica.promotionToken` with the value obtained from the former primary cluster (refer to *Demoting a Primary to a Replica Cluster*).

The updated `replica` section in `cluster-eu-central` 's spec should look like this:

```
replica:
  primary: cluster-eu-central
  promotionToken: <PROMOTION_TOKEN>
  source: cluster-eu-south
```

> **Warning**
>
> It is crucial to apply the changes to the *primary* and *promotionToken* fields simultaneously. If the promotion token is omitted, a failover will be triggered, necessitating a rebuild of the former primary.

After making these adjustments, CloudNativePG will initiate the promotion of the replica cluster to a primary cluster. Initially, CloudNativePG will wait for the designated primary cluster to replicate all Write-Ahead Logging (WAL) information up to the specified Log Sequence Number (LSN) contained in the token. Once this target is achieved, the promotion process will commence. The new primary cluster will switch timelines, archive the history file and new WAL, thereby unblocking the replication process in the `cluster-eu-south` cluster, which will then operate as a replica.

To verify the role change, use the `cnpg` plugin to check the status of the cluster:

```
kubectl cnpg status cluster-eu-central
```

This command will provide you with the current status of `cluster-eu-central` , confirming its promotion to primary.

By following these steps, you ensure a smooth and controlled promotion process, minimizing disruption and maintaining data integrity across your PostgreSQL clusters.

36.4 Standalone Replica Clusters

> **Important**
>
> Standalone Replica Clusters were previously known as Replica Clusters before the introduction of the Distributed Topology strategy in CloudNativePG 1.24.

In CloudNativePG, a Standalone Replica Cluster is a PostgreSQL cluster in continuous recovery with the following configurations:

- `.spec.replica.enabled` set to `true`
- A physical replication source defined via the `.spec.replica.source` field, pointing to an `externalClusters` name

When `.spec.replica.enabled` is set to `false` , the replica cluster exits continuous recovery mode and becomes a primary cluster, completely detached from the original source.

> **Warning**
>
> Disabling replication is an **irreversible** operation. Once replication is disabled and the designated primary is promoted to primary, the replica cluster and the source cluster become two independent clusters definitively.

> **Important**
>
> Standalone replica clusters are suitable for several use cases, primarily involving read-only workloads. If you are planning to setup a disaster recovery solution, look into "Distributed Topology" above.

36.4.1 Main Differences with Distributed Topology

Although Standalone Replica Clusters can be used for disaster recovery purposes, they differ from the "Distributed Topology" strategy in several key ways:

- **Lack of Distributed Database Concept**: Standalone Replica Clusters do not support the concept of a distributed database, whether in simple forms (two clusters) or more complex configurations (e.g., three clusters in a circular topology).
- **No Global Primary Cluster**: There is no notion of a global primary cluster in Standalone Replica Clusters.
- **No Controlled Switchover**: A Standalone Replica Cluster can only be promoted to primary. The former primary cluster must be re-cloned, as controlled switchover is not possible.

Failover is identical in both strategies, requiring the former primary to be re-cloned if it ever comes back up.

36.4.2 Example of Standalone Replica Cluster using `pg_basebackup`

This **first example** defines a standalone 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.

```
bootstrap:
  pg_basebackup:
    source: cluster-example

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

The previous configuration assumes that the application database and its owning user are set to the default, `app` . If the PostgreSQL cluster being restored uses different names, you must specify them as documented in *Configure the application database* . You should also consider copying over the application user secret from the original cluster and keep it synchronized with the source. See *About PostgreSQL Roles* for more details.

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.

```
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:
```

(continues on next page)

(continued from previous page)

```
    name: <MAIN-CLUSTER>-ca
    key: ca.crt
```

36.4.3 Example of Standalone Replica Cluster from an object store

The **second example** defines a replica cluster that bootstraps from an object store using the `recovery` section and continuous recovery using both streaming replication and the given object store. For streaming replication, the replica cluster connects to the source cluster using basic authentication.

You can check the *sample YAML*

for it in the `samples/` subdirectory.

Note the `bootstrap` and `replica` sections pointing to the source cluster.

```
bootstrap:
  recovery:
    source: cluster-example

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

The previous configuration assumes that the application database and its owning user are set to the default, `app` . If the PostgreSQL cluster being restored uses different names, you must specify them as documented in *Configure the application database* . You should also consider copying over the application user secret from the original cluster and keep it synchronized with the source. See *About PostgreSQL Roles* for more details.

In the `externalClusters` section, take care to use the right namespace in the `endpointURL` and the `connectionParameters.host` . And do ensure that the necessary secrets have been copied if necessary, and that a backup of the source cluster has been created already.

```
externalClusters:
- name: <MAIN-CLUSTER>
  # Example with Barman Cloud Plugin
  plugin:
    name: barman-cloud.cloudnative-pg.io
    parameters:
      barmanObjectName: <MAIN-CLUSTER>
      serverName: <MAIN-CLUSTER>
      …
  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.

36.4.4 Example using a Volume Snapshot

If you use volume snapshots and your storage class provides snapshots cross-cluster availability, you can leverage that to bootstrap a replica cluster through a volume snapshot of the source cluster.

The **third example** defines a replica cluster that bootstraps from a volume snapshot using the `recovery` section. It uses streaming replication (via basic authentication) and the object store to fetch the WAL files.

You can check the *sample YAML*

for it in the `samples/` subdirectory.

The example assumes that the application database and its owning user are set to the default, `app` . If the PostgreSQL cluster being restored uses different names, you must specify them as documented in *Configure theapplication database* . You should also consider copying over the application user secret from the original cluster and keep it synchronized with the source. See *About PostgreSQL Roles* for more details.

36.5 Delayed replicas

CloudNativePG supports the creation of **delayed replicas** through the *.spec.replica.minApplyDelay* , leveraging PostgreSQL's *recovery_min_apply_delay* .

Delayed replicas are designed to intentionally lag behind the primary database by a specified amount of time. This delay is configurable using the `.spec.replica.minApplyDelay` option, which maps to the underlying `recovery_min_apply_delay` parameter in PostgreSQL.

The primary objective of delayed replicas is to mitigate the impact of unintended SQL statement executions on the primary database. This is especially useful in scenarios where operations such as `UPDATE` or `DELETE` are performed without a proper `WHERE` clause.

To configure a delay in a replica cluster, adjust the `.spec.replica.minApplyDelay` option. This parameter determines how much time the replicas will lag behind the primary. For example:

```
# ...
replica:
  enabled: true
  source: cluster-example
  # Enforce a delay of 8 hours
  minApplyDelay: 8h
# ...
```

The above example helps safeguard against accidental data modifications by providing a buffer period of 8 hours to detect and correct issues before they propagate to the replicas.

Monitor and adjust the delay as needed based on your recovery time objectives and the potential impact of unintended primary database operations.

The main use cases of delayed replicas can be summarized into:

1. mitigating human errors: reduce the risk of data corruption or loss resulting from unintentional SQL operations on the primary database
2. recovery time optimization: facilitate quicker recovery from unintended changes by having a delayed replica that allows you to identify and rectify issues before changes are applied to other replicas.

3. enhanced data protection: safeguard critical data by introducing a time buffer that provides an opportunity to intervene and prevent the propagation of undesirable changes.

Warning

The *minApplyDelay* option of delayed replicas cannot be used in conjunction with *promotionToken* .

By integrating delayed replicas into your replication strategy, you can enhance the resilience and data protection capabilities of your PostgreSQL environment. Adjust the delay duration based on your specific needs and the criticality of your data.

Important

Always measure your goals. Depending on your environment, it might be more efficient to rely on volume snapshot-based recovery for faster outcomes. Evaluate and choose the approach that best aligns with your unique requirements and infrastructure.

CHAPTER 37

Kubernetes Upgrade and Maintenance

Maintaining an up-to-date Kubernetes cluster is crucial for ensuring optimal performance and security, particularly for self-managed clusters, especially those running on bare metal infrastructure. Regular updates help address technical debt and mitigate business risks, despite the controlled downtimes associated with temporarily removing a node from the cluster for maintenance purposes. For further insights on embracing risk in operations, refer to the Embracing Risk chapter from the Site Reliability Engineering book.

37.1 Importance of Regular Updates

Updating Kubernetes involves planning and executing maintenance tasks, such as applying security updates to underlying Linux servers, replacing malfunctioning hardware components, or upgrading the cluster to the latest Kubernetes version. These activities are essential for maintaining a robust and secure infrastructure.

37.2 Maintenance Operations in a Cluster

Typically, maintenance operations are carried out on one node at a time, following a structured process :

1. eviction of workloads (`drain`): workloads are gracefully moved away from the node to be updated, ensuring a smooth transition.
2. performing the operation: the actual maintenance operation, such as a system update or hardware replacement, is executed.
3. rejoining the node to the cluster (`uncordon`): the updated node is reintegrated into the cluster, ready to resume its responsibilities.

This process requires either stopping workloads for the entire upgrade duration or migrating them to other nodes in the cluster.

37.3 Temporary PostgreSQL Cluster Degradation

While the standard approach ensures service reliability and leverages Kubernetes' self-healing capabilities, there are scenarios where operating with a temporarily degraded cluster may be acceptable. This is particularly relevant for PostgreSQL clusters relying on **node-local storage**, where the storage is local to the Kubernetes worker node running the PostgreSQL database. Node-local storage, or simply *local storage*, is employed to enhance performance.

> **Note**
>
> If your database files reside on shared storage accessible over the network, the default self-healing behavior of the operator can efficiently handle scenarios where volumes are reused by pods on different nodes after a drain operation. In such cases, you can skip the remaining sections of this document.

37.4 Pod Disruption Budgets

By default, CloudNativePG safeguards Postgres cluster operations. If a node is to be drained and contains a cluster's primary instance, a switchover happens ahead of the drain. Once the instance in the node is downgraded to replica, the draining can resume. For single-instance clusters, a switchover is not possible, so CloudNativePG will prevent draining the node where the instance is housed. Additionally, in clusters with 3 or more instances, CloudNativePG guarantees that only one replica at a time is gracefully shut down during a drain operation.

Each PostgreSQL `Cluster` is equipped with two associated `PodDisruptionBudget` resources - you can easily confirm it with the `kubectl get pdb` command.

Our recommendation is to leave pod disruption budgets enabled for every production Postgres cluster. This can be effortlessly managed by toggling the `.spec.enablePDB` option, as detailed in the *API Reference* .

37.5 PostgreSQL Clusters used for Development or Testing

For PostgreSQL clusters used for development purposes, often consisting of a single instance, it is essential to disable pod disruption budgets. Failure to do so will prevent the node hosting that cluster from being drained.

The following example illustrates how to disable pod disruption budgets for a 1-instance development cluster:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: dev
spec:
  instances: 1
  enablePDB: false

  storage:
    size: 1Gi
```

This configuration ensures smoother maintenance procedures without restrictions on draining the node during development activities.

37.6 Node Maintenance Window

Important

While CloudNativePG will continue supporting the node maintenance window, it is currently recommended to transition to direct control of pod disruption budgets, as explained in the previous section. This section is retained mainly for backward compatibility.

Prior to release 1.23, CloudNativePG had just one declarative mechanism to manage Kubernetes upgrades when dealing with local storage: you had to temporarily put the cluster in **maintenance mode** through the `nodeMaintenanceWindow` option to avoid standard self-healing procedures to kick in, while, for example, enlarging the partition on the physical node or updating the node itself.

Warning

Limit the duration of the maintenance window to the shortest amount of time possible. In this phase, some of the expected behaviors of Kubernetes are either disabled or running with some limitations, including self-healing, rolling updates, and Pod disruption budget.

The `nodeMaintenanceWindow` option of the cluster has two further settings:

`inProgress` : Boolean value that states if the maintenance window for the nodes is currently in progress or not. By default, it is set to `off` . During the maintenance window, the `reusePVC` option below is evaluated by the operator.

`reusePVC` : Boolean value that defines if an existing PVC is reused or not during the maintenance operation. By default, it is set to `on` . When **enabled**, Kubernetes waits for the node to come up again and then reuses the existing PVC; the `PodDisruptionBudget` policy is temporarily removed. When **disabled**, Kubernetes forces the recreation of the Pod on a different node with a new PVC by relying on PostgreSQL's physical streaming replication, then destroys the old PVC together with the Pod. This scenario is generally not recommended unless the database's size is small, and re-cloning the new PostgreSQL instance takes shorter than waiting. This behavior does **not** apply to clusters with only one instance and reusePVC disabled: see section below.

Note

When performing the *kubectl drain* command, you will need to add the *--delete-emptydir-data* option. Don't be afraid: it refers to another volume internally used by the operator - not the PostgreSQL data directory.

Important

PodDisruptionBudget management can be disabled by setting the *.spec.enablePDB* field to *false* . In that case, the operator won't create *PodDisruptionBudgets* and will delete them if they were previously created.

37.6.1 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.

CHAPTER 38

PostgreSQL Upgrades

PostgreSQL upgrades fall into two categories:

- *Minor Version Upgrades* (e.g., from 17.0 to 17.1)
- *Major Version Upgrades* (e.g., from 16.x to 17.0)

38.1 Minor Version Upgrades

PostgreSQL version numbers follow a `major.minor` format. For instance, in version 17.1:

- `17` is the major version
- `1` is the minor version

Minor releases are fully compatible with earlier and later minor releases of the same major version. They include bug fixes and security updates but do not introduce changes to the internal storage format. For example, PostgreSQL 17.1 is compatible with 17.0 and 17.5.

38.1.1 Upgrading a Minor Version in CloudNativePG

To upgrade to a newer minor version, simply update the PostgreSQL container image reference in your cluster definition, either directly or via image catalogs. CloudNativePG will trigger a *rolling update of the cluster* , replacing each instance one by one, starting with the replicas. Once all replicas have been updated, it will perform either a switchover or a restart of the primary to complete the process.

38.2 Major Version Upgrades

Major PostgreSQL releases introduce changes to the internal data storage format, requiring a more structured upgrade process.

CloudNativePG supports three methods for performing major upgrades:

1. *Logical dump/restore* – Blue/green deployment, offline.
2. *Native logical replication* – Blue/green deployment, online.

3. Physical with `pg_upgrade` – In-place upgrade, offline (covered in the *Offline In-Place Major Upgrades of PostgreSQL* below).

Each method has trade-offs in terms of downtime, complexity, and data volume handling. The best approach depends on your upgrade strategy and operational constraints.

Important

We strongly recommend testing all methods in a controlled environment before proceeding with a production upgrade.

38.3 Offline In-Place Major Upgrades

CloudNativePG performs an **offline in-place major upgrade** when a new operand container image with a higher PostgreSQL major version is declaratively requested for a cluster.

Important

Major upgrades are only supported between images based on the same operating system distribution. For example, if your previous version uses a *bullseye* image, you cannot upgrade to a *bookworm* image.

Warning

There is a bug in PostgreSQL 17.0 through 17.5 that prevents successful upgrades if the *max_slot_wal_keep_size* parameter is set to any value other than *-1* . The upgrade process will fail with an error related to replication slot configuration. This issue has been fixed in PostgreSQL 17.6 and 18beta2 or later versions . If you are using PostgreSQL 17.0 through 17.5, ensure that you upgrade to at least PostgreSQL 17.6 before attempting a major upgrade, or make sure to temporarily set the *max_slot_wal_keep_size* parameter to *-1* in your cluster configuration.

You can trigger the upgrade in one of two ways:

- By updating the major version in the image tag via the `.spec.imageName` option.
- Using an *Image Catalog* to manage version changes.

For details on supported image tags, see *Image Tag Requirements* .

Warning

CloudNativePG is not responsible for PostgreSQL extensions. You must ensure that extensions in the source PostgreSQL image are compatible with those in the target image and that upgrade paths are supported. Thoroughly test the upgrade process in advance to avoid unexpected issues. The *extensions management feature*

can help manage extension upgrades declaratively.

38.3.1 Upgrade Process

1. Shuts down all cluster pods to ensure data consistency.
2. Records the previous PostgreSQL version and image in the cluster's status under `.status.pgDataImageInfo`.
3. Initiates a new upgrade job, which:
   - Verifies that the binaries in the image and the data files align with a major upgrade request.
   - Creates new directories for `PGDATA`, and where applicable, WAL files and tablespaces.
   - Performs the upgrade using `pg_upgrade` with the `--link` option.
   - Upon successful completion, replaces the original directories with their upgraded counterparts.

Warning

During the upgrade process, the entire PostgreSQL cluster, including replicas, is unavailable to applications. Ensure that your system can tolerate this downtime before proceeding.

Warning

Performing an in-place upgrade is an exceptional operation that carries inherent risks. It is strongly recommended to take a full backup of the cluster before initiating the upgrade process.

Note

For detailed guidance on *pg_upgrade*, refer to the official PostgreSQL documentation.

38.3.2 Post-Upgrade Actions

If the upgrade is successful, CloudNativePG:

- Destroys the PVCs of replicas (if available).
- Scales up replicas as required.

Warning

Re-cloning replicas can be time-consuming, especially for very large databases. Plan accordingly to accommodate potential delays. After completing the upgrade, it is strongly recommended to take a full backup. Existing backup data (namely base backups and WAL files) is only available for the previous minor PostgreSQL release.

Warning

pg_upgrade doesn't transfer optimizer statistics. After the upgrade, you may want to run *ANALYZE* on your databases to update them.

If the upgrade fails, you must manually revert the major version change in the cluster's configuration and delete the upgrade job, as CloudNativePG cannot automatically decide the rollback.

Important

This process **protects your existing database from data loss**, as no data is modified during the upgrade. If the upgrade fails, a rollback is usually possible, without having to perform a full recovery from a backup. Ensure you monitor the process closely and take corrective action if needed.

38.3.3 Example: Performing a Major Upgrade

Consider the following PostgreSQL cluster running version 16:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:16-minimal-bookworm
  instances: 3
  storage:
    size: 1Gi
```

You can check the current PostgreSQL version using the following command:

```
kubectl cnpg psql cluster-example -- -qAt -c SELECT version()
```

This will return output similar to:

```
PostgreSQL 16.x ...
```

To upgrade the cluster to version 17, update the `imageName` field by changing the major version tag from `16` to `17` :

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17-minimal-bookworm
  instances: 3
  storage:
    size: 1Gi
```

38.3.4 Upgrade Process

1. Cluster shutdown – All cluster pods are terminated to ensure a consistent upgrade.
2. Upgrade job execution – A new job is created with the name of the primary pod, appended with the suffix `-major-upgrade` . This job runs `pg_upgrade` on the primary's persistent volume group.
3. Post-upgrade steps:

- The PVC groups of the replicas (`cluster-example-2` and `cluster-example-3`) are removed.
- The primary pod is restarted.

- Two new replicas (`cluster-example-4` and `cluster-example-5`) are re-cloned from the upgraded primary.

Once the upgrade is complete, you can verify the new major version by running the same command:

```
kubectl cnpg psql cluster-example -- -qAt -c SELECT version()
```

This should now return output similar to:

```
PostgreSQL 17.x ...
```

You can now update the statistics by running `ANALYZE` on the `app` database:

```
kubectl cnpg psql cluster-example -- app -c ANALYZE
```

CHAPTER 39

Kubectl Plugin

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

39.1 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.

39.1.1 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
```

39.1.2 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.

39.1.3 Debian packages

For example, let's install the 1.27.0 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/v1.27.0/
↪kubectl-cnpg_1.27.0_linux_x86_64.deb \
  --output-document kube-plugin.deb
```

Then, with superuser privileges, install from the local file using `dpkg` :

```
$ sudo dpkg -i kube-plugin.deb
Selecting previously unselected package cnpg.
(Reading database ... 6688 files and directories currently installed.)
Preparing to unpack kube-plugin.deb ...
Unpacking cnpg (1.27.0) ...
Setting up cnpg (1.27.0) ...
```

39.1.4 RPM packages

As in the example for `.rpm` packages, let's install the 1.27.0 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/v1.27.0/
↪kubectl-cnpg_1.27.0_linux_x86_64.rpm \
  --output kube-plugin.rpm
```

Then, with superuser privileges, install with `yum` , and you're ready to use:

```
$ sudo yum --disablerepo=* localinstall kube-plugin.rpm
Failed to set locale, defaulting to C.UTF-8
Dependencies resolved.
================================================================================
 Package            Architecture         Version                   Repository
↪         Size
================================================================================
Installing:
 cnpg               x86_64               1.27.0                  @commandline
↪       20 M

Transaction Summary
================================================================================
Install  1 Package

Total size: 20 M
Installed size: 78 M
Is this ok [y/N]: y
```

39.1.5 Using the Arch Linux User Repository (AUR) Package

To install the plugin from the AUR , follow these steps:

```
git clone https://aur.archlinux.org/kubectl-cnpg.git
cd kubectl-cnpg
makepkg -si
```

Or use your favorite AUR-helper, for example paru :

```
paru -S kubectl-cnpg
```

39.1.6 Using Krew

If you already have Krew installed, you can simply run:

```
kubectl krew install cnpg
```

When a new version of the plugin is released, you can update the existing installation with:

```
kubectl krew update
kubectl krew upgrade cnpg
```

39.1.7 Using Homebrew

> **Note**
>
> Please note that the Homebrew community manages the availability of the kubectl-cnpg plugin on Homebrew .

If you already have Homebrew installed, you can simply run:

```
brew install kubectl-cnpg
```

When a new version of the plugin is released, you can update the existing installation with:

```
brew update
brew upgrade kubectl-cnpg
```

> **Note**
>
> Auto-completion for the kubectl plugin is already managed by Homebrew. There's no need to create the *kubectl_complete-cnpg* script mentioned below.

39.1.8 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

39.1.9 Configuring auto-completion

To configure auto-completion for the plugin, a helper shell script needs to be installed into your current PATH. Assuming the latter contains `/usr/local/bin`, this can be done with the following commands:

```
cat > kubectl_complete-cnpg <<EOF
# !/usr/bin/env sh

#  Call the __complete command passing it all arguments

kubectl cnpg __complete "\$@"
EOF

chmod +x kubectl_complete-cnpg

#  Important: the following command may require superuser permission

sudo mv kubectl_complete-cnpg /usr/local/bin
```

Important

The name of the script needs to be exactly the one provided since it's used by the kubectl auto-complete process

39.2 Use

Once the plugin is installed and deployed, you can start using it like this:

```
kubectl cnpg COMMAND [ARGS...]
```

Note

The plugin automatically detects if the standard output channel is connected to a terminal. In such cases, it may add ANSI colors to the command output. To disable colors, use the *--color=never* option with the command.

39.2.1 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`: specifies the namespace in which to install the operator (default: `cnpg-system`).
- `--control-plane`: if set to true, the operator deployment will include a toleration and affinity for `node-role.kubernetes.io/control-plane`.
- `--replicas`: sets the number of replicas in the deployment.
- `--watch-namespace`: specifies a comma-separated list of namespaces to watch (default: all namespaces).

- `--version` : defines the minor version of the operator to be installed, such as `1.23` . If a minor version is specified, the plugin installs the latest patch version of that minor version. If no version is supplied, the plugin installs the latest `MAJOR.MINOR.PATCH` version of the operator.

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 1.23 \
  --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 1.23` install the latest patch version for minor version 1.23
- `--replicas 3` install the operator with 3 replicas
- `--watch-namespace "albert, bb, freddie"` have the operator watch for changes in the `albert` , `bb` and `freddie` namespaces only

39.2.2 Status

The `status` command provides an overview of the current status of your cluster, including:

- **general information**: name of the cluster, PostgreSQL's system ID, number of instances, current timeline and position in the WAL
- **backup**: point of recoverability, and WAL archiving status as returned by the `pg_stat_archiver` view from the primary - or designated primary in the case of a replica cluster
- **streaming replication**: information taken directly from the `pg_stat_replication` view on the primary instance
- **instances**: information about each Postgres instance, taken directly by each instance manager; in the case of a standby, the `Current LSN` field corresponds to the latest write-ahead log location that has been replayed during recovery (replay LSN).

Important

The status information above is taken at different times and at different locations, resulting in slightly inconsistent returned values. For example, the *Current Write LSN* location in the main header, might be different from the *Current LSN* field in the instances status as it is taken at two different time intervals.

```
kubectl cnpg status sandbox
```

```
Cluster Summary
Name:                default/sandbox
System ID:           7423474350493388827
PostgreSQL Image:    ghcr.io/cloudnative-pg/postgresql:16.4
Primary instance:    sandbox-1
Primary start time:  2024-10-08 18:31:57 +0000 UTC (uptime 1m14s)
```

(continues on next page)

(continued from previous page)

```
Status:              Cluster in healthy state
Instances:           3
Ready instances:     3
Size:                126M
Current Write LSN:   0/604DE38 (Timeline: 1 - WAL File: 000000010000000000000006)

Continuous Backup status
Not configured

Streaming Replication status
Replication Slots Enabled
Name       Sent LSN   Write LSN  Flush LSN  Replay LSN  Write Lag  Flush Lag  Replay␣
↪Lag  State      Sync State  Sync Priority  Replication Slot
- ---       --------   ---------  ---------  ----------  ---------  ---------  -------
↪---  -----      ----------  -------------  ----------------
sandbox-2  0/604DE38  0/604DE38  0/604DE38  0/604DE38   00:00:00   00:00:00  ␣
↪00:00:00    streaming  async       0              active
sandbox-3  0/604DE38  0/604DE38  0/604DE38  0/604DE38   00:00:00   00:00:00  ␣
↪00:00:00    streaming  async       0              active

Instances status
Name       Current LSN  Replication role  Status  QoS         Manager Version  Node
- ---       -----------  ----------------  ------  ---         ---------------  ----
sandbox-1  0/604DE38    Primary           OK      BestEffort  1.27.0           k8s-eu-
↪worker
sandbox-2  0/604DE38    Standby (async)   OK      BestEffort  1.27.0           k8s-eu-
↪worker2
sandbox-3  0/604DE38    Standby (async)   OK      BestEffort  1.27.0           k8s-eu-
↪worker
```

If you require more detailed status information, use the `--verbose` option (or `-v` for short). The level of detail increases each time the flag is repeated:

```
kubectl cnpg status sandbox --verbose
```

```
Cluster Summary
Name:                default/sandbox
System ID:           7423474350493388827
PostgreSQL Image:    ghcr.io/cloudnative-pg/postgresql:16.4
Primary instance:    sandbox-1
Primary start time:  2024-10-08 18:31:57 +0000 UTC (uptime 2m4s)
Status:              Cluster in healthy state
Instances:           3
Ready instances:     3
Size:                126M
Current Write LSN:   0/6053720 (Timeline: 1 - WAL File: 000000010000000000000006)

Continuous Backup status
Not configured

Physical backups
No running physical backups found
```

(continues on next page)

(continued from previous page)

```
Streaming Replication status
Replication Slots Enabled
Name       Sent LSN   Write LSN  Flush LSN  Replay LSN  Write Lag  Flush Lag  Replay␣
↪Lag  State      Sync State  Sync Priority  Replication Slot  Slot Restart LSN  Slot␣
↪WAL Status  Slot Safe WAL Size
- ---       --------   ---------  ---------  ----------  ---------  ---------  -------
↪---  -----      ----------  -------------  ----------------  -----------------  -----
↪----------  ------------------
sandbox-2  0/6053720  0/6053720  0/6053720  0/6053720   00:00:00   00:00:00  ␣
↪00:00:00    streaming  async       0              active            0/6053720      ␣
↪  reserved         NULL
sandbox-3  0/6053720  0/6053720  0/6053720  0/6053720   00:00:00   00:00:00  ␣
↪00:00:00    streaming  async       0              active            0/6053720      ␣
↪  reserved         NULL

Unmanaged Replication Slot Status
No unmanaged replication slots found

Managed roles status
No roles managed

Tablespaces status
No managed tablespaces

Pod Disruption Budgets status
Name             Role     Expected Pods  Current Healthy  Minimum Desired Healthy ␣
↪Disruptions Allowed
- ---             ----     -------------  ---------------  ------------------------  --
↪-----------------
sandbox          replica  2              2                1                         1
sandbox-primary  primary  1              1                1                         0

Instances status
Name       Current LSN  Replication role  Status  QoS         Manager Version  Node
- ---       -----------  ----------------  ------  ---         ---------------  ----
sandbox-1  0/6053720    Primary           OK      BestEffort  1.27.0           k8s-eu-
↪worker
sandbox-2  0/6053720    Standby (async)   OK      BestEffort  1.27.0           k8s-eu-
↪worker2
sandbox-3  0/6053720    Standby (async)   OK      BestEffort  1.27.0           k8s-eu-
↪worker
```

With an additional `-v` (e.g. `kubectl cnpg status sandbox -v -v`), you can also view PostgreSQL configuration, HBA settings, and certificates.

The command also supports output in `yaml` and `json` format.

39.2.3 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 CLUSTER-INSTANCE
```

Or you can use the instance node number to promote:

```
kubectl cnpg promote CLUSTER INSTANCE
```

39.2.4 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 --cnpg-user USER
```

After the secret it's created, you can get it using `kubectl` :

```
kubectl get secret cluster-cert
```

And the content of the same in plain text using the following commands:

```
kubectl get secret cluster-cert -o json | jq -r .data | map(@base64d) | .[]
```

39.2.5 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 CLUSTER

#  this command will restart a single instance, according to the policy above

kubectl cnpg restart CLUSTER INSTANCE
```

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.

39.2.6 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
```

39.2.7 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
```

39.2.8 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` .

39.2.9 report Operator

The `operator` sub-command requests the operator to provide information regarding the operator deployment, configuration and events.

Important

All confidential information in Secrets and ConfigMaps is REDACTED. The Data map will show the **keys** but the values will be empty. The flag *-S* / *--stopRedaction* will defeat the redaction and show the values. Use only at your own risk, this will share private data.

Note

By default, operator logs are not collected, but you can enable operator log collection with the *--logs* flag

- **deployment information**: the operator Deployment and operator Pod
- **configuration**: the Secrets and ConfigMaps in the operator namespace
- **events**: the Events in the operator namespace
- **webhook configuration**: the mutating and validating webhook configurations
- **webhook service**: the webhook service
- **logs**: logs for the operator Pod (optional, off by default) in JSON-lines format

The command will generate a ZIP file containing various manifest in YAML format (by default, but settable to JSON with the `-o` flag). Use the `-f` flag to name a result file explicitly. If the `-f` flag is not used, a default time-stamped filename is created for the zip file.

Note

The report plugin obeys *kubectl* conventions, and will look for objects constrained by namespace. The CNPG Operator will generally not be installed in the same namespace as the clusters. E.g. the default installation namespace is cnpg-system

```
kubectl cnpg report operator -n cnpg-system
```

results in

```
Successfully written report to "report_operator_<TIMESTAMP>.zip" (format: "yaml")
```

With the `-f` flag set:

```
kubectl cnpg report operator -n cnpg-system -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.
```

(continues on next page)

(continued from previous page)

```
↪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(secret).yaml
  inflating: report_operator_<TIMESTAMP>/manifests/cnpg-webhook-cert(secret).yaml
```

If you activated the `--logs` option, you'd see an extra subdirectory:

```
Archive:  report_operator_<TIMESTAMP>.zip
  <snipped …>
  creating: report_operator_<TIMESTAMP>/operator-logs/
  inflating: report_operator_<TIMESTAMP>/operator-logs/cnpg-controller-manager-
↪66fb98dbc5-pxkmh-logs.jsonl
```

Note

The plugin will try to get the PREVIOUS operator's logs, which is helpful when investigating restarted operators. In all cases, it will also try to get the CURRENT operator logs. If current and previous logs are available, it will show them both.

```
====== Beginning of Previous Log =====
2023-03-28T12:56:41.251711811Z {"level":"info","ts":"2023-03-28T12:56:41Z","logger":
↪"setup","msg":"Starting CloudNativePG Operator","version":"1.27.0","build":{"Version
↪":"1.27.0+dev107","Commit":"cc9bab17","Date":"2023-03-28"}}
2023-03-28T12:56:41.251851909Z {"level":"info","ts":"2023-03-28T12:56:41Z","logger":
↪"setup","msg":"Starting pprof HTTP server","addr":"0.0.0.0:6060"}
  <snipped …>

====== End of Previous Log =====
2023-03-28T12:57:09.854306024Z {"level":"info","ts":"2023-03-28T12:57:09Z","logger":
↪"setup","msg":"Starting CloudNativePG Operator","version":"1.27.0","build":{"Version
↪":"1.27.0+dev107","Commit":"cc9bab17","Date":"2023-03-28"}}
2023-03-28T12:57:09.854363943Z {"level":"info","ts":"2023-03-28T12:57:09Z","logger":
↪"setup","msg":"Starting pprof HTTP server","addr":"0.0.0.0:6060"}
```

If the operator hasn't been restarted, you'll still see the `====== Begin …` and `====== End …` guards, with no content inside.

You can verify that the confidential information is REDACTED by default:

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

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

(continues on next page)

(continued from previous page)

```
    fieldsV1:
```

With the `-S` (`--stopRedaction`) option activated, secrets are shown:

```
kubectl cnpg report operator -n cnpg-system -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\(secret\).yaml
```

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

39.2.10 report Cluster

The `cluster` sub-command gathers the following:

- **cluster resources**: the cluster information, same as `kubectl get cluster -o yaml`
- **cluster pods**: pods in the cluster namespace matching the cluster name
- **cluster jobs**: jobs, if any, in the cluster namespace matching the cluster name
- **events**: events in the cluster namespace
- **pod logs**: logs for the cluster Pods (optional, off by default) in JSON-lines format
- **job logs**: logs for the Pods created by jobs (optional, off by default) in JSON-lines format

The `cluster` sub-command accepts the `-f` and `-o` flags, as the `operator` does. If the `-f` flag is not used, a default timestamped report name will be used. Note that the cluster information does not contain configuration Secrets / ConfigMaps, so the `-S` is disabled.

> **Note**
>
> By default, cluster logs are not collected, but you can enable cluster log collection with the *--logs* flag

Usage:

```
kubectl cnpg report cluster CLUSTER [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 CLUSTER -f report.zip [-n 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 CLUSTER [-n 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
```

39.2.11 Logs

The `kubectl cnpg logs` command allows to follow the logs of a collection of pods related to CloudNativePG in a single go.

It has at the moment one available sub-command: `cluster` .

39.2.12 Cluster logs

The `cluster` sub-command gathers all the pod logs for a cluster in a single stream or file. This means that you can get all the pod logs in a single terminal window, with a single invocation of the command.

As in all the cnpg plugin sub-commands, you can get instructions and help with the `-h` flag:

```
kubectl cnpg logs cluster -h
```

The `logs` command will display logs in JSON-lines format, unless the `--timestamps` flag is used, in which case, a human-readable timestamp will be prepended to each line. In this case, lines will no longer be valid JSON, and tools such as `jq` may not work as desired.

If the `logs cluster` sub-command is given the `-f` flag (aka `--follow`), it will follow the cluster pod logs, and will also watch for any new pods created in the cluster after the command has been invoked. Any new pods found, including pods that have been restarted or re-created, will also have their pods followed. The logs will be displayed in the terminal's standard-out. This command will only exit when the cluster has no more pods left, or when it is interrupted by the user.

If `logs` is called without the `-f` option, it will read the logs from all cluster pods until the time of invocation and display them in the terminal's standard-out, then exit. The `-o` or `--output` flag can be provided, to specify the name of the file where the logs should be saved, instead of displaying over standard-out. The `--tail` flag can be used to specify how many log lines will be retrieved from each pod in the cluster. By default, the `logs cluster` sub-command will display all the logs from each pod in the cluster. If combined with the "follow" flag `-f` , the number of logs specified by `--tail` will be retrieved until the current time, and from then the new logs will be followed.

NOTE: unlike other `cnpg` plugin commands, the `-f` is used to denote "follow" rather than specify a file. This keeps with the convention of `kubectl logs` , which takes `-f` to mean the logs should be followed.

Usage:

```
kubectl cnpg logs cluster CLUSTER [flags]
```

Using the `-f` option to follow:

```
kubectl cnpg report cluster CLUSTER -f
```

Using `--tail` option to display 3 lines from each pod and the `-f` option to follow:

```
kubectl cnpg report cluster CLUSTER -f --tail 3
```

```
{"level":"info","ts":"2023-06-30T13:37:33Z","logger":"postgres","msg":"2023-06-30␣
↪13:37:33.142 UTC [26] LOG:  ending log output to stderr","source":"/controller/log/
↪postgres","logging_pod":"cluster-example-3"}
{"level":"info","ts":"2023-06-30T13:37:33Z","logger":"postgres","msg":"2023-06-30␣
↪13:37:33.142 UTC [26] HINT:  Future log output will go to log destination \"csvlog\
↪".","source":"/controller/log/postgres","logging_pod":"cluster-example-3"}
…
…
```

With the `-o` option omitted, and with `--output` specified:

```
$ kubectl cnpg logs cluster CLUSTER --output my-cluster.log

Successfully written logs to "my-cluster.log"
```

39.2.13 Pretty

The `pretty` sub-command reads a log stream from standard input, formats it into a human-readable output, and attempts to sort the entries by timestamp.

It can be used in combination with `kubectl cnpg logs cluster` , as shown in the following example:

```
$ kubectl cnpg logs cluster cluster-example | kubectl cnpg logs pretty
2024-10-15T17:35:00.336 INFO     cluster-example-1 instance-manager Starting␣
↪CloudNativePG Instance Manager
2024-10-15T17:35:00.336 INFO     cluster-example-1 instance-manager Checking for free␣
↪disk space for WALs before starting PostgreSQL
2024-10-15T17:35:00.347 INFO     cluster-example-1 instance-manager starting␣
↪tablespace manager
2024-10-15T17:35:00.347 INFO     cluster-example-1 instance-manager starting external␣
↪server manager
[...]
```

Alternatively, it can be used in combination with other commands that produce CNPG logs in JSON format, such as `stern` , or `kubectl logs` , as in the following example:

```
$ kubectl logs cluster-example-1 | kubectl cnpg logs pretty
2024-10-15T17:35:00.336 INFO     cluster-example-1 instance-manager Starting␣
↪CloudNativePG Instance Manager
2024-10-15T17:35:00.336 INFO     cluster-example-1 instance-manager Checking for free␣
↪disk space for WALs before starting PostgreSQL
2024-10-15T17:35:00.347 INFO     cluster-example-1 instance-manager starting␣
↪tablespace manager
2024-10-15T17:35:00.347 INFO     cluster-example-1 instance-manager starting external␣
↪server manager
[...]
```

The `pretty` sub-command also supports advanced log filtering, allowing users to display logs for specific pods or loggers, or to filter logs by severity level. Here's an example:

```
$ kubectl cnpg logs cluster cluster-example | kubectl cnpg logs pretty --pods cluster-
↪example-1 --loggers postgres --log-level info
2024-10-15T17:35:00.509 INFO     cluster-example-1 postgres         2024-10-15␣
↪17:35:00.509 UTC [29] LOG:  redirecting log output to logging collector process
2024-10-15T17:35:00.509 INFO     cluster-example-1 postgres         2024-10-15␣
↪17:35:00.509 UTC [29] HINT:  Future log output will appear in directory "/
↪controller/log"...
2024-10-15T17:35:00.510 INFO     cluster-example-1 postgres         2024-10-15␣
↪17:35:00.509 UTC [29] LOG:  ending log output to stderr
2024-10-15T17:35:00.510 INFO     cluster-example-1 postgres         ending log output␣
↪to stderr
[...]
```

The `pretty` sub-command will try to sort the log stream, to make logs easier to reason about. In order to achieve this, it gathers the logs into groups, and within groups it sorts by timestamp. This is the only way to sort interactively, as `pretty` may be piped from a command in "follow" mode. The sub-command will add a group separator line, `---` , at the end of each sorted group. The size of the grouping can be configured via the `--sorting-group-size` flag (default: 1000), as illustrated in the following example:

```
$ kubectl cnpg logs cluster cluster-example | kubectl cnpg logs pretty --sorting-
↪group-size=3
2024-10-15T17:35:20.426 INFO     cluster-example-2 instance-manager Starting␣
↪CloudNativePG Instance Manager
2024-10-15T17:35:20.426 INFO     cluster-example-2 instance-manager Checking for free␣
↪disk space for WALs before starting PostgreSQL
```

(continues on next page)

(continued from previous page)

```
2024-10-15T17:35:20.438 INFO     cluster-example-2 instance-manager starting␣
↪tablespace manager
- --
2024-10-15T17:35:20.438 INFO     cluster-example-2 instance-manager starting external␣
↪server manager
2024-10-15T17:35:20.438 INFO     cluster-example-2 instance-manager starting␣
↪controller-runtime manager
2024-10-15T17:35:20.439 INFO     cluster-example-2 instance-manager Starting␣
↪EventSource
- --
[...]
```

To explore all available options, use the `-h` flag for detailed explanations of the supported flags and their usage.

Note

You can also increase the verbosity of the log by adding more *-v* options.

39.2.14 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 INSTANCE
```

The following example removes the `cluster-example-2` pod and the associated PVCs:

```
kubectl cnpg destroy cluster-example 2
```

39.2.15 Cluster Hibernation

There are times when you may need to temporarily suspend a CloudNativePG `Cluster` while preserving its data, allowing you to resume operations later. This feature is known as **cluster hibernation**.

Hibernation is managed declaratively using the `cnpg.io/hibernation` annotation.

Note

For more details, see the *Declarative hibernation*

documentation page.

To simplify the process, the `cnpg` plugin for `kubectl` provides a `hibernate` command, which acts as a convenient shortcut for applying the annotation.

To hibernate a cluster, run:

```
kubectl cnpg hibernate on CLUSTER
```

This command applies the `cnpg.io/hibernation=on` annotation to the cluster, suspending its execution.

To resume a hibernated cluster, use:

```
kubectl cnpg hibernate off CLUSTER
```

This will remove the hibernation state by setting `cnpg.io/hibernation=off`.

You can check the cluster's status at any time with:

```
kubectl cnpg status CLUSTER
```

This will display the current state of the cluster, including whether it is hibernated.

39.2.16 Benchmarking the database with pgbench

Pgbench can be run against an existing PostgreSQL cluster with following command:

```
kubectl cnpg pgbench CLUSTER -- --time 30 --client 1 --jobs 1
```

Refer to the *Benchmarking pgbench section* for more details.

39.2.17 Benchmarking the storage with fio

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

```
kubectl cnpg fio FIO_JOB_NAME [-n NAMESPACE]
```

Refer to the *Benchmarking fio section* for more details.

39.2.18 Requesting a new physical backup

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

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

```
kubectl cnpg backup CLUSTER
```

or, if using volume snapshots:

```
kubectl cnpg backup CLUSTER -m volumeSnapshot
```

The created backup will be named after the request time:

```
$ kubectl cnpg backup cluster-example
backup/cluster-example-20230121002300 created
```

By default, a newly created backup will use the backup target policy defined in the cluster to choose which instance to run on. However, you can override this policy with the `--backup-target` option.

In the case of volume snapshot backups, you can also use the `--online` option to request an online/hot backup or an offline/cold one: additionally, you can also tune online backups by explicitly setting the `--immediate-checkpoint` and `--wait-for-archive` options.

The Appendix B - Backup on object stores contains more information about the configuration settings.

39.2.19 Launching psql

The `kubectl cnpg psql CLUSTER` 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 (17.5 (Debian 17.5-1.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 (17.5 (Debian 17.5-1.pgdg110+1))

Type "help" for help.

postgres=# select pg_is_in_recovery();
 pg_is_in_recovery
- ------------------
 t
(1 row)

postgres=# \q
```

This command will start `kubectl exec` , and the `kubectl` executable must be reachable in your `PATH` variable to correctly work.

39.2.20 Snapshotting a Postgres cluster

Warning

The *kubectl cnpg snapshot* command has been removed. Please use the *Requesting a new physical backup* to request backups using volume snapshots.

39.2.21 Using pgAdmin4 for evaluation/demonstration purposes only

pgAdmin stands as the most popular and feature-rich open-source administration and development platform for PostgreSQL. For more information on the project, please refer to the official documentation .

Given that the pgAdmin Development Team maintains official Docker container images, you can install pgAdmin in your environment as a standard Kubernetes deployment.

Important

Deployment of pgAdmin in Kubernetes production environments is beyond the scope of this document and, more broadly, of the CloudNativePG project.

However, **for the purposes of demonstration and evaluation**, CloudNativePG offers a suitable solution. The `cnpg` plugin implements the `pgadmin4` command, providing a straightforward method to connect to a given database `Cluster` and navigate its content in a local environment such as `kind`.

For example, you can install a demo deployment of pgAdmin4 for the `cluster-example` cluster as follows:

```
kubectl cnpg pgadmin4 cluster-example
```

This command will produce:

```
ConfigMap/cluster-example-pgadmin4 created
Deployment/cluster-example-pgadmin4 created
Service/cluster-example-pgadmin4 created
Secret/cluster-example-pgadmin4 created

[...]
```

After deploying pgAdmin, forward the port using kubectl and connect through your browser by following the on-screen instructions.

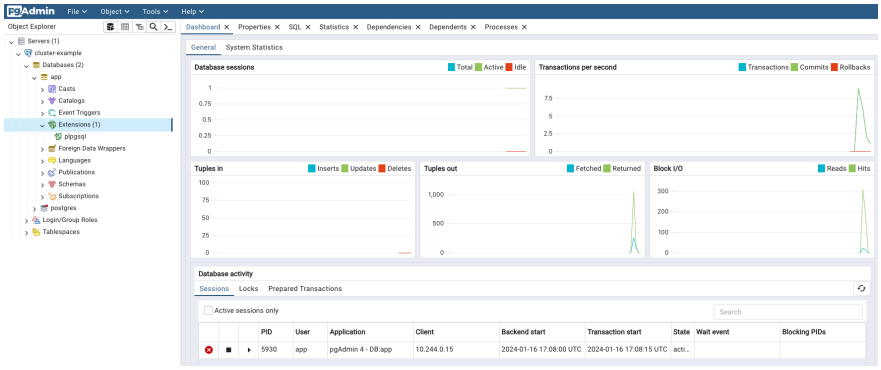


Fig. 1: Screenshot of desktop installation of pgAdmin

As usual, you can use the `--dry-run` option to generate the YAML file:

```
kubectl cnpg pgadmin4 --dry-run cluster-example
```

pgAdmin4 can be installed in either desktop or server mode, with the default being server.

In `server` mode, authentication is required using a randomly generated password, and users must manually specify the database to connect to.

On the other hand, `desktop` mode initiates a pgAdmin web interface without requiring authentication. It automatically connects to the `app` database as the `app` user, making it ideal for quick demos, such as on a local deployment using `kind`:

```
kubectl cnpg pgadmin4 --mode desktop cluster-example
```

After concluding your demo, ensure the termination of the pgAdmin deployment by executing:

```
kubectl cnpg pgadmin4 --dry-run cluster-example | kubectl delete -f -
```

> **Warning**
>
> Never deploy pgAdmin in production using the plugin.

39.2.22 Logical Replication Publications

The `cnpg publication` command group is designed to streamline the creation and removal of *PostgreSQL logical replication publications* . Be aware that these commands are primarily intended for assisting in the creation of logical replication publications, particularly on remote PostgreSQL databases.

> **Warning**
>
> It is crucial to have a solid understanding of both the capabilities and limitations of PostgreSQL's native logical replication system before using these commands. In particular, be mindful of the *logical replication restrictions* .

39.2.23 Creating a new publication

To create a logical replication publication, use the `cnpg publication create` command. The basic structure of this command is as follows:

```
kubectl cnpg publication create \
  --publication PUBLICATION_NAME \
  [--external-cluster EXTERNAL_CLUSTER]
  LOCAL_CLUSTER [options]
```

There are two primary use cases:

- With `--external-cluster` : Use this option to create a publication on an external cluster (i.e. defined in the `externalClusters` stanza). The commands will be issued from the `LOCAL_CLUSTER` , but the publication will be for the data in `EXTERNAL_CLUSTER` .
- Without `--external-cluster` : Use this option to create a publication in the `LOCAL_CLUSTER` PostgreSQL `Cluster` (by default, the `app` database).

> **Warning**
>
> When connecting to an external cluster, ensure that the specified user has sufficient permissions to execute the *CREATE PUBLICATION* command.

You have several options, similar to the CREATE PUBLICATION

command, to define the group of tables to replicate. Notable options include:

- If you specify the `--all-tables` option, you create a publication `FOR ALL TABLES` .
- Alternatively, you can specify multiple occurrences of:
- `--table` : Add a specific table (with an expression) to the publication.
- `--schema` : Include all tables in the specified database schema (available from PostgreSQL 15).

The `--dry-run` option enables you to preview the SQL commands that the plugin will execute.

For additional information and detailed instructions, type the following command:

```
kubectl cnpg publication create --help
```

Example

Given a `source-cluster` and a `destination-cluster`, we would like to create a publication for the data on `source-cluster`. The `destination-cluster` has an entry in the `externalClusters` stanza pointing to `source-cluster`.

We can run:

```
kubectl cnpg publication create destination-cluster  \
  --external-cluster=source-cluster --all-tables
```

which will create a publication for all tables on `source-cluster`, running the SQL commands on the `destination-cluster`.

Or instead, we can run:

```
kubectl cnpg publication create source-cluster \
  --publication=app --all-tables
```

which will create a publication named `app` for all the tables in the `source-cluster`, running the SQL commands on the source cluster.

> **Note**
>
> There are two sample files that have been provided for illustration and inspiration: *logical-source* and *logical-destination*.

39.2.24 Dropping a publication

The `cnpg publication drop` command seamlessly complements the `create` command by offering similar key options, including the publication name, cluster name, and an optional external cluster. You can drop a `PUBLICATION` with the following command structure:

```
kubectl cnpg publication drop \
  --publication PUBLICATION_NAME \
  [--external-cluster EXTERNAL_CLUSTER]
  LOCAL_CLUSTER [options]
```

To access further details and precise instructions, use the following command:

```
kubectl cnpg publication drop --help
```

39.2.25 Logical Replication Subscriptions

The `cnpg subscription` command group is a dedicated set of commands designed to simplify the creation and removal of *PostgreSQL logical replication subscriptions*. These commands are specifically crafted to aid in the establishment of logical replication subscriptions, especially when dealing with remote PostgreSQL databases.

> **Warning**
>
> Before using these commands, it is essential to have a comprehensive understanding of both the capabilities and limitations of PostgreSQL's native logical replication system. In particular, be mindful of the *logical replication restrictions* .

In addition to subscription management, we provide a helpful command for synchronizing all sequences from the source cluster. While its applicability may vary, this command can be particularly useful in scenarios involving major upgrades or data import from remote servers.

39.2.26 Creating a new subscription

To create a logical replication subscription, use the `cnpg subscription create` command. The basic structure of this command is as follows:

```
kubectl cnpg subscription create \
  --subscription SUBSCRIPTION_NAME \
  --publication PUBLICATION_NAME \
  --external-cluster EXTERNAL_CLUSTER \
  LOCAL_CLUSTER [options]
```

This command configures a subscription directed towards the specified publication in the designated external cluster, as defined in the `externalClusters` stanza of the `LOCAL_CLUSTER` .

For additional information and detailed instructions, type the following command:

```
kubectl cnpg subscription create --help
```

Example

As in the section on publications, we have a `source-cluster` and a `destination-cluster` , and we have already created a publication called `app` .

The following command:

```
kubectl cnpg subscription create destination-cluster \
  --external-cluster=source-cluster \
  --publication=app --subscription=app
```

will create a subscription for `app` on the destination cluster.

> **Warning**
>
> Prioritize testing subscriptions in a non-production environment to ensure their effectiveness and identify any potential issues before implementing them in a production setting.

> **Note**
>
> There are two sample files that have been provided for illustration and inspiration: *logical-source* and *logical-destination* .

39.2.27 Dropping a subscription

The `cnpg subscription drop` command seamlessly complements the `create` command. You can drop a `SUBSCRIPTION` with the following command structure:

```
kubectl cnpg subcription drop \
  --subscription SUBSCRIPTION_NAME \
  LOCAL_CLUSTER [options]
```

To access further details and precise instructions, use the following command:

```
kubectl cnpg subscription drop --help
```

39.2.28 Synchronizing sequences

One notable constraint of PostgreSQL logical replication, implemented through publications and subscriptions, is the lack of sequence synchronization. This becomes particularly relevant when utilizing logical replication for live database migration, especially to a higher version of PostgreSQL. A crucial step in this process involves updating sequences before transitioning applications to the new database (*cutover*).

To address this limitation, the `cnpg subscription sync-sequences` command offers a solution. This command establishes a connection with the source database, retrieves all relevant sequences, and subsequently updates local sequences with matching identities (based on database schema and sequence name).

You can use the command as shown below:

```
kubectl cnpg subscription sync-sequences \
  --subscription SUBSCRIPTION_NAME \
  LOCAL_CLUSTER
```

For comprehensive details and specific instructions, utilize the following command:

```
kubectl cnpg subscription sync-sequences --help
```

Example

As in the previous sections for publication and subscription, we have a `source-cluster` and a `destination-cluster` . The publication and the subscription, both called `app` , are already present.

The following command will synchronize the sequences involved in the `app` subscription, from the source cluster into the destination cluster.

```
kubectl cnpg subscription sync-sequences destination-cluster \
  --subscription=app
```

> **Warning**
>
> Prioritize testing subscriptions in a non-production environment to guarantee their effectiveness and detect any potential issues before deploying them in a production setting.

39.3 Integration with K9s

The `cnpg` plugin can be easily integrated in K9s , a popular terminal-based UI to interact with Kubernetes clusters.

See *k9s/plugins.yml* for details.

39.4 Permissions required by the plugin

The plugin requires a set of Kubernetes permissions that depends on the command to execute. These permissions may affect resources and sub-resources like Pods, PDBs, PVCs, and enable actions like `get` , `delete` , `patch` . The following table contains the full details:

Command	Resource Permissions
backup	clusters: get backups: create
certificate	clusters: get secrets: get,create
destroy	pods: get,delete jobs: delete,list PVCs: list,delete,update
fencing	clusters: get,patch pods: get
fio	PVCs: create configmaps: create deployment: create
hibernate	clusters: get,patch,delete pods: list,get,delete pods/exec: create jobs: list PVCs: get,list,update,patch,delete
install	none
logs	clusters: get pods: list pods/log: get
maintenance	clusters: get,patch,list
pgadmin4	clusters: get configmaps: create deployments: create services: create secrets: create
pgbench	clusters: get jobs: create
promote	clusters: get clusters/status: patch pods: get
psql	pods: get,list pods/exec: create
publication	clusters: get pods: get,list pods/exec: create
reload	clusters: get,patch
report cluster	clusters: get pods: list pods/log: get jobs: list events: list PVCs: list
report operator	configmaps: get deployments: get events: list pods: list pods/log: get secrets: get services: get mutatingwebhookconfigurations: list[^1] validatingwebhookconfigurations: list[^1] If OLM is present on the K8s cluster, also: clusterserviceversions: list installplans: list subscriptions: list
restart	clusters: get,patch pods: get,delete
status	clusters: get pods: list pods/exec: create pods/proxy: create PDBs: list
subscription	clusters: get pods: get,list pods/exec: create
version	none

///Footnotes Go Here///

Additionally, assigning the `list` permission on the `clusters` will enable autocompletion for multiple commands.

39.4.1 Role examples

It is possible to create roles with restricted permissions. The following example creates a role that only has access to the cluster logs:

```
- --
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
```

(continues on next page)

(continued from previous page)

```
  name: cnpg-log
rules:
  - verbs:
      - get
    apiGroups:
      - postgresql.cnpg.io
    resources:
      - clusters
  - verbs:
      - list
    apiGroups:
      -
    resources:
      - pods
  - verbs:
      - get
    apiGroups:
      -
    resources:
      - pods/log
```

The next example shows a role with the minimal permissions required to get the cluster status using the plugin's `status` command:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: cnpg-status
rules:
  - verbs:
      - get
    apiGroups:
      - postgresql.cnpg.io
    resources:
      - clusters
  - verbs:
      - list
    apiGroups:
      -
    resources:
      - pods
  - verbs:
      - create
    apiGroups:
      -
    resources:
      - pods/exec
  - verbs:
      - create
    apiGroups:
      -
    resources:
```

(continues on next page)

(continued from previous page)

```
    - pods/proxy
  - verbs:
    - list
    apiGroups:
    - policy
    resources:
    - poddisruptionbudgets
```

Important

Keeping the verbs restricted per *resources* and per *apiGroups* helps to prevent inadvertently granting more than intended permissions.

CHAPTER 40

Automated failover

In the case of unexpected errors on the primary for longer than the `.spec.failoverDelay` (by default `0` seconds), the cluster will go into **failover mode**. This may happen, for example, when:

- The primary pod has a disk failure
- The primary pod is deleted
- The `postgres` container on the primary has any kind of sustained failure

In the failover scenario, the primary cannot be assumed to be working properly.

After cases like the ones above, the readiness probe for the primary pod will start failing. This will be picked up in the controller's reconciliation loop. The controller will initiate the failover process, in two steps:

1. First, it will mark the `TargetPrimary` as `pending`. This change of state will force the primary pod to shutdown, to ensure the WAL receivers on the replicas will stop. The cluster will be marked in failover phase ("Failing over").
2. Once all WAL receivers are stopped, there will be a leader election, and a new primary will be named. The chosen instance will initiate promotion to primary, and, after this is completed, the cluster will resume normal operations. Meanwhile, the former primary pod will restart, detect that it is no longer the primary, and become a replica node.

Important

The two-phase procedure helps ensure the WAL receivers can stop in an orderly fashion, and that the failing primary will not start streaming WALs again upon restart. These safeguards prevent timeline discrepancies between the new primary and the replicas.

During the time the failing primary is being shut down:

1. It will first try a PostgreSQL's *fast shutdown* with `.spec.switchoverDelay` seconds as timeout. This graceful shutdown will attempt to archive pending WALs.
2. If the fast shutdown fails, or its timeout is exceeded, a PostgreSQL's *immediate shutdown* is initiated.

> **Note**
>
> "Fast" mode does not wait for PostgreSQL clients to disconnect and will terminate an online backup in progress. All active transactions are rolled back and clients are forcibly disconnected, then the server is shut down. "Immediate" mode will abort all PostgreSQL server processes immediately, without a clean shutdown.

40.1 RTO and RPO impact

Failover may result in the service being impacted (*RTO and RPO impact*) and/or data being lost (PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode}):

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.

40.2 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.

40.3 Failover Quorum (Quorum-based Failover)

> **Warning**
>
> *Failover quorum* is an experimental feature introduced in version 1.27.0. Use with caution in production environments.

Failover quorum is a mechanism that enhances data durability and safety during failover events in CloudNativePG-managed PostgreSQL clusters.

Quorum-based failover allows the controller to determine whether to promote a replica to primary based on the state of a quorum of replicas. This is useful when stronger data durability is required than the one offered by *Zero-Data-Loss Clusters Through Synchronous Replication* and default automated failover procedures.

When synchronous replication is not enabled, some data loss is expected and accepted during failover, as a replica may lag behind the primary when promoted.

With synchronous replication enabled, the guarantee is that the application will not receive explicit acknowledgment of the successful commit of a transaction until the WAL data is known to be safely received by all required synchronous standbys. This is not enough to guarantee that the operator is able to promote the most advanced replica.

For example, in a three-node cluster with synchronous replication set to `ANY 1 (...)`, data is written to the primary and one standby before a commit is acknowledged. If both the primary and the aligned standby become unavailable (such as during a network partition), the remaining replica may not have the latest data. Promoting it could lose some data that the application considered committed.

Quorum-based failover addresses this risk by ensuring that failover only occurs if the operator can confirm the presence of all synchronously committed data in the instance to promote, and it does not occur otherwise.

This feature allows users to choose their preferred trade-off between data durability and data availability.

Failover quorum can be enabled by setting the annotation `alpha.cnpg.io/failoverQuorum="true"` in the `Cluster` resource.

> **Note**
>
> When this feature is out of the experimental phase, the annotation *alpha.cnpg.io/failoverQuorum* will be replaced by a configuration option in the *Cluster* resource.

40.3.1 How it works

Before promoting a replica to primary, the operator performs a quorum check, following the principles of the Dynamo `R + W > N` consistency model[^1].

In the quorum failover, these values assume the following meaning:

- `R` is the number of *promotable replicas* (read quorum);
- `W` is the number of replicas that must acknowledge the write before the `COMMIT` is returned to the client (write quorum);
- `N` is the total number of potentially synchronous replicas;
- Promotable replicas* are replicas that have these properties:
  - are part of the cluster; - are able to report their state to the operator; - are potentially synchronous;

If `R + W > N`, then we can be sure that among the promotable replicas there is at least one that has confirmed all the synchronous commits, and we can safely promote it to primary. If this is not the case, the controller will not promote any replica to primary, and will wait for the situation to change.

Users can force a promotion of a replica to primary through the `kubectl cnpg promote` command even if the quorum check is failing.

Warning

Manual promotion should only be used as a last resort. Before proceeding, make sure you fully understand the risk of data loss and carefully consider the consequences of prioritizing the resumption of write workloads for your applications.

An additional CRD is used to track the quorum state of the cluster. A `Cluster` with the quorum failover enabled will have a `FailoverQuorum` resource with the same name as the `Cluster` resource. The `FailoverQuorum` CR is created by the controller when the quorum failover is enabled, and it is updated by the primary instance during its reconciliation loop, and read by the operator during quorum checks. It is used to track the latest known configuration of the synchronous replication.

Important

Users should not modify the *FailoverQuorum* resource directly. During PostgreSQL configuration changes, when it is not possible to determine the configuration, the *FailoverQuorum* resource will be reset, preventing any failover until the new configuration is applied.

The `FailoverQuorum` resource works in conjunction with PostgreSQL synchronous replication.

Warning

There is no guarantee that *COMMIT* operations returned to the client but that have not been performed synchronously, such as those made explicitly disabling synchronous replication with *SET synchronous_commit TO local*, will be present on a promoted replica.

40.3.2 Quorum Failover Example Scenarios

In the following scenarios, `R` is the number of promotable replicas, `W` is the number of replicas that must acknowledge a write before commit, and `N` is the total number of potentially synchronous replicas. The "Failover" column indicates whether failover is allowed under quorum failover rules.

40.3.3 Scenario 1: Three-node cluster, failing pod(s)

A cluster with `instances: 3`, `synchronous.number=1`, and `dataDurability=required`.

- If only the primary fails, two promotable replicas remain (R=2). Since `R + W > N` (2 + 1 > 2), failover is allowed and safe.
- If both the primary and one replica fail, only one promotable replica remains (R=1). Since `R + W = N` (1 + 1 = 2), failover is not allowed to prevent possible data loss.

R	W	N	Failover
2	1	2	OK
1	1	2	NO

40.3.4 Scenario 2: Three-node cluster, network partition

A cluster with `instances: 3`, `synchronous.number: 1`, and `dataDurability: required` experiences a network partition.

- If the operator can communicate with the primary, no failover occurs. The cluster can be impacted if the primary cannot reach any standby, since it won't commit transactions due to synchronous replication requirements.
- If the operator cannot reach the primary but can reach both replicas (R=2), failover is allowed. If the operator can reach only one replica (R=1), failover is not allowed, as the synchronous one may be the other one.

R	W	N	Failover
2	1	2	OK
1	1	2	NO

40.3.5 Scenario 3: Five-node cluster, network partition

A cluster with `instances: 5`, `synchronous.number=2`, and `dataDurability=required` experiences a network partition.

- If the operator can communicate with the primary, no failover occurs. The cluster can be impacted if the primary cannot reach at least two standbys, as since it won't commit transactions due to synchronous replication requirements.
- If the operator cannot reach the primary but can reach at least three replicas (R=3), failover is allowed. If the operator can reach only two replicas (R=2), failover is not allowed, as the synchronous one may be the other one.

R	W	N	Failover
3	2	4	OK
2	2	4	NO

40.3.6 Scenario 4: Three-node cluster with remote synchronous replicas

A cluster with `instances: 3` and remote synchronous replicas defined in `standbyNamesPre` or `standbyNamesPost`. We assume that the primary is failing.

This scenario requires an important consideration. Replicas listed in `standbyNamesPre` or `standbyNamesPost` are not counted in `R` (they cannot be promoted), but are included in `N` (they may have received synchronous writes). So, if `synchronous.number <= len(standbyNamesPre) + len(standbyNamesPost)`, failover is not possible, as no local replica can be guaranteed to have the required data. The operator prevents such configurations during validation, but some invalid configurations are shown below for clarity.

Example configurations:

Configuration #1 (valid):

```
instances: 3
postgresql:
```

(continues on next page)

(continued from previous page)

```
  synchronous:
    method: any
    number: 2
    standbyNamesPre:
      - angus
```

In this configuration, when the primary fails, `R = 2` (the local replicas), `W = 2` , and `N = 3` (2 local replicas + 1 remote), allowing failover. In case of an additional replica failing (`R = 1`) failover is not allowed.

R	W	N	Failover
3	2	4	OK
2	2	4	NO

Configuration #2 (invalid):

```
instances: 3
postgresql:
  synchronous:
    method: any
    number: 1
    maxStandbyNamesFromCluster: 1
    standbyNamesPre:
      - angus
```

In this configuration, `R = 2` (the local replicas), `W = 1` , and `N = 3` (2 local replicas + 1 remote). Failover is not possible in this setup, so quorum failover can not be enabled with this configuration.

R	W	N	Failover
1	1	2	NO

Configuration #3 (invalid):

```
instances: 3
postgresql:
  synchronous:
    method: any
    number: 1
    maxStandbyNamesFromCluster: 0
    standbyNamesPre:
      - angus
      - malcolm
```

In this configuration, `R = 0` (the local replicas), `W = 1` , and `N = 2` (0 local replicas + 2 remote). Failover is not possible in this setup, so quorum failover can not be enabled with this configuration.

R	W	N	Failover
0	1	2	NO

40.3.7 Scenario 5: Three-node cluster, preferred data durability, network partition

Consider a cluster with `instances: 3`, `synchronous.number=1`, and `dataDurability=preferred` that experiences a network partition.

- If the operator can communicate with both the primary and the API server, the primary continues to operate, removing unreachable standbys from the `synchronous_standby_names` set.
- If the primary cannot reach the operator or API server, a quorum check is performed. The `FailoverQuorum` status cannot have changed, as the primary cannot have received new configuration. If the operator can reach both replicas, failover is allowed (`R=2`). If only one replica is reachable (`R=1`), failover is not allowed.

R	W	N	Failover
2	1	2	OK
1	1	2	NO

^1

CHAPTER 41

Troubleshooting

In this page, you can find some basic information on how to troubleshoot CloudNativePG in your Kubernetes cluster deployment.

> **Hint**
>
> As a Kubernetes administrator, you should have the kubectl page bookmarked!

41.1 Before you start

41.1.1 Kubernetes environment

What can make a difference in a troubleshooting activity is to provide clear information about the underlying Kubernetes system.

Make sure you know:

- the Kubernetes distribution and version you are using
- the specifications of the nodes where PostgreSQL is running
- as much as you can about the actual Other S3-compatible Object Storages providers , including storage class and benchmarks you have done before going into production.
- which relevant Kubernetes applications you are using in your cluster (i.e. Prometheus, Grafana, Istio, 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

41.1.2 Useful utilities

On top of the mandatory `kubectl` utility, for troubleshooting, we recommend the following plugins/utilities to be available in your system:

- Backup {postgresql-cnpg-io-v1-Backup} for `kubectl`
- jq , a lightweight and flexible command-line JSON processor
- grep , searches one or more input files for lines containing a match to a specified pattern. It is already available in most *nix distros. If you are on Windows OS, you can use findstr as an alternative to `grep` or directly use wsl

and install your preferred *nix distro and use the tools mentioned above.

41.2 First steps

To quickly get an overview of the cluster or installation, the `kubectl` plugin is the primary tool to use:

1. the *status subcommand* provides an overview of a cluster
2. the *report subcommand* provides the manifests for clusters and the operator deployment. It can also include logs using the `--logs` option. The report generated via the plugin will include the full cluster manifest.

The plugin can be installed on air-gapped systems via packages. Please refer to the *plugin document* for complete instructions.

41.3 Are there backups?

After getting the cluster manifest with the plugin, you should verify if backups are set up and working.

Before proceeding with troubleshooting operations, it may be advisable to perform an emergency backup depending on your findings regarding backups. Refer to the following section for instructions.

It is **extremely risky** to operate a production database without keeping regular backups.

41.4 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.

41.5 Logs

All resources created and managed by CloudNativePG log to standard output in accordance with Kubernetes conventions, using *Standard output logging of PostgreSQL error messages in JSON format* .

While logs are typically processed at the infrastructure level and include those from CloudNativePG, accessing logs directly from the command line interface is critical during troubleshooting. You have three primary options for doing so:

- Use the `kubectl logs` command to retrieve logs from a specific resource, and apply `jq` for better readability.
- Use the *kubectl cnpg logs* for CloudNativePG-specific logging.
- Leverage specialized open-source tools like stern , which can aggregate logs from multiple resources (e.g., all pods in a PostgreSQL cluster by selecting the `cnpg.io/clusterName` label), filter log entries, customize output formats, and more.

> **Note**
>
> The following sections provide examples of how to retrieve logs for various resources when troubleshooting CloudNativePG.

41.6 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>
```

41.6.1 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
```

(continues on next page)

(continued from previous page)

```
System ID:           7044925089871458324
PostgreSQL Image:    ghcr.io/cloudnative-pg/postgresql:17.5-3
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
```

41.7 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.

Get PostgreSQL container image version:

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

Output:

```
Image Name:    ghcr.io/cloudnative-pg/postgresql:17.5-3
```

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

41.8 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:

```
kubectl logs -n <NAMESPACE> <CLUSTER>-<N> | \
  jq -r select(.logger=="postgres") | [.ts, .record.message] | @csv
```

If the timestamp is displayed in Unix Epoch time, you can convert it to 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
```

41.8.1 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:

```
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
```

41.9 Backup information

You can list the backups that have been created for a named cluster with:

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

41.10 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.

41.11 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* .

41.12 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.

41.12.1 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.

```
$ 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
```

(continues on next page)

(continued from previous page)

```
      status: "False"
      type: LastBackupSucceeded

    - message: Cluster Is Not Ready
      reason: ClusterIsNotReady
      status: "False"
      type: Ready
```

41.13 Networking

CloudNativePG requires basic networking and connectivity in place. You can find more information in the *Networking* section.

If installing CloudNativePG in an existing environment, there might be network policies in place, or other network configuration made specifically for the cluster, which could have an impact on the required connectivity between the operator and the cluster pods and/or the between the pods.

You can look for existing network policies with the following command:

```
kubectl get networkpolicies
```

There might be several network policies set up by the Kubernetes network administrator.

```
$ kubectl get networkpolicies
NAME                   POD-SELECTOR                        AGE
allow-prometheus       cnpg.io/cluster=cluster-example     47m
default-deny-ingress   <none>                              57m
```

41.14 PostgreSQL core dumps

Although rare, PostgreSQL can sometimes crash and generate a core dump in the `PGDATA` folder. When that happens, normally it is a bug in PostgreSQL (and most likely it has already been solved - this is why it is important to always run the latest minor version of PostgreSQL).

CloudNativePG allows you to control what to include in the core dump through the `cnpg.io/coredumpFilter` annotation.

> **Note**
>
> Please refer to *Labels and annotations*

for more details on the standard annotations that CloudNativePG provides.

By default, the `cnpg.io/coredumpFilter` is set to `0x31` in order to exclude shared memory segments from the dump, as this is the safest approach in most cases.

> **Note**
>
> Please refer to Core dump filtering settings . for more details on how to set the bitmask that controls the core dump filter.

> **Important**
>
> Beware that this setting only takes effect during Pod startup and that changing the annotation doesn't trigger an automated rollout of the instances.

Although you might not personally be involved in inspecting core dumps, you might be asked to provide them so that a Postgres expert can look into them. First, verify that you have a core dump in the `PGDATA` directory with the following command (please run it against the correct pod where the Postgres instance is running):

```
kubectl exec -ti POD -c postgres \
  -- find /var/lib/postgresql/data/pgdata -name core.*
```

Under normal circumstances, this should return an empty set. Suppose, for example, that we have a core dump file:

```
/var/lib/postgresql/data/pgdata/core.14177
```

Once you have verified the space on disk is sufficient, you can collect the core dump on your machine through `kubectl cp` as follows:

```
kubectl cp POD:/var/lib/postgresql/data/pgdata/core.14177 core.14177
```

You now have the file. Make sure you free the space on the server by removing the core dumps.

41.15 Some known issues

41.15.1 Storage is full

In case the storage is full, the PostgreSQL pods will not be able to write new data, or, in case of the disk containing the WAL segments being full, PostgreSQL will shut down.

If you see messages in the logs about the disk being full, you should increase the size of the affected PVC. You can do this by editing the PVC and changing the `spec.resources.requests.storage` field. After that, you should also update the Cluster resource with the new size to apply the same change to all the pods. Please look at the *Volume expansion* in the documentation.

If the space for WAL segments is exhausted, the pod will be crash-looping and the cluster status will report `Not enough disk space`. Increasing the size in the PVC and then in the Cluster resource will solve the issue. See also the *Disk Full Failure*

41.15.2 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
```

41.15.3 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
```

41.15.4 Cluster stuck in `Creating new replica`

Cluster is stuck in "Creating a new replica", while pod logs don't show relevant problems. This has been found to be related to the next issue *Networking is impaired by installed Network Policies* . Networking issues are reflected in the status column as follows:

```
Instance Status Extraction Error: HTTP communication issue
```

41.15.5 Networking is impaired by installed Network Policies

As pointed out in the *Networking* , local network policies could prevent some of the required connectivity.

A tell-tale sign that connectivity is impaired is the presence in the operator logs of messages like:

```
"Cannot extract Pod status", […snipped…] "Get \"http://<pod IP>:8000/pg/status\":␣
↪dial tcp <pod IP>:8000: i/o timeout"
```

You should list the network policies, and look for any policies restricting connectivity.

```
$ kubectl get networkpolicies
NAME                   POD-SELECTOR                       AGE
allow-prometheus       cnpg.io/cluster=cluster-example    47m
default-deny-ingress   <none>                             57m
```

For example, in the listing above, `default-deny-ingress` seems a likely culprit. You can drill into it:

```
$ kubectl get networkpolicies default-deny-ingress -o yaml
<…snipped…>
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

In the *networking page* you can find a network policy file that you can customize to create a `NetworkPolicy` explicitly allowing the operator to connect cross-namespace to cluster pods.

41.15.6 Error while bootstrapping the data directory

If your Cluster's initialization job crashes with a "Bus error (core dumped) child process exited with exit code 135", you likely need to fix the Cluster hugepages settings.

The reason is the incomplete support of hugepages in the cgroup v1 that should be fixed in v2. For more information, check the PostgreSQL BUG #17757: Nothonoring huge_pages setting during initdb causes DB crash inKubernetes .

To check whether hugepages are enabled, run `grep HugePages /proc/meminfo` on the Kubernetes node and check if hugepages are present, their size, and how many are free.

If the hugepages are present, you need to configure how much hugepages memory every PostgreSQL pod should have available.

For example:

```
postgresql:
  parameters:
    shared_buffers: "128MB"

resources:
  requests:
    memory: "512Mi"
  limits:
    hugepages-2Mi: "512Mi"
```

Please remember that you must have enough hugepages memory available to schedule every Pod in the Cluster (in the example above, at least 512MiB per Pod must be free).

41.15.7 Bootstrap job hangs in running status

If your Cluster's initialization job hangs while in `Running` status with the message: "error while waiting for the API server to be reachable", you probably have a network issue preventing communication with the Kubernetes API server. Initialization jobs (like most of jobs) need to access the Kubernetes API. Please check your networking.

Another possible cause is when you have sidecar injection configured. Sidecars such as Istio may make the network temporarily unavailable during startup. If you have sidecar injection enabled, retry with injection disabled.

41.15.8 Replicas take over two minutes to reconnect after a failover

When the primary instance fails, the operator promotes the most advanced standby to the primary role. Other standby instances then attempt to reconnect to the `-rw` service for replication. However, during this reconnection process, `kube-proxy` may not have updated its routing information yet. As a result, the initial `SYN` packet sent by the standby instances might fail to reach its intended destination.

If the network is configured to silently drop packets instead of rejecting them, standby instances will not receive a response and will retry the connection after an exponential backoff period. On Linux systems, the default value

for the `tcp_syn_retries` kernel parameter is 6, meaning the system will attempt to establish the connection for approximately 127 seconds before giving up. This prolonged retry period can significantly delay the reconnection process. For more details, consult the *tcp_syn_retries documentation* .

You can work around this issue by setting `STANDBY_TCP_USER_TIMEOUT` in the *Operator configuration* . This will cause the standby instances to close the TCP connection if the initial `SYN` packet is not acknowledged within the specified timeout, allowing them to retry the connection more quickly.

CHAPTER 42

Fencing

Fencing in CloudNativePG is the ultimate process of protecting the data in one, more, or even all instances of a PostgreSQL cluster when they appear to be malfunctioning. When an instance is fenced, the PostgreSQL server process (`postmaster`) is guaranteed to be shut down, while the pod is kept running. This makes sure that, until the fence is lifted, data on the pod is not modified by PostgreSQL and that the file system can be investigated for debugging and troubleshooting purposes.

42.1 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
```

(continues on next page)

(continued from previous page)

```
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"]
[...]
```

42.2 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 "*"
```

42.3 How fencing works

Once an instance is set for fencing, the procedure to shut down the `postmaster` process is initiated, identical to the one of the switchover. This consists of an initial fast shutdown with a timeout set to `.spec.switchoverDelay` , followed by an immediate shutdown. Then:

- the Pod will be kept alive
- the Pod won't be marked as *Ready*
- all the changes that don't require the Postgres instance to be up will be reconciled, including:
  - configuration files
  - certificates and all the cryptographic material
- metrics will not be collected, except `cnpg_collector_fencing_on` which will be set to 1

> **Warning**
>
> If a **primary instance** is fenced, its postmaster process is shut down but no failover is performed, interrupting the operativity of the applications. When the fence will be lifted, the primary instance will be started up again without performing a failover.

Given that, we advise users to fence primary instances only if strictly required.

If a fenced instance is deleted, the pod will be recreated normally, but the postmaster won't be started. This can be extremely helpful when instances are `Crashlooping`.

CHAPTER 43

Declarative hibernation

CloudNativePG is designed to keep PostgreSQL clusters up, running and available anytime.

There are some kinds of workloads that require the database to be up only when the workload is active. Batch-driven solutions are one such case.

In batch-driven solutions, the database needs to be up only when the batch process is running.

The declarative hibernation feature enables saving CPU power by removing the database Pods, while keeping the database PVCs.

43.1 Hibernation

To hibernate a cluster, set the `cnpg.io/hibernation=on` annotation:

```
$ kubectl annotate cluster <cluster-name> --overwrite cnpg.io/hibernation=on
```

A hibernated cluster won't have any running Pods, while the PVCs are retained so that the cluster can be rehydrated at a later time. Replica PVCs will be kept in addition to the primary's PVC.

The hibernation procedure will delete the primary Pod and then the replica Pods, avoiding switchover, to ensure the replicas are kept in sync.

The hibernation status can be monitored by looking for the `cnpg.io/hibernation` condition:

```
$ kubectl get cluster <cluster-name> -o "jsonpath={.status.conditions[?(.type==\"cnpg.
↪io/hibernation\")]}"

{
        "lastTransitionTime":"2023-03-05T16:43:35Z",
        "message":"Cluster has been hibernated",
        "reason":"Hibernated",
        "status":"True",
        "type":"cnpg.io/hibernation"
}
```

The hibernation status can also be read with the `status` sub-command of the `cnpg` plugin for `kubectl` :

```
$ kubectl cnpg status <cluster-name>
Cluster Summary
Name:               cluster-example
Namespace:          default
PostgreSQL Image:   ghcr.io/cloudnative-pg/postgresql:17.5
Primary instance:   cluster-example-2
Status:             Cluster in healthy state
Instances:          3
Ready instances:    0

Hibernation
Status   Hibernated
Message  Cluster has been hibernated
Time     2023-03-05 16:43:35 +0000 UTC
[..]
```

43.2 Rehydration

To rehydrate a cluster, either set the `cnpg.io/hibernation` annotation to `off` :

```
$ kubectl annotate cluster <cluster-name> --overwrite cnpg.io/hibernation=off
```

Or, just unset it altogether:

```
$ kubectl annotate cluster <cluster-name> cnpg.io/hibernation-
```

The Pods will be recreated and the cluster will resume operation.

CHAPTER 44

PostGIS

PostGIS is a very popular open source extension for PostgreSQL that introduces support for storing GIS (Geographic Information Systems) objects in the database and be queried via SQL.

Important

This section assumes you are familiar with PostGIS and provides some basic information about how to create a new PostgreSQL cluster with a PostGIS database in Kubernetes via CloudNativePG.

The CloudNativePG Community maintains container images that are built on top of the official *PostGIS images hosted on DockerHub* . For more information please visit:

- The *postgis-containers*
- The *postgis-containers*

44.1 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* .

44.2 Create a new PostgreSQL cluster with PostGIS

Let's suppose you want to create a new PostgreSQL 17 cluster with PostGIS 3.2.

The first step is to ensure you use the right PostGIS container image for the operand, and properly set the `.spec.imageName` option in the `Cluster` resource.

The below provides some guidance on how the creation of a PostGIS cluster can be done.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgis-example
spec:
  instances: 1
  imageName: ghcr.io/cloudnative-pg/postgis:17
  storage:
    size: 1Gi
  postgresql:
    parameters:
      log_statement: ddl
- --
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: postgis-example-app
spec:
  name: app
  owner: app
  cluster:
    name: postgis-example
  extensions:
  - name: postgis
  - name: postgis_topology
  - name: fuzzystrmatch
  - name: postgis_tiger_geocoder
```

> **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 17 - you should use a specific image name or, preferably, the SHA256 digest for true immutability.

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgis-example
spec:
  instances: 1
  imageName: ghcr.io/cloudnative-pg/postgis:17
  storage:
    size: 1Gi
  postgresql:
    parameters:
      log_statement: ddl
- --
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: postgis-example-app
spec:
  name: app
  owner: app
  cluster:
    name: postgis-example
  extensions:
  - name: postgis
  - name: postgis_topology
  - name: fuzzystrmatch
  - name: postgis_tiger_geocoder
```

The example leverages the `Database` resource's declarative extension management to add the specified extensions to the `app` database.

Note

For more details, see the *Managing Extensions in a Database* .

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 cnpg psql postgis-example -- app
psql (17.5 (Debian 17.5-1.pgdg110+2))
Type "help" for help.

app=# SELECT * FROM pg_available_extensions WHERE name ~ ^postgis ORDER BY 1;
           name           | default_version | installed_version |                 ␣
↪      comment
- ------------------------+-----------------+-------------------+-------------------
↪----------------------------------------
 postgis                  | 3.5.2           | 3.5.2             | PostGIS geometry␣
```

(continues on next page)

(continued from previous page)

```
↪and geography spatial types and functions
 postgis-3                | 3.5.2           |                    | PostGIS geometry␣
↪and geography spatial types and functions
 postgis_raster           | 3.5.2           |                    | PostGIS raster␣
↪types and functions
 postgis_raster-3         | 3.5.2           |                    | PostGIS raster␣
↪types and functions
 postgis_sfcgal           | 3.5.2           |                    | PostGIS SFCGAL␣
↪functions
 postgis_sfcgal-3         | 3.5.2           |                    | PostGIS SFCGAL␣
↪functions
 postgis_tiger_geocoder   | 3.5.2           | 3.5.2              | PostGIS tiger␣
↪geocoder and reverse geocoder
 postgis_tiger_geocoder-3 | 3.5.2           |                    | PostGIS tiger␣
↪geocoder and reverse geocoder
 postgis_topology         | 3.5.2           | 3.5.2              | PostGIS topology␣
↪spatial types and functions
 postgis_topology-3       | 3.5.2           |                    | PostGIS topology␣
↪spatial types and functions
(10 rows)
```

The next step is to verify that the extensions listed in the `Database` resource have been correctly installed in the `app` database.

```
app=# \dx
                                    List of installed extensions
          Name          | Version |   Schema   |                          Description
- ----------------------+---------+------------+--------------------------------------
↪-----------------------
 fuzzystrmatch          | 1.2     | public     | determine similarities and distance␣
↪between strings
 plpgsql                | 1.0     | pg_catalog | PL/pgSQL procedural language
 postgis                | 3.5.2   | public     | PostGIS geometry and geography␣
↪spatial types and functions
 postgis_tiger_geocoder | 3.5.2   | tiger      | PostGIS tiger geocoder and reverse␣
↪geocoder
 postgis_topology       | 3.5.2   | topology   | PostGIS topology spatial types and␣
↪functions
(5 rows)
```

Finally:

```
app=# SELECT postgis_full_version();
                                                                                postgis_
↪full_version
- ----------------------------------------------------------------------------------------
↪----------------------------------------------------------------------------------------
↪---
 POSTGIS="3.5.2 dea6d0a" [EXTENSION] PGSQL="170" GEOS="3.9.0-CAPI-1.16.2" PROJ="7.2.1␣
↪NETWORK_ENABLED=OFF URL_ENDPOINT=https://cdn.proj.org USER_WRITABLE_DIRECTORY=/tmp/
↪proj DATABASE_PATH=/usr/share/proj/proj.db" (compiled against PROJ 7.2.1) LIBXML="2.
↪9.10" LIBJSON="0.15" LIBPROTOBUF="1.3.3" WAGYU="0.5.0 (Internal)" TOPOLOGY
(1 row)
```

CHAPTER 45

End-to-End Tests

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

Kubernetes versions 1.27 through 1.32, and PostgreSQL versions 13 through 17, are tested for each commit, helping detect bugs at an early stage of the development process.

For each tested version of Kubernetes and PostgreSQL, a Kubernetes cluster is created using kind , run on the GitHub Actions platform, and the following suite of E2E tests are performed on that cluster:

- **Basic:**
- Installation of the operator
- Creation of a Cluster
- Usage of a persistent volume for data storage
- **Service connectivity:**
- Connection via services, including read-only
- Connection via user-provided server and/or client certificates
- PgBouncer
- **Self-healing:**
- Failover
- Switchover
- Primary endpoint switch in case of failover in less than 10 seconds
- Primary endpoint switch in case of switchover in less than 20 seconds
- Recover from a degraded state in less than 60 seconds
- PVC Deletion
- Corrupted PVC
- **Backup and Restore:**

- Backup and restore from Volume Snapshots
- Backup and ScheduledBackups execution using Barman Cloud on S3
- Backup and ScheduledBackups execution using Barman Cloud on Azure blob storage
- Restore from backup using Barman Cloud on S3
- Restore from backup using Barman Cloud on Azure blob storage
- Point-in-time recovery (PITR) on Azure, S3 storage
- Wal-Restore (sequential / parallel)
- **Operator:**
- Operator Deployment
- Operator configuration via ConfigMap
- Operator pod deletion
- Operator pod eviction
- Operator upgrade
- Operator High Availability
- **Observability:**
- Metrics collection
- PgBouncer Metrics
- JSON log format
- **Replication:**
- Replication Slots
- Synchronous replication
- Scale-up and scale-down of a Cluster
- Logical replication via declarative Publication / Subscription
- **Replica clusters**
- Bootstrapping a replica cluster from backup
- Bootstrapping a replica cluster via streaming
- Bootstrapping via volume snapshots
- Detaching a replica cluster
- **Plugin:**
- Cluster Hibernation using CNPG plugin
- Fencing
- Creation of a connection certificate
- **Postgres Configuration:**
- Manage PostgreSQL configuration changes
- Rolling updates when changing PostgreSQL images
- Rolling updates when changing ImageCatalog/ClusterImageCatalog images

- Rolling updates on hot standby sensitive parameter changes
- Database initialization via InitDB
- **Pod Scheduling:**
- Tolerations and taints
- Pod affinity using `NodeSelector`
- Rolling updates on PodSpec drift detection
- In-place upgrades
- Multi-Arch availability
- **Cluster Metadata:**
- ConfigMap for Cluster Labels and Annotations
- Object metadata
- **Recovery:**
- Data corruption
- pg_basebackup
- **Importing Databases:**
- Microservice approach
- Monolith approach
- **Storage:**
- Storage expansion
- Dedicated PG_WAL persistent volume
- **Security:** - AppArmor annotation propagation. Executed only on Azure environment
- **Maintenance:**
- Node Drain with maintenance window
- Node Drain with single-instance cluster with/without Pod Disruption Budgets
- **Hibernation** - Declarative hibernation / rehydration
- **Volume snapshots**
- Backup/restore for cold and online snapshots
- Point-in-time recovery (PITR) for cold and online snapshots
- Backups via plugin for cold and online snapshots
- Declarative backups for cold and online snapshots
- **Managed Roles**
- Creation and update of managed roles
- Password maintenance using Kubernetes secrets
- **Tablespaces**
- Declarative creation of tablespaces
- Declarative creation of temporary tablespaces

- Backup / recovery from object storage
- Backup / recovery from volume snapshots
- **Declarative databases**
- Declarative creation of databases with default (retain) reclaim policy
- Declarative creation of databases with delete reclaim policy
- **Major version upgrade** - Upgrade to the latest major version

CHAPTER 46

Container Image Requirements

The CloudNativePG operator for Kubernetes is designed to work with any compatible PostgreSQL container image that meets the following requirements:

- PostgreSQL executables must be available in the system path:
- `initdb`
- `postgres`
- `pg_ctl`
- `pg_controldata`
- `pg_basebackup`
- Proper locale settings configured

Optional Components:

- PGAudit extension (only required if audit logging is needed)
- `du` (used for `kubectl cnpg status`)

Important

Only PostgreSQL versions officially supported by PGDG are allowed.

Note

Barman Cloud executables are no longer required in CloudNativePG. The recommended approach is to use the dedicated Barman Cloud Plugin .

No entry point or command is required in the image definition. CloudNativePG automatically overrides it with its instance manager.

> **Warning**
>
> CloudNativePG only supports **Primary with multiple/optional Hot Standby Servers architecture** for PostgreSQL application container images.

The CloudNativePG community provides and maintains public PostgreSQL container images

that are fully compatible with CloudNativePG. These images are published on ghcr.io .

46.1 Image Tag Requirements

To ensure the operator makes informed decisions, it must accurately detect the PostgreSQL major version. This detection can occur in two ways:

1. Utilizing the `major` field of the `imageCatalogRef` , if defined.
2. Auto-detecting the major version from the image tag of the `imageName` if not explicitly specified.

For auto-detection to work, the image tag must adhere to a specific format. It should commence with a valid PostgreSQL major version number (e.g., 15.6 or 16), optionally followed by a dot and the patch level.

Following this, the tag can include any character combination valid and accepted in a Docker tag, preceded by a dot, an underscore, or a minus sign.

Examples of accepted image tags:

- `12.1`
- `13.3.2.1-1`
- `13.4`
- `14`
- `15.5-10`
- `16.0`

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

> **Note**
>
> Image tag requirements do no apply for images defined in a catalog.

CHAPTER 47

Image Volume Extensions

CloudNativePG supports the **dynamic loading of PostgreSQL extensions** into a `Cluster` at Pod startup using the *Kubernetes `ImageVolume* feature <https://kubernetes.io/docs/tasks/configure-pod-container/image-volumes/>`_

and the `extension_control_path` GUC introduced in PostgreSQL 18, to which this project contributed.

This feature allows you to mount a PostgreSQL extension , packaged as an OCI-compliant container image, as a read-only and immutable volume inside a running pod at a known filesystem path.

You can make the extension available either globally, using the *shared_preload_libraries* , or at the database level through the `CREATE EXTENSION` command. For the latter, you can use the *Configure the application database*

to ensure consistent, automated extension setup within your PostgreSQL databases.

47.1 Benefits

Image volume extensions decouple the distribution of PostgreSQL operand container images from the distribution of extensions. This eliminates the need to define and embed extensions at build time within your PostgreSQL images—a major adoption blocker for PostgreSQL as a containerized workload, including from a security and supply chain perspective.

As a result, you can:

- Use the *official PostgreSQL `minimal* operand images <https://github.com/cloudnative-pg/postgres-containers?tab=readme-ov-file#minimal-images>`_

provided by CloudNativePG.

- Dynamically add the extensions you need to your `Cluster` definitions, without rebuilding or maintaining custom PostgreSQL images.
- Reduce your operational surface by using immutable, minimal, and secure base images while adding only the extensions required for each workload.

Extension images must be built according to the *Image Specifications* .

47.2 Requirements

To use image volume extensions with CloudNativePG, you need:

- **PostgreSQL 18 or later**, with support for `extension_control_path`.
- **Kubernetes 1.33**, with the `ImageVolume` feature gate enabled.
- **CloudNativePG-compatible extension container images**, ensuring:
  - Matching PostgreSQL major version of the `Cluster` resource.
  - Compatible operating system distribution of the `Cluster` resource.
  - Matching CPU architecture of the `Cluster` resource.

47.3 How it works

Extension images are defined in the `.spec.postgresql.extensions` stanza of a `Cluster` resource, which accepts an ordered list of extensions to be added to the PostgreSQL cluster.

> **Note**
>
> For field-level details, see the API reference for `ExtensionConfiguration <API Reference>`.

Each image volume is mounted at `/extensions/<EXTENSION_NAME>`.

By default, CloudNativePG automatically manages the relevant GUCs, setting:

- `extension_control_path` to `/extensions/<EXTENSION_NAME>/share`, allowing PostgreSQL to locate any extension control file within `/extensions/<EXTENSION_NAME>/share/extension`
- `dynamic_library_path` to `/extensions/<EXTENSION_NAME>/lib`

These values are appended in the order in which the extensions are defined in the `extensions` list, ensuring deterministic path resolution within PostgreSQL. This allows PostgreSQL to discover and load the extension without requiring manual configuration inside the pod.

> **Note**
>
> Depending on how your extension container images are built and their layout, you may need to adjust the default *extension_control_path* and *dynamic_library_path* values to match the image structure.

> **Important**
>
> If the extension image includes shared libraries, they must be compiled with the same PostgreSQL major version, operating system distribution, and CPU architecture as the PostgreSQL container image used by your cluster, to ensure compatibility and prevent runtime issues.

47.4 How to add a new extension

Adding an extension to a database in CloudNativePG involves a few steps:

1. Define the extension image in the `Cluster` resource so that PostgreSQL can discover and load it.

2. Add the library to *shared_preload_libraries*

if the extension requires it.

3. Declare the extension in the `Database` resource where you want it installed, if the extension supports `CREATE EXTENSION`.

> **Warning**
>
> Avoid making changes to extension images and PostgreSQL configuration settings (such as *shared_preload_libraries*) simultaneously. First, allow the pod to roll out with the new extension image, then update the PostgreSQL configuration. This limitation will be addressed in a future release of CloudNativePG.

For illustration purposes, this guide uses a simple, fictitious extension named `foo` that supports `CREATE EXTENSION` .

47.4.1 Adding a new extension to a `Cluster` resource

You can add an `ImageVolume` -based extension to a `Cluster` using the `.spec.postgresql.extensions` stanza. For example:

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: foo-18
spec:
  # ...
  postgresql:
    extensions:
      - name: foo
        image:
          reference: # registry path for your extension image
  # ...
```

The `name` field is **mandatory** and **must be unique within the cluster**, as it determines the mount path (`/extensions/foo` in this example). It must consist of *lowercase alphanumeric characters or hyphens (``-``)* and must start and end with an alphanumeric character.

The `image` stanza follows the *Kubernetes `ImageVolume* API <https://kubernetes.io/docs/tasks/configure-pod-container/image-volumes/>`_ . The `reference` must point to a valid container registry path for the extension image.

> **Important**
>
> When a new extension is added to a running *Cluster* , CloudNativePG will automatically trigger a *Rolling Updates* to attach the new image volume to each pod. Before adding a new extension in production, ensure you have thoroughly tested it in a staging environment to prevent configuration issues that could leave your PostgreSQL cluster in an unhealthy state.

Once mounted, CloudNativePG will automatically configure PostgreSQL by appending:

- `/extensions/foo/share` to `extension_control_path`
- `/extensions/foo/lib` to `dynamic_library_path`

This ensures that the PostgreSQL container is ready to serve the `foo` extension when requested by a database, as described in the next section. The `CREATE EXTENSION foo` command, triggered automatically during the reconciliation of the `Database resource <PostgreSQL Database Management>` , will work without additional configuration, as PostgreSQL will locate:

- the extension control file at `/extensions/foo/share/extension/foo.control`
- the shared library at `/extensions/foo/lib/foo.so`

47.4.2 Adding a new extension to a `Database` resource

Once the extension is available in the PostgreSQL instance, you can leverage declarative databases to *manage the lifecycle of your extensions*

within the target database.

Continuing with the `foo` example, you can request the installation of the `foo` extension in the `app` database of the `foo-18` cluster using the following resource definition:

```
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: foo-app
spec:
  name: app
  owner: app
  cluster:
    name: foo-18
  extensions:
    - name: foo
      version: 1.0
```

CloudNativePG will automatically reconcile this resource, executing the `CREATE EXTENSION foo` command inside the `app` database if it is not already installed, ensuring your desired state is maintained without manual intervention.

47.5 Advanced Topics

In some cases, the default expected structure may be insufficient for your extension image, particularly when:

- The extension requires additional system libraries.
- Multiple extensions are bundled in the same image.
- The image uses a custom directory structure.

Following the *"convention over configuration"* paradigm, CloudNativePG allows you to finely control the configuration of each extension image through the following fields:

- `extension_control_path` : A list of relative paths within the container image to be appended to PostgreSQL's `extension_control_path` , allowing it to locate extension control files.
- `dynamic_library_path` : A list of relative paths within the container image to be appended to PostgreSQL's `dynamic_library_path` , enabling it to locate shared library files for extensions.
- `ld_library_path` : A list of relative paths within the container image to be appended to the `LD_LIBRARY_PATH` environment variable of the instance manager process, allowing PostgreSQL to locate required system libraries at runtime.

This flexibility enables you to support complex or non-standard extension images while maintaining clarity and predictability.

47.5.1 Setting Custom Paths

If your extension image does not use the default `lib` and `share` directories for its libraries and control files, you can override the defaults by explicitly setting `extension_control_path` and `dynamic_library_path` .

For example:

```
spec:
  postgresql:
    extensions:
      - name: my-extension
        extension_control_path:
          - my/share/path
        dynamic_library_path:
          - my/lib/path
        image:
          reference: # registry path for your extension image
```

CloudNativePG will configure PostgreSQL with:

- `/extensions/my-extension/my/share/path` appended to `extension_control_path`
- `/extensions/my-extension/my/lib/path` appended to `dynamic_library_path`

This allows PostgreSQL to discover your extension's control files and shared libraries correctly, even with a non-standard layout.

47.5.2 Multi-extension Images

You may need to include multiple extensions within the same container image, adopting a structure where each extension's files reside in their own subdirectory.

For example, to package PostGIS and pgRouting together in a single image, each in its own subdirectory:

```
#  ...

spec:
  # ...
  postgresql:
    extensions:
      - name: geospatial
        extension_control_path:
          - postgis/share
          - pgrouting/share
        dynamic_library_path:
          - postgis/lib
          - pgrouting/lib
        # ...
        image:
          reference: # registry path for your geospatial image
      # ...
    # ...
  # ...
```

47.5.3 Including System Libraries

Some extensions, such as PostGIS, require system libraries that may not be present in the base PostgreSQL image. To support these requirements, you can package the necessary libraries within your extension container image and make them available to PostgreSQL using the `ld_library_path` field.

For example, if your extension image includes a `system` directory with the required libraries:

```
#  ...

spec:
  # ...
  postgresql:
    extensions:
      - name: postgis
        # ...
        ld_library_path:
          - syslib
        image:
          reference: # registry path for your PostGIS image
      # ...
    # ...
  # ...
```

CloudNativePG will set the `LD_LIBRARY_PATH` environment variable to include `/extensions/postgis/system`, allowing PostgreSQL to locate and load these system libraries at runtime.

Important

Since *ld_library_path* must be set when the PostgreSQL process starts, changing this value requires a **cluster restart** for the new value to take effect. CloudNativePG does not currently trigger this restart automatically; you will need to manually restart the cluster (e.g., using *cnpg restart*) after modifying *ld_library_path* .

47.6 Image Specifications

A standard extension container image for CloudNativePG includes two required directories at its root:

- `share` : contains the extension control file (e.g., `<EXTENSION>.control`) and any SQL files.
- `lib` : contains the extension's shared library (e.g., `<EXTENSION>.so`) and any additional required libraries.

Following this structure ensures that the extension will be automatically discoverable and usable by PostgreSQL within CloudNativePG without requiring manual configuration.

Important

We encourage PostgreSQL extension developers to publish OCI-compliant extension images following this layout as part of their artifact distribution, making their extensions easily consumable within Kubernetes environments. Ideally, extension images should target a specific operating system distribution and architecture, be tied to a particular PostgreSQL version, and be built using the distribution's native packaging system (for example, using Debian or RPM packages). This approach ensures consistency, security, and compatibility with the PostgreSQL images used in your clusters.

47.7 Caveats

Currently, adding, removing, or updating an extension image triggers a restart of the PostgreSQL pods. This behavior is inherited from how image volumes

work in Kubernetes.

Before performing an extension update, ensure you have:

- Thoroughly tested the update process in a staging environment.
- Verified that the extension image contains the required upgrade path between the currently installed version and the target version.
- Updated the `version` field for the extension in the relevant `Database` resource definition to align with the new version in the image.

These steps help prevent downtime or data inconsistencies in your PostgreSQL clusters during extension updates.

CHAPTER 48

CNPG-I

The **CloudNativePG Interface** (CNPG-I) is a standard way to extend and customize CloudNativePG without modifying its core codebase.

48.1 Why CNPG-I?

CloudNativePG supports a wide range of use cases, but sometimes its built-in functionality isn't enough, or adding certain features directly to the main project isn't practical.

Before CNPG-I, users had two main options:

- Fork the project to add custom behavior, or
- Extend the upstream codebase by writing custom components on top of it.

Both approaches created maintenance overhead, slowed upgrades, and delayed delivery of critical features.

CNPG-I solves these problems by providing a stable, gRPC-based integration point for extending CloudNativePG at key points in a cluster's lifecycle —such as backups, recovery, and sub-resource reconciliation— without disrupting the core project.

CNPG-I can extend:

- The operator, and/or
- The instance manager running inside PostgreSQL pods.

48.2 Registering a plugin

CNPG-I is inspired by the Kubernetes *Container Storage Interface (CSI)* . The operator communicates with registered plugins using **gRPC**, following the CNPG-I protocol .

CloudNativePG discovers plugins **at startup**. You can register them in one of two ways:

- Sidecar container – run the plugin inside the operator's Deployment
- Standalone Deployment – run the plugin as a separate workload in the same namespace

In both cases, the plugin must be packaged as a container image.

48.2.1 Sidecar Container

When running as a sidecar, the plugin must expose its gRPC server via a **Unix domain socket**. This socket must be placed in a directory shared with the operator container, mounted at the path set in `PLUGIN_SOCKET_DIR` (default: `/plugin`).

Example:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: controller-manager
spec:
  template:
    spec:
      containers:
      - image: cloudnative-pg:latest
        [...]
        name: manager
        volumeMounts:
        - mountPath: /plugins
          name: cnpg-i-plugins

      - image: cnpg-i-plugin-example:latest
        name: cnpg-i-plugin-example
        volumeMounts:
        - mountPath: /plugins
          name: cnpg-i-plugins
      volumes:
      - name: cnpg-i-plugins
        emptyDir: {}
```

48.2.2 Standalone Deployment (recommended)

Running a plugin as its own Deployment decouples its lifecycle from the operator's and allows independent scaling. In this setup, the plugin exposes a TCP gRPC endpoint behind a Service, with **mTLS** for secure communication.

> **Warning**
>
> CloudNativePG does **not** discover plugins dynamically. If you deploy a new plugin, you must **restart the operator** to detect it.

Example Deployment:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cnpg-i-plugin-example
spec:
  template:
    [...]
```

(continues on next page)

(continued from previous page)

```
    spec:
      containers:
      - name: cnpg-i-plugin-example
        image: cnpg-i-plugin-example:latest
        ports:
        - containerPort: 9090
          protocol: TCP
```

The related Service for the plugin must include:

- The label `cnpg.io/plugin: <plugin-name>` — required for CloudNativePG to discover the plugin
- The annotation `cnpg.io/pluginPort: <port>` — specifies the port where the plugin's gRPC server is exposed

Example Service:

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    cnpg.io/pluginPort: "9090"
  labels:
    cnpg.io/pluginName: cnpg-i-plugin-example.my-org.io
  name: cnpg-i-plugin-example
spec:
  ports:
  - port: 9090
    protocol: TCP
    targetPort: 9090
  selector:
    app: cnpg-i-plugin-example
```

48.2.3 Configuring TLS Certificates

When a plugin runs as a `Deployment` , communication with CloudNativePG happens over the network. To secure it, **mTLS is enforced**, requiring TLS certificates for both sides.

Certificates must be stored as Kubernetes TLS Secrets

and referenced in the plugin's Service annotations (`cnpg.io/pluginClientSecret` and `cnpg.io/pluginServerSecret`):

```
apiVersion: v1
kind: Service
metadata:
  annotations:
    cnpg.io/pluginClientSecret: cnpg-i-plugin-example-client-tls
    cnpg.io/pluginServerSecret: cnpg-i-plugin-example-server-tls
    cnpg.io/pluginPort: "9090"
  name: barman-cloud
  namespace: postgresql-operator-system
spec:
    [...]
```

> **Note**
>
> You can provide your own certificate bundles, but the recommended method is to use Cert-manager .

48.3 Using a plugin

To enable a plugin, configure the `.spec.plugins` section in your `Cluster` resource. Refer to the CloudNativePG API Reference for the full PluginConfiguration

specification.

Example:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-with-plugins
spec:
  instances: 1
  storage:
    size: 1Gi
  plugins:
  - name: cnpg-i-plugin-example.my-org.io
    enabled: true
    parameters:
      key1: value1
      key2: value2
```

Each plugin may have its own parameters—check the plugin's documentation for details. The `name` field in `spec.plugins` depends on how the plugin is deployed:

- Sidecar container: use the Unix socket file name
- Deployment: use the value from the Service's `cnpg.io/pluginName` label

48.4 Community plugins

The CNPG-I protocol has quickly become a proven and reliable pattern for extending CloudNativePG while keeping the core project maintainable. Over time, the community has built and shared plugins that address real-world needs and serve as examples for developers.

For a complete and up-to-date list of plugins built with CNPG-I, please refer to the CNPG-I GitHub page .

CHAPTER 49

Operator capability levels

These capabilities were implemented by CloudNativePG, classified using the Operator SDK definition of Capability Levels

framework.

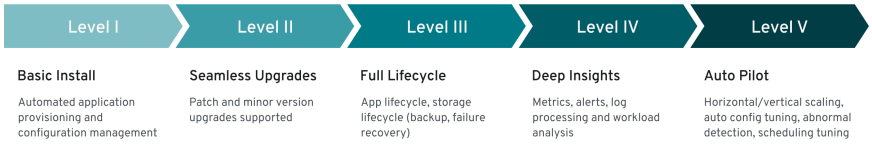


Fig. 1: Operator Capability Levels

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.

49.1 Level 1: Basic install

Capability level 1 involves installing and configuring the operator. This category includes usability and user experience enhancements, such as improvements in how you interact with the operator and a PostgreSQL cluster configuration.

Important

We consider information security part of this level.

49.1.1 Operator deployment via declarative configuration

The operator is installed in a declarative way using a Kubernetes manifest that defines four major `CustomResourceDefinition` objects: `Cluster` , `Pooler` , `Backup` , and `ScheduledBackup` .

49.1.2 PostgreSQL cluster deployment via declarative configuration

You define a PostgreSQL cluster (operand) using the `Cluster` custom resource in a fully declarative way. The PostgreSQL version is determined by the operand container image defined in the CR, which is automatically fetched from the requested registry. When deploying an operand, the operator also creates the following resources: `Pod` , `Service` , `Secret` , `ConfigMap` ,`PersistentVolumeClaim` , `PodDisruptionBudget` , `ServiceAccount` , `RoleBinding` , and `Role` .

49.1.3 Override of operand images through the CRD

The operator is designed to support any operand container image with PostgreSQL inside. By default, the operator uses the latest available minor version of the latest stable major version supported by the PostgreSQL community and published on ghcr.io. You can use any compatible image of PostgreSQL supporting the primary/standby architecture directly by setting the `imageName` attribute in the CR. The operator also supports `imagePullSecrets` to access private container registries, and it supports digests and tags for finer control of container image immutability. If you prefer not to specify an image name, you can leverage *image catalogs* by simply referencing the PostgreSQL major version. Moreover, image catalogs enable you to effortlessly create custom catalogs, directing to images based on your specific requirements.

49.1.4 Labels and annotations

You can configure the operator to support inheriting labels and annotations that are defined in a cluster's metadata. The goal is to improve the organization of the CloudNativePG deployment in your Kubernetes infrastructure.

49.1.5 Self-contained instance manager

Instead of relying on an external tool to coordinate PostgreSQL instances in the Kubernetes cluster pods, such as Patroni or Stolon, the operator injects the operator executable inside each pod, in a file named `/controller/manager` . The application is used to control the underlying PostgreSQL instance and to reconcile the pod status with the instance based on the PostgreSQL cluster topology. The instance manager also starts a web server that's invoked by the `kubelet` for probes. Unix signals invoked by the `kubelet` are filtered by the instance manager. Where appropriate, they're forwarded to the `postgres` process for fast and controlled reactions to external events. The instance manager is written in Go and has no external dependencies.

49.1.6 Storage configuration

Storage is a critical component in a database workload. Taking advantage of the Kubernetes native capabilities and resources in terms of storage, the operator gives you enough flexibility to choose the right storage for your workload requirements, based on what the underlying Kubernetes environment can offer. This implies choosing a particular storage class in a public cloud environment or fine-tuning the generated PVC through a PVC template in the CR's `storage` parameter.

For better performance and finer control, you can also choose to host your cluster's write-ahead log (WAL, also known as `pg_wal`) on a separate volume, preferably on different storage. The *Benchmarking* section of the documentation provides detailed instructions on benchmarking both storage and the database before production. It relies on the `cnpg` plugin to ensure optimal performance and reliability.

49.1.7 Replica configuration

The operator detects replicas in a cluster through a single parameter, called `instances` . If set to `1` , the cluster comprises a single primary PostgreSQL instance with no replica. If higher than `1` , the operator manages `instances -1` replicas, including high availability (HA) through automated failover and rolling updates through switchover operations.

CloudNativePG manages replication slots for all replicas in the high-availability cluster. It also supports user-defined physical replication slots on the primary and enables logical decoding failover—natively for PostgreSQL 17 and later using `sync_replication_slots` , and through the `pg_failover_slots` extension for earlier versions.

49.1.8 Service Configuration

By default, CloudNativePG creates three Kubernetes ManagedServices {postgresql-cnpg-io-v1-ManagedServices}

for applications to access the cluster via the network:

- One pointing to the primary for read/write operations.
- One pointing to replicas for read-only queries.
- A generic one pointing to any instance for read operations.

You can disable the read-only and read services via configuration. Additionally, you can leverage the service template capability to create custom service resources, including load balancers, to access PostgreSQL outside Kubernetes. This is particularly useful for DBaaS purposes.

49.1.9 Database configuration

The operator is designed to bootstrap a PostgreSQL cluster with a single database. The operator transparently manages network access to the cluster through three Kubernetes services provisioned and managed for read-write, read, and read-only workloads. Using the convention-over-configuration approach, the operator creates a database called `app` , by default owned by a regular Postgres user with the same name. You can specify both the database name and the user name, if required, as part of the bootstrap.

Additional databases can be created or managed via *declarative database management* using the `Database` CRD, also supporting extensions, schemas and foreign data wrappers (FDW).

Although no configuration is required to run the cluster, you can customize both PostgreSQL runtime configuration and PostgreSQL host-based authentication rules in the `postgresql` section of the CR.

49.1.10 Configuration of Postgres roles, users, and groups

CloudNativePG supports *management of PostgreSQL roles, users, and groups through declarative configuration* using the `.spec.managed.roles` stanza.

49.1.11 Pod security standards

For InfoSec requirements, the operator doesn't require privileged mode for any container. It enforces a read-only root filesystem to guarantee containers immutability for both the operator and the operand pods. It also explicitly sets the required security contexts.

49.1.12 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

49.1.13 Topology spread constraints

The cluster's `topologySpreadConstraints` section enables additional control of scheduling pods across topologies, enhancing what affinity and anti-affinity can offer.

49.1.14 Command-line interface

CloudNativePG doesn't have its own command-line interface. It relies on the best command-line interface for Kubernetes, kubectl, by providing a plugin called `cnpg` . This plugin enhances and simplifies your PostgreSQL cluster management experience.

49.1.15 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.

49.1.16 Operator's certification authority

The operator creates a certification authority for itself. It creates and signs with the operator certification authority a leaf certificate for the webhook server to use. This certificate ensures safe communication between the Kubernetes API server and the operator.

49.1.17 Cluster's certification authority

The operator creates a certification authority for every PostgreSQL cluster. This certification authority is used to issue and renew TLS certificates for clients' authentication, including streaming replication standby servers (instead of passwords). Support for a custom certification authority for client certificates is available through secrets, which also includes integration with cert-manager. Certificates can be issued with the `cnpg` plugin for kubectl.

49.1.18 TLS connections

The operator transparently and natively supports TLS/SSL connections to encrypt client/server communications for increased security using the cluster's certification authority. Support for custom server certificates is available through secrets, which also includes integration with cert-manager.

49.1.19 Certificate authentication for streaming replication

To authorize streaming replication connections from the standby servers, the operator relies on TLS client certificate authentication. This method is used instead of relying on a password (and therefore a secret).

49.1.20 Continuous configuration management

The operator enables you to apply changes to the `Cluster` resource YAML section of the PostgreSQL configuration. Depending on the configuration option, it also makes sure that all instances are properly reloaded or restarted.

49.1.21 Import of existing PostgreSQL databases

The operator provides a declarative way to import existing Postgres databases in a new CloudNativePG cluster in Kubernetes, using offline migrations. The same feature also covers offline major upgrades of PostgreSQL databases. Offline means that applications must stop their write operations at the source until the database is imported. The feature extends the `initdb` bootstrap method to create a new PostgreSQL cluster using a logical snapshot of the data available in another PostgreSQL database. This data can be accessed by way of the network through a superuser connection. Import is from any supported version of Postgres. It relies on `pg_dump` and `pg_restore` being executed from the new cluster primary for all databases that are part of the operation and, if requested, for roles.

49.1.22 PostGIS clusters

CloudNativePG supports the installation of clusters with the *PostGIS Container Images*

open source extension for geographical databases. This extension is one of the most popular extensions for PostgreSQL.

49.1.23 Basic LDAP authentication for PostgreSQL

The operator allows you to configure LDAP authentication for your PostgreSQL clients, using either the *simple bind* or *search+bind* mode, as described in the LDAP authentication section of the PostgreSQL documentation .

49.1.24 Multiple installation methods

The operator can be installed through a Kubernetes manifest by way of `kubectl apply` , to be used in a traditional Kubernetes installation in public and private cloud environments. CloudNativePG also supports installation by way of a Helm chart or OLM bundle from OperatorHub.io.

49.1.25 Convention over configuration

The operator supports the convention-over-configuration paradigm, deciding standard default values while allowing you to override them and customize them. You can specify a deployment of a PostgreSQL cluster using the `Cluster` CRD in a couple of lines of YAML code.

49.2 Level 2: Seamless upgrades

Capability level 2 is about enabling updates of the operator and the actual workload, in this case PostgreSQL servers. This includes PostgreSQL minor release updates (security and bug fixes normally) as well as major online upgrades.

49.2.1 Operator Upgrade

Upgrading the operator is seamless and can be done as a new deployment. After upgrading the controller, a rolling update of all deployed PostgreSQL clusters is initiated. You can choose to update all clusters simultaneously or distribute their upgrades over time.

Thanks to the instance manager's injection, upgrading the operator does not require changes to the operand, allowing the operator to manage older versions of it.

Additionally, CloudNativePG supports *In-place updates of the instance manager*

following an operator upgrade. In-place updates do not require a rolling update or a subsequent switchover of the cluster.

49.2.2 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. This is normally initiated by security updates or Postgres minor version updates. In the presence of standby servers, the operator performs rolling updates starting from the replicas. It does this by dropping the existing pod and creating a new one with the new requested operand image that reuses the underlying storage. Depending on the value of the `primaryUpdateStrategy` , the operator proceeds with a switchover before updating the former primary (`unsupervised`). Or, it waits for the user to manually issue the switchover procedure (`supervised`) by way of the `cnpg` plugin for kubectl. The setting to use depends on the business requirements, as the operation might generate some downtime for the applications. This downtime can range from a few seconds to minutes, based on the actual database workload.

49.2.3 Offline In-Place Major Upgrades of PostgreSQL

CloudNativePG supports declarative offline in-place major upgrades when a new operand container image with a higher PostgreSQL major version is applied to a cluster. The upgrade can be triggered by updating the image tag via the `.spec.imageName` option or by using an image catalog to manage version changes. During the upgrade, all cluster pods are shut down to ensure data consistency. A new job is then created to validate the upgrade conditions, execute `pg_upgrade` , and create new directories for `PGDATA` , WAL files, and tablespaces if needed. Once the upgrade is complete, replicas are re-created. Failed upgrades can be rolled back.

49.2.4 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` , and `Upgrading Postgres major version` .

49.3 Level 3: Full lifecycle

Capability level 3 requires the operator to manage aspects of business continuity and scalability.

- Disaster recovery* is a business continuity component that requires that both backup and recovery of a database work correctly. While as a starting point, the goal is to achieve PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode} < 5 minutes, the long-term goal is to implement RPO=0 backup solutions. *High availability* is the other important component of business continuity. Through PostgreSQL native physical replication and hot standby replicas, it allows the operator to perform failover and switchover operations. This area includes enhancements in:
- Control of PostgreSQL physical replication, such as synchronous replication, (cascading) replication clusters, and so on
- Connection pooling, to improve performance and control through a connection pooling layer with pgBouncer

49.3.1 PostgreSQL WAL archive

The operator supports PostgreSQL continuous archiving of WAL files to an object store (AWS S3 and S3-compatible, Azure Blob Storage, Google Cloud Storage, and gateways like MinIO).

WAL archiving is defined at the cluster level, declaratively, through the `backup` parameter in the cluster definition. This is done by specifying an S3 protocol destination URL (for example, to point to a specific folder in an AWS S3 bucket) and, optionally, a generic endpoint URL.

WAL archiving, a prerequisite for continuous backup, doesn't require any further user action. The operator transparently sets the `archive_command` to rely on `barman-cloud-wal-archive` to ship WAL files to the defined endpoint. You can decide the compression algorithm, as well as the number of parallel jobs to concurrently upload WAL files in the archive. In addition, `Instance Manager` checks the correctness of the archive destination by performing the `barman-cloud-check-wal-archive` command before beginning to ship the first set of WAL files.

49.3.2 PostgreSQL Backups

CloudNativePG provides a pluggable interface (CNPG-I) for managing application-level backups using PostgreSQL's native physical backup mechanisms—namely base backups and continuous WAL archiving. This design enables flexibility and extensibility while ensuring consistency and performance.

The CloudNativePG Community officially supports the Barman Cloud Plugin , which enables continuous physical backups to object stores, along with full and Point-In-Time Recovery (PITR) capabilities.

In addition to CNPG-I plugins, CloudNativePG also natively supports backups using Kubernetes volume snapshots, when supported by the underlying storage class and CSI driver.

You can initiate base backups in two ways:

- On-demand, using the `Backup` custom resource
- Scheduled, using the `ScheduledBackup` custom resource, with a cron-like schedule format

Volume snapshots leverage the Kubernetes API and are particularly effective for very large databases (VLDBs) due to their speed and storage efficiency.

Both volume snapshots and CNPG-I-based backups support:

- Hot backups: Taken while PostgreSQL is running, ensuring minimal disruption.
- Cold backups: Performed by temporarily stopping PostgreSQL to ensure a fully consistent snapshot, when required.

49.3.3 Backups from a standby

The operator supports offloading base backups onto a standby without impacting the PgBouncerPoolMode {postgresql-cnpg-io-v1-PgBouncerPoolMode} of the database. This allows resources to be preserved on the primary, in particular I/O, for standard database operations.

49.3.4 Full restore from a backup

The operator enables you to bootstrap a new cluster (with its settings) starting from an existing and accessible backup, either on a volume snapshot, or in an object store, or via a plugin.

Once the bootstrap process is completed, the operator initiates the instance in recovery mode. It replays all available WAL files from the specified archive, exiting recovery and starting as a primary. Subsequently, the operator clones the requested number of standby instances from the primary. CloudNativePG supports parallel WAL fetching from the archive.

49.3.5 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 .

49.3.6 Zero-Data-Loss Clusters Through Synchronous Replication

Achieve *zero data loss* (RPO=0) in your local high-availability CloudNativePG cluster with support for both quorum-based and priority-based synchronous replication. The operator offers a flexible way to define the number of expected synchronous standby replicas available at any time, and allows customization of the `synchronous_standby_names` option as needed.

49.3.7 Replica clusters

Establish a robust cross-Kubernetes cluster topology for PostgreSQL clusters, harnessing the power of native streaming and cascading replication. With the `replica` option, you can configure an autonomous cluster to consistently replicate data from another PostgreSQL source of the same major version. This source can be located anywhere, provided you have access to a WAL archive for fetching WAL files or a direct streaming connection via TLS between the two endpoints.

Notably, the source PostgreSQL instance can exist outside the Kubernetes environment, whether in a physical or virtual setting.

Replica clusters can be instantiated through various methods, including volume snapshots, a recovery object store (using the Barman Cloud backup format), or streaming using `pg_basebackup`. Both WAL file shipping and WAL streaming are supported. The deployment of replica clusters significantly elevates the business continuity posture of PostgreSQL databases within Kubernetes, extending across multiple data centers and facilitating hybrid and multi-cloud setups. (While anticipating Kubernetes federation native capabilities, manual switchover across data centers remains necessary.)

Additionally, the flexibility extends to creating delayed replica clusters intentionally lagging behind the primary cluster. This intentional lag aims to minimize the Recovery Time Objective (*RTO and RPO impact*) in the event of unintended errors, such as incorrect `DELETE` or `UPDATE` SQL operations.

49.3.8 Distributed Database Topologies

Leverage replica clusters to define *Distributed Database Topologies*

for PostgreSQL that span across various Kubernetes clusters, facilitating hybrid and multi-cloud deployments. With CloudNativePG, you gain powerful capabilities, including:

- **Declarative Primary Control**: Easily specify which PostgreSQL cluster acts as the primary.
- **Seamless Primary Switchover**: Effortlessly demote the current primary and promote another PostgreSQL cluster, typically located in a different region, without needing to re-clone the former primary.

This setup can efficiently operate across two or more regions, can rely entirely on object stores for replication, and guarantees a maximum RPO (Recovery Point Objective) of 5 minutes. This advanced feature is uniquely provided by CloudNativePG, ensuring robust data integrity and continuity across diverse environments.

49.3.9 Tablespace support

CloudNativePG seamlessly integrates robust support for PostgreSQL tablespaces by facilitating the declarative definition of individual persistent volumes. This innovative feature empowers you to efficiently distribute I/O operations across a diverse array of storage devices. Through the transparent orchestration of tablespaces, CloudNativePG enhances the performance and scalability of PostgreSQL databases, ensuring a streamlined and optimized experience for managing large scale data storage in cloud-native environments. Support for temporary tablespaces is also included.

49.3.10 Customizable Startup, Liveness, and Readiness Probes

CloudNativePG configures startup, liveness, and readiness probes for PostgreSQL containers, which are managed by the Kubernetes kubelet. These probes interact with the `/startupz`, `/healthz`, and `/readyz` endpoints exposed by the instance manager's web server to monitor the Pod's health and readiness.

All probes are configured with default settings but can be fully customized to meet specific needs, allowing for fine-tuning to align with your environment and workloads.

For detailed configuration options and advanced usage, refer to the *Postgres instance manager* documentation.

49.3.11 Rolling deployments

The operator supports rolling deployments to minimize the downtime. If a PostgreSQL cluster is exposed publicly, the service load-balances the read-only traffic only to available pods during the initialization or the update.

49.3.12 Scale up and down of replicas

The operator allows you to scale up and down the number of instances in a PostgreSQL cluster. New replicas are started up from the primary server and participate in the cluster's HA infrastructure. The CRD declares a "scale" subresource that allows you to use the `kubectl scale` command.

49.3.13 Maintenance window and PodDisruptionBudget for Kubernetes nodes

The operator creates a `PodDisruptionBudget` resource to limit the number of concurrent disruptions to one primary instance. This configuration prevents the maintenance operation from deleting all the pods in a cluster, allowing the specified number of instances to be created. The PodDisruptionBudget is applied during the node-draining operation, preventing any disruption of the cluster service.

While this strategy is correct for Kubernetes clusters where storage is shared among all the worker nodes, it might not be the best solution for clusters using local storage or for clusters installed in a private cloud. The operator allows you to specify a maintenance window and configure the reaction to any underlying node eviction. The `ReusePVC` option in the maintenance window section enables to specify the strategy to use. Allocate new storage in a different PVC for the evicted instance, or wait for the underlying node to be available again.

49.3.14 Fencing

Fencing is the process of protecting the data in one, more, or even all instances of a PostgreSQL cluster when they appear to be malfunctioning. When an instance is fenced, the PostgreSQL server process is guaranteed to be shut down, while the pod is kept running. This ensures that, until the fence is lifted, data on the pod isn't modified by PostgreSQL and that you can investigate file system for debugging and troubleshooting purposes.

49.3.15 Hibernation

CloudNativePG supports *hibernation of a running PostgreSQL cluster*

in a declarative manner, through the `cnpg.io/hibernation` annotation. Hibernation enables saving CPU power by removing the database pods while keeping the database PVCs. This feature simulates scaling to 0 instances.

49.3.16 Reuse of persistent volumes storage in pods

When the operator needs to create a pod that was deleted by the user or was evicted by a Kubernetes maintenance operation, it reuses the `PersistentVolumeClaim`, if available. This ability avoids the need to clone the data from the primary again.

49.3.17 CPU and memory requests and limits

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

49.3.18 Connection pooling with PgBouncer

CloudNativePG provides native support for connection pooling with PgBouncerIntegrationStatus {postgresql-cnpg-io-v1-PgBouncerIntegrationStatus} , one of the most popular open source connection poolers for PostgreSQL. From an architectural point of view, the native implementation of a PgBouncer connection pooler introduces a new layer to access the database. This optimizes the query flow toward the instances and makes the use of the underlying PostgreSQL resources more efficient. Instead of connecting directly to a PostgreSQL service, applications can now connect to the PgBouncer service and start reusing any existing connection.

49.3.19 Logical Replication

CloudNativePG supports PostgreSQL's logical replication in a declarative manner using `Publication` and `Subscription` custom resource definitions.

Logical replication is particularly useful together with the import facility for online data migrations (even from public DBaaS solutions) and major PostgreSQL upgrades.

49.4 Level 4: Deep insights

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

CloudNativePG was designed to provide everything needed to easily integrate with industry-standard and community-accepted tools for flexible monitoring and logging.

49.4.1 Prometheus exporter with configurable queries

The instance manager provides a pluggable framework. By way of its own web server listening on the `metrics` port (9187), it exposes an endpoint to export metrics for the Prometheus monitoring and alerting tool. The operator supports custom monitoring queries defined as `ConfigMap` or `Secret` objects using a syntax that's compatible with postgres_exporter . CloudNativePG provides a set of basic monitoring queries for PostgreSQL that can be integrated and adapted to your context.

49.4.2 Grafana dashboard

CloudNativePG comes with a Grafana dashboard that you can use as a base to monitor all critical aspects of a PostgreSQL cluster, and customize.

49.4.3 Standard output logging of PostgreSQL error messages in JSON format

Every log message is delivered to standard output in JSON format. The first level is the definition of the timestamp, the log level, and the type of log entry, such as `postgres` for the canonical PostgreSQL error message channel. As a result, every pod managed by CloudNativePG can be easily and directly integrated with any downstream log processing stack that supports JSON as source data type.

49.4.4 Real-time query monitoring

CloudNativePG transparently and natively supports:

- The essential pg_stat_statements , which enables tracking of planning and execution statistics of all SQL statements executed by a PostgreSQL server
- The auto_explain , which provides a means for logging execution plans of slow statements automatically, without having to manually run `EXPLAIN` (helpful for tracking down un-optimized queries)
- The *pg_failover_slots* , which makes logical replication slots usable across a physical failover, ensuring resilience in change data capture (CDC) contexts based on PostgreSQL's native logical replication

49.4.5 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.

49.4.6 Kubernetes events

Record major events as expected by the Kubernetes API, such as creating resources, removing nodes, and upgrading. Events can be displayed by using the `kubectl describe` and `kubectl get events` commands.

49.5 Level 5: Auto pilot

Capability level 5 is focused on automated scaling, healing, and tuning through the discovery of anomalies and insights that emerged from the observability layer.

49.5.1 Automated failover for self-healing

In case of detected failure on the primary, the operator changes the status of the cluster by setting the most aligned replica as the new target primary. As a consequence, the instance manager in each alive pod initiates the required procedures to align itself with the requested status of the cluster. It does this by either becoming the new primary or by following it. In case the former primary comes back up, the same mechanism avoids a split-brain by preventing applications from reaching it, running `pg_rewind` on the server and restarting it as a standby.

49.5.2 Automated recreation of a standby

If the pod hosting a standby is removed, the operator initiates the procedure to re-create a standby server.

CHAPTER 50

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.

50.1 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.

50.2 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 standbyreplicas).

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.

50.3 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.

50.4 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 and Maintenance* section for details).

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

CHAPTER 51

Examples

The examples show configuration files for setting up your PostgreSQL cluster.

Important

These examples are for demonstration and experimentation purposes. You can execute them on a personal Kubernetes cluster with Minikube or Kind, as described in *Quick start* .

See also

For a list of available options, see *API Reference* .

51.1 Basics

Basic cluster :

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

  storage:
    size: 1Gi
```

A basic example of a cluster.

Custom cluster :

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

  # Parameters and pg_hba configuration will be append
  # to the default ones to make the cluster work
  postgresql:
    parameters:
      max_worker_processes: "60"
    pg_hba:
      # To access through TCP/IP you will need to get username
      # and password from the secret cluster-example-custom-app
      - host all all all md5

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Require 1Gi of space per instance using default storage class
  storage:
    size: 1Gi
```

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

Cluster with customized storage class :

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

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi
```

: A basic cluster that uses a specified storage class of `standard` .

Cluster with persistent volume claim (PVC) template configured :

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgresql-pvc-template
spec:
  instances: 3

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    size: 1Gi
    pvcTemplate:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage: 1Gi
      storageClassName: standard
      volumeMode: Filesystem
```

: A basic cluster with an explicit persistent volume claim template.

Extended configuration example :

```yaml
#  Example of definition of a test cluster using all the elements available

#  in the CRD. Please change values appropriately for your environment.

#  Remember that you can take advantage of convention over configuration

#  and normally you dont need to use all these definitions.

apiVersion: v1
data:
  password:␣
↪VHhWZVE0bk44MlNTaVlIb3N3cU9VUlp2UURhTDRLcE5FbHNDRUVlOWJ3RHhNZDczS2NrSWVYelM1Y1U2TGlDMg==
  username: YXBw
kind: Secret
metadata:
  name: cluster-example-app-user
type: kubernetes.io/basic-auth
- --
apiVersion: v1
data:
  password:␣
↪dU4zaTFIaDBiWWJDYzRUeVZBYWNCaG1TemdxdHpxeG1PVmpBbjBRSUNoc0pyU211OVBZMmZ3MnE4RUtLTHBaOQ==
  username: cG9zdGdyZXM=
```

(continues on next page)

(continued from previous page)

```
kind: Secret
metadata:
  name: cluster-example-superuser
type: kubernetes.io/basic-auth
- --
apiVersion: v1
kind: Secret
metadata:
  name: backup-creds
data:
  ACCESS_KEY_ID: a2V5X2lk
  ACCESS_SECRET_KEY: c2VjcmV0X2tleQ==
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-full
spec:
  description: "Example of cluster"
  imageName: ghcr.io/cloudnative-pg/postgresql:17.5
  # imagePullSecret is only required if the images are located in a private registry
  # imagePullSecrets:
  #   - name: private_registry_access
  instances: 3
  startDelay: 300
  stopDelay: 300
  primaryUpdateStrategy: unsupervised

  postgresql:
    parameters:
      shared_buffers: 256MB
      pg_stat_statements.max: 10000
      pg_stat_statements.track: all
      auto_explain.log_min_duration: 10s
    pg_hba:
      - host all all 10.244.0.0/16 md5

  bootstrap:
    initdb:
      database: app
      owner: app
      secret:
        name: cluster-example-app-user
    # Alternative bootstrap method: start from a backup
    #recovery:
    #  backup:
    #    name: backup-example

  enableSuperuserAccess: true
  superuserSecret:
    name: cluster-example-superuser
```

(continues on next page)

(continued from previous page)

```
  storage:
    storageClass: standard
    size: 1Gi

  backup:
    barmanObjectStore:
      destinationPath: s3://cluster-example-full-backup/
      endpointURL: http://custom-endpoint:1234
      s3Credentials:
        accessKeyId:
          name: backup-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: backup-creds
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
        encryption: AES256
      data:
        compression: gzip
        encryption: AES256
        immediateCheckpoint: false
        jobs: 2
    retentionPolicy: "30d"

  resources:
    requests:
      memory: "512Mi"
      cpu: "1"
    limits:
      memory: "1Gi"
      cpu: "2"

  affinity:
    enablePodAntiAffinity: true
    topologyKey: failure-domain.beta.kubernetes.io/zone

  nodeMaintenanceWindow:
    inProgress: false
    reusePVC: false
```

: A cluster that sets most of the available options.

Bootstrap cluster with SQL files :

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: post-init-sql-configmap
data:
  configmap.sql: |
    create table configmaps (i integer);
    insert into configmaps (select generate_series(1,10000));
```

(continues on next page)

(continued from previous page)

```
- --
apiVersion: v1
kind: Secret
metadata:
  name: post-init-sql-secret
stringData:
  secret.sql: |
    create table secrets (i integer);
    insert into secrets (select generate_series(1,10000));
- --
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    initdb:
      database: appdb
      owner: appuser
      postInitSQL:
        - create table numbers (i integer)
        - insert into numbers (select generate_series(1,10000))
      postInitTemplateSQL:
        - create extension intarray
      postInitApplicationSQL:
        - create table application_numbers (i integer)
        - insert into application_numbers (select generate_series(1,10000))
      postInitApplicationSQLRefs:
        configMapRefs:
        - name: post-init-sql-configmap
          key: configmap.sql
        secretRefs:
        - name: post-init-sql-secret
          key: secret.sql

  storage:
    size: 1Gi
```

: A cluster example that executes a set of queries defined in a secret and a `ConfigMap` right after the database is created.

Sample cluster with customized ``pg_hba`` configuration :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  postgresql:
    pg_hba:
      - hostssl app all all cert
```

(continues on next page)

(continued from previous page)

```
  storage:
    size: 1Gi
```

: A basic cluster that enables the user app to authenticate using certificates.

Sample cluster with Secret and ConfigMap mounted using projected volume template :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-projected-volume
spec:
  instances: 3
  projectedVolumeTemplate:
    sources:
      - secret:
          name: sample-secret
          items:
            - key: tls.crt
              path: certificate/tls.crt
            - key: tls.key
              path: certificate/tls.key
      - configMap:
          name: sample-configmap
          items:
            - key: key1
              path: config/key1
            - key: key2
              path: config/key2
  storage:
    size: 1Gi

- --
apiVersion: v1
data:
  tls.crt:␣
↪LS0tLS1CRUdJTiBDRVJUSUZJQ0FURS0tLS0tCk1JSUNQekNDQWVXZ0F3SUJBZ0lRSHVDY2lKcDVkZis0dHZBcDBrTk9rekFLQmo
  tls.key:␣
↪LS0tLS1CRUdJTiBFQyBQUklWQVRFIEtFWS0tLS0tCk1IY0NBUUVFSUtucWxVQjFqU2sxWW5VZzAyb0tGbXlRdDJLUEZwaFc0K1l
kind: Secret
metadata:
  name: sample-secret
type: kubernetes.io/tls
- --
apiVersion: v1
data:
  key1: value1
  key2: value2
  key3: value3
kind: ConfigMap
metadata:
  name: sample-configmap
```

A basic cluster with the existing `Secret` and `ConfigMap` mounted into Postgres pod using projected volume mount.

51.2 Backups

Customized storage class and backups : *Prerequisites*: Bucket storage must be available. The sample config is for AWS. Change it to suit your setup. :

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

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi

  # Backup properties
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/path/to/folder
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

A cluster with backups configured.

Backup : *Prerequisites*:

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

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
```

(continues on next page)

(continued from previous page)

```
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi

  # Backup properties
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/path/to/folder
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

applied and healthy. :

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

: An example of a backup that runs against the previous sample.

Simple cluster with backup configured for minio : *Prerequisites*: The configuration assumes minio is running and working. Update `backup.barmanObjectStore` with your minio parameters or your cloud solution. :

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

  # Persistent storage configuration
  storage:
    storageClass: csi-hostpath-sc
    size: 1Gi

  # Backup properties
```

(continues on next page)

(continued from previous page)

```
  # This assumes a local minio setup
  backup:
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio:9000
      s3Credentials:
        accessKeyId:
          name: minio
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
      data:
        additionalCommandArgs:
          - "--min-chunk-size=5MB"
          - "--read-timeout=60"
          - "-vv"
```

A basic cluster with backups configured.

Simple cluster with backup configured for Scaleway Object Storage : *Prerequisites*: The configuration assumes a Scaleway Object Storage bucket exists. Update `backup.barmanObjectStore` with your Scaleway parameters. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg-backup-scaleway
spec:
  instances: 3
  storage:
    storageClass: standard
    size: 1Gi
  backup:
    barmanObjectStore:
      destinationPath: "s3://<bucket>/backups/"     # change <bucket> with your␣
↪buckets name.
      endpointURL: "https://s3.<region>.scw.cloud"  # change <region> with your␣
↪buckets location/region.
      s3Credentials:
        accessKeyId:
          name: scaleway
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: scaleway
          key: ACCESS_SECRET_KEY
        region:
          name: scaleway
          key: ACCESS_REGION
```

A basic cluster with backups configured to work with Scaleway Object Storage..

51.3 Replica clusters

Replica cluster by way of backup from an object store : *Prerequisites*:

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

  # Example of rolling update strategy:
  # - unsupervised: automated update of the primary once all
  #                 replicas have been upgraded (default)
  # - supervised: requires manual supervision to perform
  #               the switchover of the primary
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: standard
    size: 1Gi

  # Backup properties
  backup:
    barmanObjectStore:
      destinationPath: s3://BUCKET_NAME/path/to/folder
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

applied and healthy, and a backup

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: cluster-example-trigger-backup
spec:
  cluster:
    name: cluster-example-with-backup
```

applied and completed. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-replica-from-backup-simple
spec:
```

(continues on next page)

(continued from previous page)

```
  instances: 1

  bootstrap:
    recovery:
      source: cluster-example-backup

  replica:
    enabled: true
    source: cluster-example-backup

  storage:
    size: 1Gi

  externalClusters:
  - name: cluster-example-backup
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio:9000
      s3Credentials:
        accessKeyId:
          name: minio
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio
          key: ACCESS_SECRET_KEY
```

: A replica cluster following a cluster with backup configured.

Replica cluster by way of volume snapshot : *Prerequisites*:

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-volume-snapshot
spec:
  instances: 3
  primaryUpdateStrategy: unsupervised

  # Persistent storage configuration
  storage:
    storageClass: csi-hostpath-sc
    size: 1Gi
  walStorage:
    storageClass: csi-hostpath-sc
    size: 1Gi

  # Backup properties
  backup:
    volumeSnapshot:
       className: csi-hostpath-snapclass
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio:9000
```

(continues on next page)

(continued from previous page)

```
      s3Credentials:
        accessKeyId:
          name: minio
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
```

applied and healthy, and a volume snapshot

```
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: backup-with-volume-snapshot
spec:
  method: volumeSnapshot
  cluster:
    name: cluster-example-with-volume-snapshot
```

applied and completed. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-replica-from-snapshot
spec:
  instances: 1

  storage:
    storageClass: csi-hostpath-sc
    size: 1Gi
  walStorage:
    storageClass: csi-hostpath-sc
    size: 1Gi

  bootstrap:
    recovery:
      source: cluster-example-with-volume-snapshot
      volumeSnapshots:
        storage:
          name: cluster-example-with-volume-snapshot-2-1692618163
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io
        walStorage:
          name: cluster-example-with-volume-snapshot-2-wal-1692618163
          kind: VolumeSnapshot
          apiGroup: snapshot.storage.k8s.io

  replica:
    enabled: true
```

(continues on next page)

(continued from previous page)

```
    source: cluster-example-with-volume-snapshot

  externalClusters:
    - name: cluster-example-with-volume-snapshot

      connectionParameters:
        host: cluster-example-with-volume-snapshot-rw.default.svc
        user: postgres
        dbname: postgres
      password:
        name: cluster-example-with-volume-snapshot-superuser
        key: password

      barmanObjectStore:
        destinationPath: s3://backups/
        endpointURL: http://minio:9000
        s3Credentials:
          accessKeyId:
            name: minio
            key: ACCESS_KEY_ID
          secretAccessKey:
            name: minio
            key: ACCESS_SECRET_KEY
        wal:
          maxParallel: 8
```

: A replica cluster following a cluster with volume snapshot configured.

Replica cluster by way of streaming (pg_basebackup) : *Prerequisites*:

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

  storage:
    size: 1Gi
```

applied and healthy. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-replica-example
spec:
  instances: 1

  bootstrap:
    pg_basebackup:
      source: cluster-example
```

(continues on next page)

(continued from previous page)

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

  storage:
    size: 1Gi
  # note the namespace default in the host name `cluster-example-rw.default.svc`
  # remember to change accordingly with the namespace of the main cluster
  externalClusters:
  - name: cluster-example
    connectionParameters:
      host: cluster-example-rw.default.svc
      user: streaming_replica
      sslmode: verify-full
      dbname: postgres
    # NOTE: if this cluster is created in a different namespace than the main cluster
    # remember to create the `-replication` and `-ca` secrets in the follower␣
↪namespace
    # before creating the follower cluster
    sslKey:
      name: cluster-example-replication
      key: tls.key
    sslCert:
      name: cluster-example-replication
      key: tls.crt
    sslRootCert:
      name: cluster-example-ca
      key: ca.crt
```

: A replica cluster following `cluster-example` with streaming replication.

51.4 PostGIS

PostGIS example :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgis-example
spec:
  instances: 1
  imageName: ghcr.io/cloudnative-pg/postgis:17
  storage:
    size: 1Gi
  postgresql:
    parameters:
      log_statement: ddl
- --
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
```

(continues on next page)

(continued from previous page)

```
  name: postgis-example-app
spec:
  name: app
  owner: app
  cluster:
    name: postgis-example
  extensions:
  - name: postgis
  - name: postgis_topology
  - name: fuzzystrmatch
  - name: postgis_tiger_geocoder
```

: An example of a PostGIS cluster. See *PostGIS Container Images* for details.

51.5 Managed roles

Cluster with declarative role management :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-with-roles
spec:
  instances: 3
  storage:
    size: 1Gi

  managed:
    roles:
    - name: app
      createdb: true
      login: true
    - name: dante
      ensure: present
      comment: my database-side comment
      login: true
      superuser: false
      createdb: true
      createrole: false
      inherit: false
      replication: false
      bypassrls: false
      connectionLimit: 4
      validUntil: "2053-04-12T15:04:05Z"
      inRoles:
        - pg_monitor
        - pg_signal_backend
      passwordSecret:
        name: cluster-example-dante
- --
apiVersion: v1
```

(continues on next page)

(continued from previous page)

```
data:
  username: ZGFudGU=
  password: ZGFudGU=
kind: Secret
metadata:
  name: cluster-example-dante
type: kubernetes.io/basic-auth
```

: Declares a role with the `managed` stanza. Includes password management with Kubernetes secrets.

51.6 Managed services

Cluster with managed services :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-managed-services
spec:
  instances: 1
  storage:
    size: 1Gi

  managed:
    services:
      ## disable the default services
      disabledDefaultServices: ["ro", "r"]
      additional:
        - selectorType: rw
          serviceTemplate:
            metadata:
              name: "test-rw"
              labels:
                test-label: "true"
              annotations:
                test-annotation: "true"
            spec:
              type: LoadBalancer
```

: Declares a service with the `managed` stanza. Includes default service disabled and new `rw` service template of `LoadBalancer` type defined.

51.7 Declarative tablespaces

Cluster with declarative tablespaces :

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

(continues on next page)

(continued from previous page)

```
  instances: 3

  storage:
    size: 1Gi

  tablespaces:
    - name: atablespace
      storage:
        size: 1Gi
        storageClass: standard
      temporary: true
    - name: another_tablespace
      storage:
        size: 2Gi
        storageClass: standard
      temporary: true
    - name: tablespacea1
      storage:
        size: 2Gi
        storageClass: standard
```

Cluster with declarative tablespaces and backup : *Prerequisites*: The configuration assumes minio is running and working. Update `backup.barmanObjectStore` with your minio parameters or your cloud solution. :

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

  storage:
    size: 1Gi

  backup:
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio:9000
      s3Credentials:
        accessKeyId:
          name: minio
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip

  tablespaces:
    - name: atablespace
      storage:
        size: 1Gi
```

(continues on next page)

(continued from previous page)

```
        storageClass: standard
    - name: another_tablespace
      storage:
        size: 2Gi
        storageClass: standard
    - name: tablespacea1
      storage:
        size: 2Gi
        storageClass: standard
```

Restored cluster with tablespaces from object store : *Prerequisites*: The previous cluster applied and a base backup completed. Remember to update `bootstrap.recovery.backup.name` with the backup name. :

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

  storage:
    size: 1Gi

  bootstrap:
    recovery:
      backup:
        name: cluster-example-with-tablespaces-20231128093940

  tablespaces:
    atablespace:
      storage:
        size: 1Gi
        storageClass: standard
    another_tablespace:
      storage:
        size: 2Gi
        storageClass: standard
    tablespacea1:
      storage:
        size: 2Gi
        storageClass: standard
```

For a list of available options, see *API Reference* .

51.8 Pooler configuration

Pooler with custom service config :

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

(continues on next page)

(continued from previous page)

```
spec:
  cluster:
    name: cluster-example
  instances: 3
  type: rw
  serviceTemplate:
    metadata:
      labels:
        app: pooler
    spec:
      type: LoadBalancer
  pgbouncer:
    poolMode: session
    parameters:
      max_client_conn: "1000"
      default_pool_size: "10"
```

51.9 Logical replication via declarative Publication and Subscription objects

Two test manifests contain everything needed to set up logical replication:

Source cluster with a publication :

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

  imageName: ghcr.io/cloudnative-pg/postgresql:17

  storage:
    size: 1Gi

  bootstrap:
    initdb:
      postInitApplicationSQL:
        - CREATE TABLE numbers (i SERIAL PRIMARY KEY, m INTEGER)
        - INSERT INTO numbers (m) (SELECT generate_series(1,10000))
        - ALTER TABLE numbers OWNER TO app
        - CREATE TABLE numbers_two (i SERIAL PRIMARY KEY, m INTEGER)
        - INSERT INTO numbers_two (m) (SELECT generate_series(1,10000))
        - ALTER TABLE numbers_two OWNER TO app
        - CREATE SCHEMA another_schema
        - ALTER SCHEMA another_schema OWNER TO app
        - CREATE TABLE another_schema.numbers_three (i SERIAL PRIMARY KEY, m INTEGER)
        - INSERT INTO another_schema.numbers_three (m) (SELECT generate_series(1,
↪10000))
        - ALTER TABLE another_schema.numbers_three OWNER TO app
```

(continues on next page)

(continued from previous page)

```
  replicationSlots:
    highAvailability:
      synchronizeLogicalDecoding: true

  managed:
    roles:
      - name: app
        login: true
        replication: true

  postgresql:
    parameters:
      hot_standby_feedback: on
      sync_replication_slots: on

- --
apiVersion: postgresql.cnpg.io/v1
kind: Publication
metadata:
  name: cluster-example-pub
spec:
  name: pub
  dbname: app
  cluster:
    name: cluster-example
  target:
    allTables: true
```

Sets up a cluster, `cluster-example` with some tables created in the `app` database, and, importantly, *adds replication to the app user*. A publication is created for the cluster on the `app` database: note that the publication will be reconciled only after the cluster's primary is up and running.

Destination cluster with a subscription : *Prerequisites*: The source cluster with publication, defined as above. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-dest
spec:
  instances: 1

  storage:
    size: 1Gi

  bootstrap:
    initdb:
      import:
        type: microservice
        schemaOnly: true
        databases:
          - app
        source:
```

(continues on next page)

(continued from previous page)

```
          externalCluster: cluster-example

  externalClusters:
  - name: cluster-example
    connectionParameters:
      host: cluster-example-rw.default.svc
      user: app
      dbname: app
    password:
      name: cluster-example-app
      key: password
- --
apiVersion: postgresql.cnpg.io/v1
kind: Subscription
metadata:
  name: cluster-example-dest-sub
spec:
  cluster:
    name: cluster-example-dest
  name: sub
  dbname: app
  publicationName: pub
  externalClusterName: cluster-example
  parameters:
    failover: true
```

Sets up a cluster `cluster-example-dest` with:

- the source cluster defined in the `externalClusters` stanza. Note that it uses the `app` role to connect, which assumes the source cluster grants it `replication` privilege.
- a bootstrap import of microservice type, with `schemaOnly` enabled

A subscription is created on the destination cluster: note that the subscription will be reconciled only after the destination cluster's primary is up and running.

After both clusters have been reconciled, together with the publication and subscription objects, you can verify that that tables in the source cluster, and the data in them, have been replicated in the destination cluster

In addition, there are some standalone example manifests:

A plain Publication targeting All Tables : *Prerequisites*: an existing cluster `cluster-example` . :

```
apiVersion: postgresql.cnpg.io/v1
kind: Publication
metadata:
  name: publication-example
spec:
  cluster:
    name: cluster-example
  name: pub-all
  dbname: app
  target:
    allTables: true
```

A Publication with a constrained publication target : *Prerequisites*: an existing cluster `cluster-example` . :

```
apiVersion: postgresql.cnpg.io/v1
kind: Publication
metadata:
  name: publication-example-objects
spec:
  cluster:
    name: cluster-example
  name: pub-objects
  dbname: app
  target:
    objects:
      - tablesInSchema: public
      - table:
          schema: another_schema
          name: numbers_three
          only: true
```

A plain Subscription : Prerequisites: an existing cluster `cluster-example` set up as source, with a publication `pub-all` . A cluster `cluster-example-dest` set up as a destination cluster, including the `externalClusters` stanza with connection parameters to the source cluster, including a role with replication privilege. :

```
apiVersion: postgresql.cnpg.io/v1
kind: Subscription
metadata:
  name: subscription-sample
spec:
  name: sub
  dbname: app
  publicationName: pub-all
  cluster:
    name: cluster-example-dest
  externalClusterName: cluster-example
```

All the above manifests create publications or subscriptions on the `app` database. The Database CRD offers a convenient way to create databases declaratively. With it, logical replication could be set up for arbitrary databases. Which brings us to the next section.

51.10 Declarative management of Postgres databases

A plain Database : *Prerequisites*: an existing cluster `cluster-example` . :

```
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: db-one
spec:
  name: one
  owner: app
  cluster:
    name: cluster-example
```

A Database with ICU local specifications : *Prerequisites*: an existing cluster `cluster-example` running Postgres 16 or more advanced. :

```
#  NOTE: this manifest will only work properly if the Postgres version supports

#  ICU locales and rules (version 16 and newer)

apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: db-icu
spec:
  name: declarative-icu
  owner: app
  encoding: UTF8
  localeProvider: icu
  icuLocale: en
  icuRules: fr
  template: template0
  cluster:
    name: cluster-example
```

CHAPTER 52

Networking

CloudNativePG assumes the underlying Kubernetes cluster has the required connectivity already set up. Networking on Kubernetes is an important and extended topic; please refer to the *Kubernetes documentation* for further information.

If you're following the quickstart guide to install CloudNativePG on a local KinD or K3d cluster, you should not encounter any networking issues as neither platform will add any networking restrictions by default.

However, when deploying CloudNativePG on existing infrastructure, networking restrictions might be in place that could impair the communication of the operator with PostgreSQL clusters. Specifically, existing Network Policies

might restrict certain types of traffic.

Or, you might be interested in adding network policies in your environment for increased security. As mentioned in the *security document* , please ensure the operator can reach every cluster pod on ports 8000 and 5432, and that pods can connect to each other.

52.1 Cross-namespace network policy for the operator

Following the quickstart guide or using helm chart for deployment will install the operator in a dedicated namespace (`cnpg-system` by default). We recommend that you create clusters in a different namespace.

The operator *must* be able to connect to cluster pods. This might be precluded if there is a `NetworkPolicy` restricting cross-namespace traffic.

For example, the *kubernetes guide on network policies*

contains an example policy denying all ingress traffic by default.

If your local kubernetes setup has this kind of restrictive network policy, you will need to create a `NetworkPolicy` to explicitly allow connection from the operator namespace and pod to the cluster namespace and pods. You can find an example in the file in this repository. Please note, you'll need to adjust the cluster name and cluster namespace to match your specific setup, and also the operator namespace if it is not the default namespace.

```
#  This network policy allows ingress from the CNPG operator

#  installed in namespace cnpg-system into the pods corresponding
```

(continues on next page)

(continued from previous page)

```
#  to a cluster named cluster-example

apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-operator
spec:
  podSelector:
    matchLabels:
      cnpg.io/cluster: cluster-example # The label value must be the cluster name
  ingress:
    - from:
        - namespaceSelector:
            matchLabels:
              kubernetes.io/metadata.name: cnpg-system # Namespace where the operator␣
↪is deployed
          podSelector:
            matchLabels:
              app.kubernetes.io/name: cloudnative-pg # Matches the Operator pod
      ports:
        - port: 8000
        - port: 5432
```

52.2 Cross-cluster networking

While *Bootstrapping a Replica Cluster* from another cluster or when using the `externalClusters` section, ensure connectivity among all clusters, object stores, and namespaces involved.

Again, we refer you to the *Kubernetes documentation*

for setup information.

CHAPTER 53

Benchmarking

The CNPG kubectl plugin provides an easy way for benchmarking a PostgreSQL deployment in Kubernetes using CloudNativePG.

Benchmarking is focused on two aspects:

- the **database**, by relying on pgbench
- the **storage**, by relying on fio

> **Important**
>
> *pgbench* and *fio* must be run in a staging or pre-production environment. Do not use these plugins in a production environment, as it might have catastrophic consequences on your databases and the other workloads/applications that run in the same shared environment.

53.1 pgbench

The `kubectl` CNPG plugin command `pgbench` executes a user-defined `pgbench` job against an existing Postgres Cluster.

Through the `--dry-run` flag you can generate the manifest of the job for later modification/execution.

A common command structure with `pgbench` is the following:

```
kubectl cnpg pgbench \
  -n <namespace> <cluster-name> \
  --job-name <pgbench-job> \
  --db-name <db-name> \
  -- <pgbench options>
```

Important

Please refer to the pgbench

for information about the specific options to be used in your jobs.

This example creates a job called `pgbench-init` that initializes for `pgbench` OLTP-like purposes the `app` database in a `Cluster` named `cluster-example` , using a scale factor of 1000:

```
kubectl cnpg pgbench \
  --job-name pgbench-init \
  cluster-example \
  -- --initialize --scale 1000
```

Note

This will generate a database with 100000000 records, taking approximately 13GB of space on disk.

You can see the progress of the job with:

```
kubectl logs jobs/pgbench-run
```

The following example creates a job called `pgbench-run` executing `pgbench` against the previously initialized database for 30 seconds, using a single connection:

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

The next example runs `pgbench` against an existing database by using the `--db-name` flag and the `pgbench` namespace:

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

By default, jobs do not expire. You can enable automatic deletion with the `--ttl` flag. The job will be deleted after the specified duration (in seconds).

```
kubectl cnpg pgbench \
  --job-name pgbench-run \
  --ttl 600 \
  cluster-example \
  -- --time 30 --client 1 --jobs 1
```

If you want to run a `pgbench` job on a specific worker node, you can use the `--node-selector` option. Suppose you want to run the previous initialization job on a node having the `workload=pgbench` label, you can run:

```
kubectl cnpg pgbench \
  --db-name pgbench \
  --job-name pgbench-init \
  --node-selector workload=pgbench \
  cluster-example \
  -- --initialize --scale 1000
```

The job status can be fetched by running:

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

NAME       COMPLETIONS   DURATION   AGE
job-name   1/1           15s        41s
```

Once the job is completed the results can be gathered by executing:

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

53.2 fio

The kubectl CNPG plugin command `fio` executes a fio job with default values and read operations. Through the `--dry-run` flag you can generate the manifest of the job for later modification/execution.

Note

The kubectl plugin command *fio* will create a deployment with predefined fio job values using a ConfigMap. If you want to provide custom job values, we recommend generating a manifest using the *--dry-run* flag and providing your custom job values in the generated ConfigMap.

Example of default usage:

```
kubectl cnpg fio <fio-name>
```

Example with custom values:

```
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
```

(continues on next page)

(continued from previous page)

```
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 image for that.

The Pod created by the deployment will be ready when it starts serving the results. You can forward the port of the pod created by the deployment

```
kubectl port-forward -n <namespace> deployment/<fio-name> 8000
```

and then use a browser and connect to `http://localhost:8000/` to get the data.

The default 8k block size has been chosen to emulate a PostgreSQL workload. Disks that cap the amount of available IOPS can show very different throughput values when changing this parameter.

Below is an example diagram of sequential writes on a local disk mounted on a dedicated Kubernetes node (1 hour benchmark):

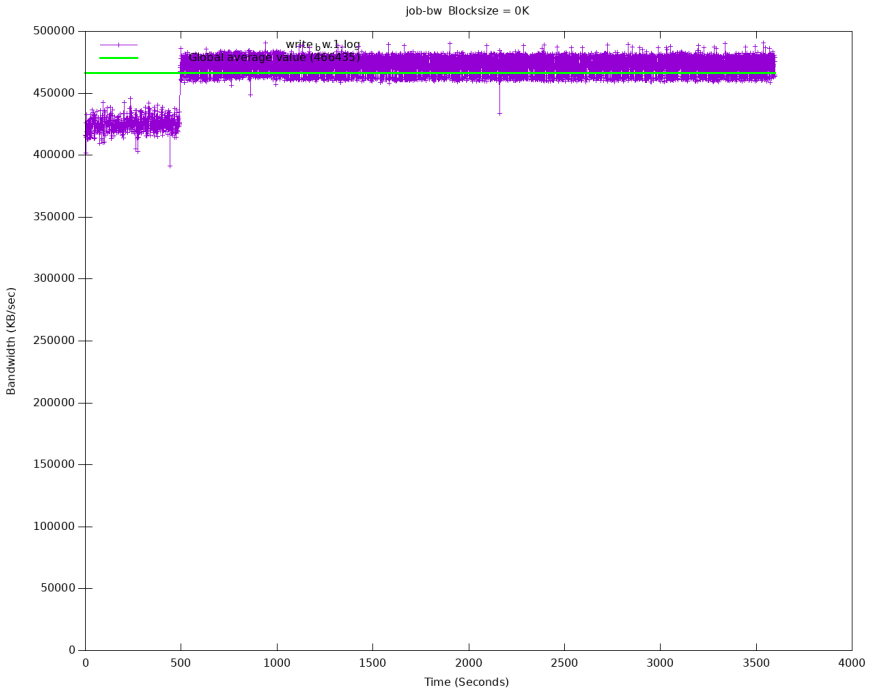


Fig. 1: Sequential writes bandwidth

After all testing is done, fio deployment and resources can be deleted by:

```
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.

CHAPTER 54

Frequently Asked Questions (FAQ)

54.1 Running PostgreSQL in Kubernetes

Everyone knows that stateful workloads like PostgreSQL cannot run in Kubernetes. Why do you say the contrary?

An *independent research survey commissioned by the Data on KubernetesCommunity* in September 2021 revealed that half of the respondents run most of their production workloads on Kubernetes. 90% of them believe that Kubernetes is ready for stateful workloads, and 70% of them run databases in production. Databases like Postgres. However, according to them, significant challenges remain, such as the knowledge gap (Kubernetes and Cloud Native, in general, have a steep learning curve) and the quality of Kubernetes operators. The latter is the reason why we believe that an operator like CloudNativePG highly contributes to the success of your project.

For database fanatics like us, a real game-changer has been the introduction of the support for local persistent volumes in *Kubernetes 1.14 in April 2019* .

CloudNativePG is built on immutable application containers. What does it mean?

According to the microservice architectural pattern, a container is designed to run a single application or process. As a result, such container images are built to run the main application as the single entry point (the so-called PID 1 process).

In Kubernetes terms, the application is referred to as workload. Workloads can be stateless like a web application server or stateful like a database. Mapping this concept to PostgreSQL, an immutable application container is a single "postgres" process that is running and tied to a single and specific version - the one in the immutable container image.

No other processes such as SSH or systemd, or syslog are allowed.

Immutable Application Containers are in contrast with Mutable System Containers, which are still a very common way to interpret and use containers.

Immutable means that a container won't be modified during its life: no updates, no patches, no configuration changes. If you must update the application code or apply a patch, you build a new image and redeploy it. Immutability makes deployments safer and more repeatable.

For more information, please refer to *"Why EDB chose immutable application containers"* .

What does Cloud Native mean?

The Cloud Native Computing Foundation defines the term " *Cloud Native* ". However, since the start of the Cloud Native PostgreSQL/CloudNativePG operator at 2ndQuadrant, the development team has been interpreting Cloud Native as three main concepts:

1. An existing, healthy, genuine, and prosperous DevOps culture, founded on people, as well as principles and processes, which enables teams and organizations (as teams of teams) to continuously change so to innovate and accelerate the delivery of outcomes and produce value for the business in safer, more efficient, and more engaging ways
2. A microservice architecture that is based on Immutable Application Containers
3. A way to manage and orchestrate these containers, such as Kubernetes

Currently, the standard de facto for container orchestration is Kubernetes, which automates the deployment, administration and scalability of Cloud Native Applications.

Another definition of Cloud Native that resonates with us is the one defined by Ibryam and Huß in *"Kubernetes Patterns", published by O'Reilly* :

> Principles, Patterns, Tools to automate containerized microservices at scale

Can I run CloudNativePG on bare metal Kubernetes?

Yes, definitely. You can run Kubernetes on bare metal. And you can dedicate one or more physical worker nodes with locally attached storage to PostgreSQL workloads for maximum and predictable I/O performance.

The actual Cloud Native PostgreSQL project, from which CloudNativePG originated, was born after a pilot project in 2019 that benchmarked storage and PostgreSQL on the same bare metal server, first directly in Linux, and then inside Kubernetes. As expected, the experiment showed only negligible performance impact introduced by the container running in Kubernetes through local persistent volumes, allowing the Cloud Native initiative to continue.

Why should I use PostgreSQL replication instead of file system replication?

Please read the *Architecture: Synchronizing the state*

section.

Why should I use an operator instead of running PostgreSQL as a container?

The most basic approach to running PostgreSQL in Kubernetes is to have a pod, which is the smallest unit of deployment in Kubernetes, running a Postgres container with no replica. The volume hosting the Postgres data directory is mounted on the pod, and it usually resides on network storage. In this case, Kubernetes restarts the pod in case of a problem or moves it to another Kubernetes node.

The most sophisticated approach is to run PostgreSQL using an operator. An operator is an extension of the Kubernetes controller and defines how a complex application works in business continuity contexts. The operator pattern is currently state of the art in Kubernetes for this purpose. An operator simulates the work of a human operator in an automated and programmatic way.

Postgres is a complex application, and an operator not only needs to deploy a cluster (the first step), but also properly react after unexpected events. The typical example is that of a failover.

An operator relies on Kubernetes for capabilities like self-healing, scalability, replication, high availability, backup, recovery, updates, access, resource control, storage management, and so on. It also facilitates the integration of a PostgreSQL cluster in the log management and monitoring infrastructure.

CloudNativePG enables the definition of the desired state of a PostgreSQL cluster via declarative configuration. Kubernetes continuously makes sure that the current state of the infrastructure matches the desired one through reconciliation loops initiated by the Kubernetes controller. If the desired state and the actual state don't match, reconciliation loops trigger self-healing procedures. That's where an operator like CloudNativePG comes into play.

Are there any other operators for Postgres out there?

Yes, of course. And our advice is that you look at all of them and compare them with CloudNativePG before making your decision. You will see that most of these operators use an external failover management tool (Patroni or similar) and rely on StatefulSets.

Here is a non exhaustive list, in chronological order from their publication on GitHub:

- Crunchy Data Postgres Operator (2017)
- Zalando Postgres Operator (2017)
- Stackgres (2020)
- Percona Operator for PostgreSQL (2021)
- Kubegres (2021)

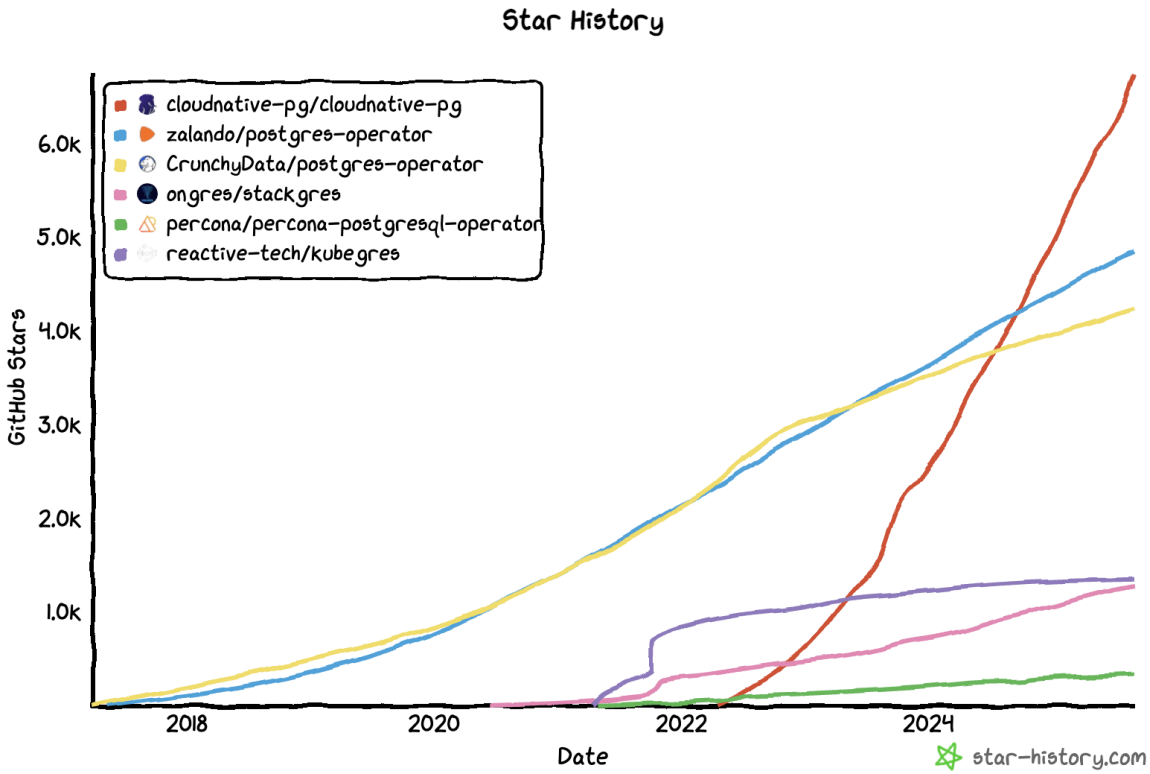


Fig. 1: star history

Feel free to report any relevant missing entry as a PR.

Note

The Data on Kubernetes Community

(which includes some of our maintainers) is working on an independent and vendor neutral project to list the operators called Operator Feature Matrix .

You say that CloudNativePG is a fully declarative operator. What do you mean by that?

The easiest way is to explain declarative configuration through an example that highlights the differences with imperative configuration. In an imperative context, the state is defined as a series of tasks to be executed in sequence. So,

we can get a three-node PostgreSQL cluster by creating the first instance, configuring the replication, cloning a second instance, and the third one.

In a declarative approach, the state of a system is defined using configuration, namely: there's a PostgreSQL 13 cluster with two replicas. This approach highly simplifies change management operations, and when these are stored in source control systems like Git, it enables the Infrastructure as Code capability. And Kubernetes takes it farther than deployment, as it makes sure that our request is fulfilled at any time.

What are the required skills to run PostgreSQL on Kubernetes?

Running PostgreSQL on Kubernetes requires both PostgreSQL and Kubernetes skills in your DevOps team. The best experience is when database administrators familiarize themselves with Kubernetes core concepts and are able to interact with Kubernetes administrators.

Our advice is for everyone that wants to fully exploit Cloud Native PostgreSQL to acquire the "Certified Kubernetes Administrator (CKA)" status from the CNCF certification program.

Why isn't CloudNativePG using StatefulSets?

CloudNativePG does not rely on `StatefulSet` resources, and instead manages the underlying PVCs directly by leveraging the selected storage class for dynamic provisioning. Please refer to the *Custom Pod Controller* section for details and reasons behind this decision.

54.2 High availability

What happens to the PostgreSQL clusters when the operator pod dies or it is not available for a certain amount of time?

The CloudNativePG operator, among other things, is responsible for self-healing capabilities. As such, they might not be available during an outage of the operator.

However, assuming that the outage does not affect the nodes where PostgreSQL clusters are running, the database will continue to serve normal operations, through the relevant Kubernetes services. Moreover, the *In-place updates of the instance manager* , which runs inside each PostgreSQL pod will still work, making sure that the database server is up, including accessory services like logging, export of metrics, continuous archiving of WAL files, etc.

To summarize:

an outage of the operator does not necessarily imply a PostgreSQL database outage; it's like running a database without a DBA or system administrator.

What are the reasons behind CloudNativePG not relying on a failover management tool like Patroni, repmgr, or Stolon?

Although part of the team that develops CloudNativePG has been heavily involved in repmgr in the past, we decided to take a different approach and directly extend the Kubernetes controller and rely on the Kubernetes API server to hold the status of a Postgres cluster, and use it as the only source of truth to:

- control High Availability of a Postgres cluster primarily via automated failover and switchover, coordinating itself with the *In-place updates of the instance manager*
- control the Kubernetes services, that is the entry points for your applications

Should I manually resync a former primary with the new one following a failover?

No. The operator does that automatically for you, and relies on `pg_rewind` to synchronize the former primary with the new one.

CHAPTER 55

API Reference

Package v1 contains API Schema definitions for the postgresql v1 API group

55.1 Resource Types

- Appendix B - Backup on object stores

55.2 Backup

A Backup resource is a request for a PostgreSQL backup by the user.

Field	Description
apiVersion **[Required]** string	*postgresql.cnpg.io/v1*
kind **[Required]** string	*Backup*
metadata **[Required]** _meta/v1.ObjectMeta_	No description provided.Refer to the Kubernetes API documentation for the fields of the *metadata* field.
spec **[Required]** *_BackupSpec_*	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
status *_BackupStatus_*	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

55.3 Cluster

Cluster is the Schema for the PostgreSQL API

Field	Description
apiVersion **[Required]** string	*postgresql.cnpg.io/v1*
kind **[Required]** string	*Cluster*
metadata **[Required]** _meta/v1.ObjectMeta_	No description provided.Refer to the Kubernetes API documentation for the fields of the *metadata* field.
spec **[Required]** *_ClusterSpec_*	Specification of the desired behavior of the cluster. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
status *_ClusterStatus_*	Most recently observed status of the cluster. This data may not be up to date. Populated by the system. Read-only. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

55.4 ClusterImageCatalog

ClusterImageCatalog is the Schema for the clusterimagecatalogs API

Field	Description
apiVersion **[Required]** string	*postgresql.cnpg.io/v1*
kind **[Required]** string	*ClusterImageCatalog*
metadata **[Required]** _meta/v1.ObjectMeta_	No description provided.Refer to the Kubernetes API documentation for the fields of the *metadata* field.
spec **[Required]** *_ImageCatalogSpec_*	Specification of the desired behavior of the ClusterImageCatalog. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

55.5 Database

Database is the Schema for the databases API

Field	Description
apiVersion **[Required]** string	*postgresql.cnpg.io/v1*
kind **[Required]** string	*Database*
metadata **[Required]** _meta/v1.ObjectMeta_	No description provided.Refer to the Kubernetes API documentation for the fields of the *metadata* field.
spec **[Required]** *_DatabaseSpec_*	Specification of the desired Database. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
status *_DatabaseStatus_*	Most recently observed status of the Database. 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

55.6 FailoverQuorum

Appears in:

FailoverQuorum contains the information about the current failover quorum status of a PG cluster. It is updated by the instance manager of the primary node and reset to zero by the operator to trigger an update.

Field	Description
apiVersion **[Required]** string	*postgresql.cnpg.io/v1*
kind **[Required]** string	*FailoverQuorum*
metadata **[Required]** _meta/v1.ObjectMeta_	No description provided.Refer to the Kubernetes API documentation for the fields of the *metadata* field.
status *_FailoverQuorumStatus_*	Most recently observed status of the failover quorum.

55.7 ImageCatalog

ImageCatalog is the Schema for the imagecatalogs API

Field	Description
apiVersion **[Required]** string	*postgresql.cnpg.io/v1*
kind **[Required]** string	*ImageCatalog*
metadata **[Required]** _meta/v1.ObjectMeta_	No description provided.Refer to the Kubernetes API documentation for the fields of the *metadata* field.
spec **[Required]** *_ImageCatalogSpec_*	Specification of the desired behavior of the ImageCatalog. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

55.8 Pooler

Pooler is the Schema for the poolers API

Field	Description
apiVersion **[Required]** string	*postgresql.cnpg.io/v1*
kind **[Required]** string	*Pooler*
metadata **[Required]** _meta/v1.ObjectMeta_	No description provided.Refer to the Kubernetes API documentation for the fields of the *metadata* field.
spec **[Required]** *_PoolerSpec_*	Specification of the desired behavior of the Pooler. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
status *_PoolerStatus_*	Most recently observed status of the Pooler. This data may not be up to date. Populated by the system. Read-only. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

55.9 Publication

Publication is the Schema for the publications API

Field	Description
apiVersion **[Required]** string	*postgresql.cnpg.io/v1*
kind **[Required]** string	*Publication*
metadata **[Required]** _meta/v1.ObjectMeta_	No description provided.Refer to the Kubernetes API documentation for the fields of the *metadata* field.
spec **[Required]** *_PublicationSpec_*	No description provided.
status **[Required]** *_PublicationStatus_*	No description provided.

55.10 ScheduledBackup

ScheduledBackup is the Schema for the scheduledbackups API

Field	Description
apiVersion **[Required]** string	*postgresql.cnpg.io/v1*
kind **[Required]** string	*ScheduledBackup*
metadata **[Required]** _meta/v1.ObjectMeta_	No description provided.Refer to the Kubernetes API documentation for the fields of the *metadata* field.
spec **[Required]** *_ScheduledBackupSpec_*	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
status *_ScheduledBackupStatus_*	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

55.11 Subscription

Subscription is the Schema for the subscriptions API

Field	Description
apiVersion **[Required]** string	*postgresql.cnpg.io/v1*
kind **[Required]** string	*Subscription*
metadata **[Required]** _meta/v1.ObjectMeta_	No description provided.Refer to the Kubernetes API documentation for the fields of the *metadata* field.
spec **[Required]** *_SubscriptionSpec_*	No description provided.
status **[Required]** *_SubscriptionStatus_*	No description provided.

55.12 AffinityConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

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

Field	Description
enablePodAntiAffinity _bool_	Activates anti-affinity for the pods. The operator will define pods anti-affinity unless this field is explicitly set to false
topologyKey _string_	TopologyKey to use for anti-affinity configuration. See k8s documentation for more info on that
nodeSelector _map[string]string_	NodeSelector is map of key-value pairs used to define the nodes on which the pods can run. More info: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/
nodeAffinity _core/v1.NodeAffinity_	NodeAffinity describes node affinity scheduling rules for the pod. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity
tolerations [_[]core/**v1.Toleration_**](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#toleration-v1-core)	Tolerations is a list of Tolerations that should be set for all the pods, in order to allow them to run on tainted nodes. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/
podAntiAffinityType _string_	PodAntiAffinityType allows the user to decide whether pod anti-affinity between cluster instance has to be considered a strong requirement during scheduling or not. Allowed values are: "preferred" (default if empty) or "required". Setting it to "required", could lead to instances remaining pending until new kubernetes nodes are added if all the existing nodes don't match the required pod anti-affinity rule. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#inter-pod-affinity-and-anti-affinity
additionalPodAntiAffinity _core/v1.PodAntiAffinity_	AdditionalPodAntiAffinity allows to specify pod anti-affinity terms to be added to the ones generated by the operator if EnablePodAntiAffinity is set to true (default) or to be used exclusively if set to false.
additionalPodAffinity _core/v1.PodAffinity_	AdditionalPodAffinity allows to specify pod affinity terms to be passed to all the cluster's pods.

55.13 AvailableArchitecture

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

AvailableArchitecture represents the state of a cluster's architecture

Field	Description
goArch **[Required]** _string_	GoArch is the name of the executable architecture
hash **[Required]** _string_	Hash is the hash of the executable

55.14 BackupConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

BackupConfiguration defines how the backup of the cluster are taken. The supported backup methods are BarmanObjectStore and VolumeSnapshot. For details and examples refer to the Backup and Recovery section of the documentation

Field	Description
volumeSnapshot _VolumeSnapshotConfiguration_	VolumeSnapshot provides the configuration for the execution of volume snapshot backups.
barmanObjectStore _github.com/cloudnative-pg/barman-cloud	The configuration for the barman-cloud tool suite
retentionPolicy _string_	RetentionPolicy is the retention policy to be used for backups and WALs (i.e. '60d'). The retention policy is expressed in the form of *XXu* where *XX* is a positive integer and *u* is in *[dwm]*- days, weeks, months. It's currently only applicable when using the BarmanObjectStore method.
target _BackupTarget_	The policy to decide which instance should perform backups. Available options are empty string, which will default to *prefer-standby* policy, *primary* to have backups run always on primary instances, *prefer-standby* to have backups run preferably on the most updated standby, if available.

55.15 BackupMethod

(Alias of `string`)

Appears in:

- BackupSpec {postgresql-cnpg-io-v1-BackupSpec}

BackupMethod defines the way of executing the physical base backups of the selected PostgreSQL instance

55.16 BackupPhase

(Alias of `string`)

Appears in:

- BackupStatus {postgresql-cnpg-io-v1-BackupStatus}

BackupPhase is the phase of the backup

55.17 BackupPluginConfiguration

Appears in:

- BackupSpec {postgresql-cnpg-io-v1-BackupSpec}

BackupPluginConfiguration contains the backup configuration used by the backup plugin

Field	Description
name **[Required]** _string_	Name is the name of the plugin managing this backup
parameters _map[string]string_	Parameters are the configuration parameters passed to the backup plugin for this backup

55.18 BackupSnapshotElementStatus

Appears in:

- BackupSnapshotStatus {postgresql-cnpg-io-v1-BackupSnapshotStatus}

BackupSnapshotElementStatus is a volume snapshot that is part of a volume snapshot method backup

Field	Description
name **[Required]** _string_	Name is the snapshot resource name
type **[Required]** _string_	Type is tho role of the snapshot in the cluster, such as PG_DATA, PG_WAL and PG_TABLESPACE
tablespaceName _string_	TablespaceName is the name of the snapshotted tablespace. Only set when type is PG_TABLESPACE

55.19 BackupSnapshotStatus

Appears in:

- BackupStatus {postgresql-cnpg-io-v1-BackupStatus}

BackupSnapshotStatus the fields exclusive to the volumeSnapshot method backup

Field	Description
elements [_[]BackupSnapshotElementStatus_](BackupSnapshotElementStatus)	The elements list, populated with the gathered volume snapshots

55.20 BackupSource

Appears in:

- BootstrapRecovery {postgresql-cnpg-io-v1-BootstrapRecovery}

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

Field	Description
LocalObjectReference _github.com/cloudnative-pg/machinery/pk	(Members of *LocalObjectReference* are embedded into this type.) No description provided.
endpointCA _github.com/cloudnative-pg/machinery/pk	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.

55.21 BackupSpec

Appears in:

- Appendix B - Backup on object stores

BackupSpec defines the desired state of Backup

Field	Description
cluster **[Required]** _github.com/cloudnative-pg/machinery/pk	The cluster to backup
target _BackupTarget_	The policy to decide which instance should perform this backup. If empty, it defaults to *cluster.spec.backup.target*. Available options are empty string, *primary* and *prefer-standby*. *primary* to have backups run always on primary instances, *prefer-standby* to have backups run preferably on the most updated standby, if available.
method _BackupMethod_	The backup method to be used, possible options are *barmanObjectStore*, *volumeSnapshot* or *plugin*. Defaults to: *barmanObjectStore*.
pluginConfiguration _BackupPluginConfiguration_	Configuration parameters passed to the plugin managing this backup
online _bool_	Whether the default type of backup with volume snapshots is online/hot (*true*, default) or offline/cold (*false*) Overrides the default setting specified in the cluster field '.spec.backup.volumeSnapshot.online'
onlineConfiguration _OnlineConfiguration_	Configuration parameters to control the online/hot backup with volume snapshots Overrides the default settings specified in the cluster '.backup.volumeSnapshot.onlineConfiguration' stanza

55.22 BackupStatus

Appears in:

- Appendix B - Backup on object stores

BackupStatus defines the observed state of Backup

Field	Description
BarmanCredentials _github.com/cloudnative-pg/barman-cloud	(Members of *BarmanCredentials* are embedded into this type.) The potential credentials for each cloud provider
endpointCA _github.com/cloudnative-pg/machinery/pk	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.
endpointURL _string_	Endpoint to be used to upload data to the cloud, overriding the automatic endpoint discovery
destinationPath _string_	The path where to store the backup (i.e. s3://bucket/path/to/folder) this path, with different destination folders, will be used for WALs and for data. This may not be populated in case of errors.
serverName _string_	The server name on S3, the cluster name is used if this parameter is omitted
encryption _string_	Encryption method required to S3 API
backupId _string_	The ID of the Barman backup
backupName _string_	The Name of the Barman backup
phase _BackupPhase_	The last backup status
startedAt _meta/v1.Time_	When the backup was started
stoppedAt _meta/v1.Time_	When the backup was terminated
beginWal _string_	The starting WAL
endWal _string_	The ending WAL

continues on next page

Table 19 – continued from previous page

Field	Description
beginLSN _string_	The starting xlog
endLSN _string_	The ending xlog
error _string_	The detected error
commandOutput _string_	Unused. Retained for compatibility with old versions.
commandError _string_	The backup command output in case of error
backupLabelFile _[]byte_	Backup label file content as returned by Postgres in case of online (hot) backups
tablespaceMapFile _[]byte_	Tablespace map file content as returned by Postgres in case of online (hot) backups
instanceID *_InstanceID_*	Information to identify the instance where the backup has been taken from
snapshotBackupStatus *_BackupSnapshotStatus_*	Status of the volumeSnapshot backup
method *_BackupMethod_*	The backup method being used
online _bool_	Whether the backup was online/hot (*true*) or offline/cold (*false*)
pluginMetadata _map[string]string_	A map containing the plugin metadata

55.23 BackupTarget

(Alias of `string`)

Appears in:

- BackupConfiguration {postgresql-cnpg-io-v1-BackupConfiguration}

BackupTarget describes the preferred targets for a backup

55.24 BootstrapConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

BootstrapConfiguration contains information about how to create the PostgreSQL cluster. Only a single bootstrap method can be defined among the supported ones. `initdb` will be used as the bootstrap method if left unspecified. Refer to the Bootstrap page of the documentation for more information.

Field	Description
initdb *_BootstrapInitDB_*	Bootstrap the cluster via initdb
recovery *_BootstrapRecovery_*	Bootstrap the cluster from a backup
pg_basebackup *_BootstrapPgBaseBackup_*	Bootstrap the cluster taking a physical backup of another compatible PostgreSQL instance

55.25 BootstrapInitDB

Appears in:

- BootstrapConfiguration {postgresql-cnpg-io-v1-BootstrapConfiguration}

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

Field	Description
database _string_	Name of the database used by the application. Default: *app*.
owner _string_	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the *database* key.
secret _github.com/cloudnative-pg/machin	Name of the secret containing the initial credentials for the owner of the user database. If empty a new secret will be created from scratch
options _[]string_	The list of options that must be passed to initdb when creating the cluster. Deprecated: This could lead to inconsistent configurations, please use the explicit provided parameters instead. If defined, explicit values will be ignored.
dataChecksums _bool_	Whether the *-k* option should be passed to initdb, enabling checksums on data pages (default: *false*)
encoding _string_	The value to be passed as option *--encoding* for initdb (default:*UTF8*)
localeCollate _string_	The value to be passed as option *--lc-collate* for initdb (default:*C*)
localeCType _string_	The value to be passed as option *--lc-ctype* for initdb (default:*C*)
locale _string_	Sets the default collation order and character classification in the new database.
localeProvider _string_	This option sets the locale provider for databases created in the new cluster. Available from PostgreSQL 16.
icuLocale _string_	Specifies the ICU locale when the ICU provider is used. This option requires *localeProvider* to be set to *icu*. Available from PostgreSQL 15.
icuRules _string_	Specifies additional collation rules to customize the behavior of the default collation. This option requires *localeProvider* to be set to *icu*. Available from PostgreSQL 16.
builtinLocale _string_	Specifies the locale name when the builtin provider is used. This option requires *localeProvider* to be set to *builtin*. Available from PostgreSQL 17.
walSegmentSize _int_	The value in megabytes (1 to 1024) to be passed to the *--wal-segsize* option for initdb (default: empty, resulting in PostgreSQL default: 16MB)
postInitSQL _[]string_	List of SQL queries to be executed as a superuser in the *postgres* database right after the cluster has been created- to be used with extreme care (by default empty)
postInitApplicationSQL _[]string_	List of SQL queries to be executed as a superuser in the application database right after the cluster has been created- to be used with extreme care (by default empty)
postInitTemplateSQL _[]string_	List of SQL queries to be executed as a superuser in the *template1* database right after the cluster has been created- to be used with extreme care (by default empty)
import *_Import_*	Bootstraps the new cluster by importing data from an existing PostgreSQL instance using logical backup (*pg_dump* and *pg_restore*)
postInitApplicationSQLRefs *_SQLRefs_*	List of references to ConfigMaps or Secrets containing SQL files to be executed as a superuser in the application database right after the cluster has been created. The references are processed in a specific order: first, all Secrets are processed, followed by all ConfigMaps. Within each group, the processing order follows the sequence specified in their respective arrays. (by default empty)
postInitTemplateSQLRefs *_SQLRefs_*	List of references to ConfigMaps or Secrets containing SQL files to be executed as a superuser in the *template1* database right after the cluster has been created. The references are processed in a specific order: first, all Secrets are processed, followed by all ConfigMaps. Within each group, the processing order follows the sequence specified in their respective arrays. (by default empty)

continues on next page

Table 21 – continued from previous page

Field	Description
postInitSQLRefs _SQLRefs_	List of references to ConfigMaps or Secrets containing SQL files to be executed as a superuser in the *postgres* database right after the cluster has been created. The references are processed in a specific order: first, all Secrets are processed, followed by all ConfigMaps. Within each group, the processing order follows the sequence specified in their respective arrays. (by default empty)

55.26 BootstrapPgBaseBackup

Appears in:

- BootstrapConfiguration {postgresql-cnpg-io-v1-BootstrapConfiguration}

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

Field	Description
source **[Required]** _string_	The name of the server of which we need to take a physical backup
database _string_	Name of the database used by the application. Default: *app*.
owner _string_	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the *database* key.
secret _github.com/cloudnative-pg/machin	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

55.27 BootstrapRecovery

Appears in:

- BootstrapConfiguration {postgresql-cnpg-io-v1-BootstrapConfiguration}

BootstrapRecovery contains the configuration required to restore from an existing cluster using 3 methodologies: external cluster, volume snapshots or backup objects. Full recovery and Point-In-Time Recovery are supported. The method can be also be used to create clusters in continuous recovery (replica clusters), also supporting cascading replication when `instances` >

1. Once the cluster exits recovery, the password for the superuser will be changed through the provided secret. Refer to the Bootstrap page of the documentation for more information.

Field	Description
backup _BackupSource_	The backup object containing the physical base backup from which to initiate the recovery procedure. Mutually exclusive with *source* and *volumeSnapshots*.
source _string_	The external cluster whose backup we will restore. This is also used as the name of the folder under which the backup is stored, so it must be set to the name of the source cluster Mutually exclusive with *backup*.

continues on next page

Table 23 – continued from previous page

Field	Description
volumeSnapshots _*DataSource*_	The static PVC data source(s) from which to initiate the recovery procedure. Currently supporting *VolumeSnapshot* and *PersistentVolumeClaim* resources that map an existing PVC group, compatible with CloudNativePG, and taken with a cold backup copy on a fenced Postgres instance (limitation which will be removed in the future when online backup will be implemented). Mutually exclusive with *backup*.
recoveryTarget _RecoveryTarget_ 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	
database _string_	Name of the database used by the application. Default: *app*.
owner _string_	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the *database* key.
secret _github.com/cloudnative-pg/machin	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

55.28 CatalogImage

Appears in:

- ImageCatalogSpec {postgresql-cnpg-io-v1-ImageCatalogSpec}

CatalogImage defines the image and major version

Field	Description
image **[Required]** _string_	The image reference
major **[Required]** _int_	The PostgreSQL major version of the image. Must be unique within the catalog.

55.29 CertificatesConfiguration

Appears in:

- CertificatesStatus {postgresql-cnpg-io-v1-CertificatesStatus}

CertificatesConfiguration contains the needed configurations to handle server certificates.

Field	Description
serverCASecret _string_	The secret containing the Server CA certificate. If not defined, a new secret will be created with a self-signed CA and will be used to generate the TLS certificate ServerTLSSecret. Contains: *`ca.crt`: CA that should be used to validate the server certificate, used as `sslrootcert` in client connection strings. ca.key*: key used to generate Server SSL certs, if ServerTLSSecret is provided, this can be omitted.

continues on next page

Table 25 – continued from previous page

Field	Description
serverTLSSecret _string_	The secret of type kubernetes.io/tls containing the server TLS certificate and key that will be set as *ssl_cert_file* and *ssl_key_file* so that clients can connect to postgres securely. If not defined, ServerCASecret must provide also *ca.key* and a new secret will be created using the provided CA.
replicationTLSSecret _string_	The secret of type kubernetes.io/tls containing the client certificate to authenticate as the *streaming_replica* user. If not defined, ClientCASecret must provide also *ca.key*, and a new secret will be created using the provided CA.
clientCASecret _string_	The secret containing the Client CA certificate. If not defined, a new secret will be created with a self-signed CA and will be used to generate all the client certificates. Contains: *`ca.crt`: CA that should be used to validate the client certificates, used as `ssl_ca_file` of all the instances. ca.key*: key used to generate client certificates, if ReplicationTLSSecret is provided, this can be omitted.
serverAltDNSNames _[]string_	The list of the server alternative DNS names to be added to the generated server TLS certificates, when required.

55.30 CertificatesStatus

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

CertificatesStatus contains configuration certificates and related expiration dates.

Field	Description
CertificatesConfiguration _*CertificatesConfiguration*_	(Members of *CertificatesConfiguration* are embedded into this type.) Needed configurations to handle server certificates, initialized with default values, if needed.
expirations _map[string]string_	Expiration dates for all certificates.

55.31 ClusterMonitoringTLSConfiguration

Appears in:

- MonitoringConfiguration {postgresql-cnpg-io-v1-MonitoringConfiguration}

ClusterMonitoringTLSConfiguration is the type containing the TLS configuration for the cluster's monitoring

Field	Description
enabled _bool_	Enable TLS for the monitoring endpoint. Changing this option will force a rollout of all instances.

55.32 ClusterSpec

Appears in:

- *Multi-availability zone Kubernetes clusters*

ClusterSpec defines the desired state of Cluster

Field	Description
description _string_	Description of this PostgreSQL cluster
inheritedMetadata *_EmbeddedObjectMetadata_*	Metadata that will be inherited by all objects related to the Cluster
imageName _string_	Name of the container image, supporting both tags (*<image>:<tag>*) and digests for deterministic and repeatable deployments (*<image>:<tag>@sha256:<digestValue>*)
imageCatalogRef *_ImageCatalogRef_*	Defines the major PostgreSQL version we want to use within an ImageCatalog
imagePullPolicy _core/v1.PullPolicy_	Image pull policy. One of *Always*, *Never* or *IfNotPresent*. If not defined, it defaults to *IfNotPresent*. Cannot be updated. More info: https://kubernetes.io/docs/concepts/containerimages#updating-images
schedulerName _string_	If specified, the pod will be dispatched by specified Kubernetes scheduler. If not specified, the pod will be dispatched by the default scheduler. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/
postgresUID _int64_	The UID of the *postgres* user inside the image, defaults to *26*
postgresGID _int64_	The GID of the *postgres* user inside the image, defaults to *26*
instances **[Required]** _int_	Number of instances required in the cluster
minSyncReplicas _int_	Minimum number of instances required in synchronous replication with the primary. Undefined or 0 allow writes to complete when no standby is available.
maxSyncReplicas _int_	The target value for the synchronous replication quorum, that can be decreased if the number of ready standbys is lower than this. Undefined or 0 disable synchronous replication.
postgresql *_PostgresConfiguration_*	Configuration of the PostgreSQL server
replicationSlots *_ReplicationSlotsConfiguration_*	Replication slots management configuration
bootstrap *_BootstrapConfiguration_*	Instructions to bootstrap this cluster
replica *_ReplicaClusterConfiguration_*	Replica cluster configuration
superuserSecret _github.com/cloudnative-pg/machinery/pk	The secret containing the superuser password. If not defined a new secret will be created with a randomly generated password
enableSuperuserAccess _bool_	When this option is enabled, the operator will use the *SuperuserSecret* to update the *postgres* user password (if the secret is not present, the operator will automatically create one). When this option is disabled, the operator will ignore the *SuperuserSecret* content, delete it when automatically created, and then blank the password of the *postgres* user by setting it to *NULL*. Disabled by default.
certificates *_CertificatesConfiguration_*	The configuration for the CA and related certificates

continues on next page

Table 28 – continued from previous page

Field	Description
imagePullSecrets [_[]github.com/cloudnative-pg/machinery//pkg.go.dev/github.com/cloudnative-pg/machinery/pkg/api/#LocalObjectReference)	The list of pull secrets to be used to pull the images
storage _*StorageConfiguration*_	Configuration of the storage of the instances
serviceAccountTemplate _*ServiceAccountTemplate*_	Configure the generation of the service account
walStorage _StorageConfiguration_	
ephemeralVolumeSource _core/v1.EphemeralVolumeSource_	EphemeralVolumeSource allows the user to configure the source of ephemeral volumes.
startDelay _int32_	The time in seconds that is allowed for a PostgreSQL instance to successfully start up (default 3600). The startup probe failure threshold is derived from this value using the formula: ceiling(startDelay / 10).
stopDelay _int32_	The time in seconds that is allowed for a PostgreSQL instance to gracefully shutdown (default 1800)
smartShutdownTimeout _int32_	The time in seconds that controls the window of time reserved for the smart shutdown of Postgres to complete. Make sure you reserve enough time for the operator to request a fast shutdown of Postgres (that is: *stopDelay- smartShutdownTimeout*).
switchoverDelay _int32_	The time in seconds that is allowed for a primary PostgreSQL instance to gracefully shutdown during a switchover. Default value is 3600 seconds (1 hour).
failoverDelay _int32_	The amount of time (in seconds) to wait before triggering a failover after the primary PostgreSQL instance in the cluster was detected to be unhealthy
livenessProbeTimeout _int32_	LivenessProbeTimeout is the time (in seconds) that is allowed for a PostgreSQL instance to successfully respond to the liveness probe (default 30). The Liveness probe failure threshold is derived from this value using the formula: ceiling(livenessProbe / 10).
affinity _*AffinityConfiguration*_	Affinity/Anti-affinity rules for Pods
topologySpreadConstraints [_[]core/**v1.TopologySpreadConstraint_**](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#topologyspreadconstraint-v1-core)	TopologySpreadConstraints specifies how to spread matching pods among the given topology. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/topology-spread-constraints/
resources _core/v1.ResourceRequirements_	Resources requirements of every generated Pod. Please refer to https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/ for more information.
ephemeralVolumesSizeLimit _*EphemeralVolumesSizeLimitConfiguration*_	EphemeralVolumesSizeLimit allows the user to set the limits for the ephemeral volumes
priorityClassName _string_	Name of the priority class which will be used in every generated Pod, if the PriorityClass specified does not exist, the pod will not be able to schedule. Please refer to https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/#priorityclass for more information
primaryUpdateStrategy _PrimaryUpdateStrategy_ or manual (*supervised*)	
primaryUpdateMethod _PrimaryUpdateMethod_	

continues on next page

Table 28 – continued from previous page

Field	Description
backup _BackupConfiguration_	The configuration to be used for backups
nodeMaintenanceWindow _NodeMaintenanceWindow_	Define a maintenance window for the Kubernetes nodes
monitoring _MonitoringConfiguration_	The configuration of the monitoring infrastructure of this cluster
externalClusters [_]ExternalCluster_](ExternalCluster)	The list of external clusters which are used in the configuration
logLevel _string_	The instances' log level, one of the following values: error, warning, info (default), debug, trace
projectedVolumeTemplate _core/v1.ProjectedVolumeSource_	Template to be used to define projected volumes, projected volumes will be mounted under */projected* base folder
env [_]core/**v1.EnvVar_**](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#envvar-v1-core)	Env follows the Env format to pass environment variables to the pods created in the cluster
envFrom [_]core/**v1.EnvFromSource_**](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#envfromsource-v1-core)	EnvFrom follows the EnvFrom format to pass environment variables sources to the pods to be used by Env
managed _ManagedConfiguration_	The configuration that is used by the portions of PostgreSQL that are managed by the instance manager
seccompProfile _core/v1.SeccompProfile_	The SeccompProfile applied to every Pod and Container. Defaults to: *RuntimeDefault*
tablespaces [_]TablespaceConfiguration_](TablespaceConfiguration)	The tablespaces configuration
enablePDB _bool_	Manage the *PodDisruptionBudget* resources within the cluster. When configured as *true* (default setting), the pod disruption budgets will safeguard the primary node from being terminated. Conversely, setting it to *false* will result in the absence of any *PodDisruptionBudget* resource, permitting the shutdown of all nodes hosting the PostgreSQL cluster. This latter configuration is advisable for any PostgreSQL cluster employed for development/staging purposes.
plugins [_]PluginConfiguration_](PluginConfiguration)	The plugins configuration, containing any plugin to be loaded with the corresponding configuration
probes _ProbesConfiguration_	The configuration of the probes to be injected in the PostgreSQL Pods.

55.33 ClusterStatus

Appears in:

- *Multi-availability zone Kubernetes clusters*

ClusterStatus defines the observed state of Cluster

Field	Description
instances _int_	The total number of PVC Groups detected in the cluster. It may differ from the number of existing instance pods.
readyInstances _int_	The total number of ready instances in the cluster. It is equal to the number of ready instance pods.
instancesStatus _map[PodStatus][]string_	InstancesStatus indicates in which status the instances are

continues on next page

Table 29 – continued from previous page

Field	Description
instancesReportedState [_map[PodName]InstanceReportedState_](InstanceReportedState)	The reported state of the instances during the last reconciliation loop
managedRolesStatus _ManagedRoles_	ManagedRolesStatus reports the state of the managed roles in the cluster
tablespacesStatus [_[]TablespaceState_](TablespaceState)	TablespacesStatus reports the state of the declarative tablespaces in the cluster
timelineID _int_	The timeline of the Postgres cluster
topology _Topology_	Instances topology.
latestGeneratedNode _int_	ID of the latest generated node (used to avoid node name clashing)
currentPrimary _string_	Current primary instance
targetPrimary _string_	Target primary instance, this is different from the previous one during a switchover or a failover
lastPromotionToken _string_	LastPromotionToken is the last verified promotion token that was used to promote a replica cluster
pvcCount _int32_	How many PVCs have been created by this cluster
jobCount _int32_	How many Jobs have been created by this cluster
danglingPVC _[]string_	List of all the PVCs created by this cluster and still available which are not attached to a Pod
resizingPVC _[]string_	List of all the PVCs that have ResizingPVC condition.
initializingPVC _[]string_	List of all the PVCs that are being initialized by this cluster
healthyPVC _[]string_	List of all the PVCs not dangling nor initializing
unusablePVC _[]string_	List of all the PVCs that are unusable because another PVC is missing
writeService _string_	Current write pod
readService _string_	Current list of read pods
phase _string_	Current phase of the cluster
phaseReason _string_	Reason for the current phase
secretsResourceVersion _SecretsResourceVersion_	The list of resource versions of the secrets managed by the operator. Every change here is done in the interest of the instance manager, which will refresh the secret data
configMapResourceVersion _ConfigMapResourceVersion_	The list of resource versions of the configmaps, managed by the operator. Every change here is done in the interest of the instance manager, which will refresh the configmap data
certificates _CertificatesStatus_	The configuration for the CA and related certificates, initialized with defaults.
firstRecoverabilityPoint _string_	The first recoverability point, stored as a date in RFC3339 format. This field is calculated from the content of FirstRecoverabilityPointByMethod. Deprecated: the field is not set for backup plugins.
firstRecoverabilityPointByMethod [_map[BackupMethod]meta/**v1.Time**_](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#time-v1-meta)	The first recoverability point, stored as a date in RFC3339 format, per backup method type. Deprecated: the field is not set for backup plugins.
lastSuccessfulBackup _string_	Last successful backup, stored as a date in RFC3339 format. This field is calculated from the content of LastSuccessfulBackupByMethod. Deprecated: the field is not set for backup plugins.

continues on next page

Table 29 – continued from previous page

Field	Description
lastSuccessfulBackupByMethod [_map[BackupMethod]meta/**v1.Time_**](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#time-v1-meta)	Last successful backup, stored as a date in RFC3339 format, per backup method type. Deprecated: the field is not set for backup plugins.
lastFailedBackup _string_	Last failed backup, stored as a date in RFC3339 format. Deprecated: the field is not set for backup plugins.
cloudNativePGCommitHash _string_	The commit hash number of which this operator running
currentPrimaryTimestamp _string_	The timestamp when the last actual promotion to primary has occurred
currentPrimaryFailingSinceTimestamp _string_	The timestamp when the primary was detected to be unhealthy This field is reported when *.spec.failoverDelay* is populated or during online upgrades
targetPrimaryTimestamp _string_	The timestamp when the last request for a new primary has occurred
poolerIntegrations *_PoolerIntegrations_*	The integration needed by poolers referencing the cluster
cloudNativePGOperatorHash _string_	The hash of the binary of the operator
availableArchitectures [_[]AvailableArchitecture_](AvailableArchitecture)	AvailableArchitectures reports the available architectures of a cluster
conditions [_[]meta/**v1.Condition_**](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#condition-v1-meta)	Conditions for cluster object
instanceNames _[]string_	List of instance names in the cluster
onlineUpdateEnabled _bool_	OnlineUpdateEnabled shows if the online upgrade is enabled inside the cluster
image _string_	Image contains the image name used by the pods
pgDataImageInfo *_ImageInfo_*	PGDataImageInfo contains the details of the latest image that has run on the current data directory.
pluginStatus [_[]PluginStatus_](PluginStatus)	PluginStatus is the status of the loaded plugins
switchReplicaClusterStatus *_SwitchReplicaClusterStatus_*	SwitchReplicaClusterStatus is the status of the switch to replica cluster
demotionToken _string_	DemotionToken is a JSON token containing the information from pg_controldata such as Database system identifier, Latest checkpoint's TimeLineID, Latest checkpoint's REDO location, Latest checkpoint's REDO WAL file, and Time of latest checkpoint
systemID _string_	SystemID is the latest detected PostgreSQL SystemID

55.34 ConfigMapResourceVersion

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

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

Field	Description
metrics _map[string]string_	A map with the versions of all the config maps used to pass metrics. Map keys are the config map names, map values are the versions

55.35 DataDurabilityLevel

(Alias of `string`)

Appears in:

- SynchronousReplicaConfiguration {postgresql-cnpg-io-v1-SynchronousReplicaConfiguration}

DataDurabilityLevel specifies how strictly to enforce synchronous replication when cluster instances are unavailable. Options are `required` or `preferred` .

55.36 DataSource

Appears in:

- BootstrapRecovery {postgresql-cnpg-io-v1-BootstrapRecovery}

DataSource contains the configuration required to bootstrap a PostgreSQL cluster from an existing storage

Field	Description
storage **[Required]** _core/v1.TypedLocalObjectReference_	Configuration of the storage of the instances
walStorage _core/v1.TypedLocalObjectReference_	Configuration of the storage for PostgreSQL WAL (Write-Ahead Log)
tablespaceStorage [_map[string]core/**v1.TypedLocalObjectReference_**](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.28/#typedlocalobjectreference-v1-core)	Configuration of the storage for PostgreSQL tablespaces

55.37 DatabaseObjectSpec

Appears in:

- ExtensionSpec {postgresql-cnpg-io-v1-ExtensionSpec}

DatabaseObjectSpec contains the fields which are common to every database object

Field	Description
name **[Required]** _string_	Name of the extension/schema
ensure *_EnsureOption_*	Specifies whether an extension/schema should be present or absent in the database. If set to *present*, the extension/schema will be created if it does not exist. If set to *absent*, the extension/schema will be removed if it exists.

55.38 DatabaseObjectStatus

Appears in:

- DatabaseStatus {postgresql-cnpg-io-v1-DatabaseStatus}

DatabaseObjectStatus is the status of the managed database objects

Field	Description
name **[Required]** _string_	The name of the object
applied **[Required]** _bool_	True of the object has been installed successfully in the database
message _string_	Message is the object reconciliation message

55.39 DatabaseReclaimPolicy

(Alias of `string`)

Appears in:

- DatabaseSpec {postgresql-cnpg-io-v1-DatabaseSpec}

DatabaseReclaimPolicy describes a policy for end-of-life maintenance of databases.

55.40 DatabaseRoleRef

Appears in:

- TablespaceConfiguration {postgresql-cnpg-io-v1-TablespaceConfiguration}

DatabaseRoleRef is a reference an a role available inside PostgreSQL

Field	Description
name _string_	No description provided.

55.41 DatabaseSpec

Appears in:

- *Configure the application database*

DatabaseSpec is the specification of a Postgresql Database, built around the `CREATE DATABASE`, `ALTER DATABASE`, and `DROP DATABASE` SQL commands of PostgreSQL.

Field	Description
cluster **[Required]** _core/v1.LocalObjectReference_	The name of the PostgreSQL cluster hosting the database.
ensure *_EnsureOption_*	Ensure the PostgreSQL database is *present* or *absent*- defaults to "present".
name **[Required]** _string_	The name of the database to create inside PostgreSQL. This setting cannot be changed.

continues on next page

Table 35 – continued from previous page

Field	Description
owner **[Required]** _string_	Maps to the *OWNER* parameter of *CREATE DATABASE*. Maps to the *OWNER TO* command of *ALTER DATABASE*. The role name of the user who owns the database inside PostgreSQL.
template _string_	Maps to the *TEMPLATE* parameter of *CREATE DATABASE*. This setting cannot be changed. The name of the template from which to create this database.
encoding _string_	Maps to the *ENCODING* parameter of *CREATE DATABASE*. This setting cannot be changed. Character set encoding to use in the database.
locale _string_	Maps to the *LOCALE* parameter of *CREATE DATABASE*. This setting cannot be changed. Sets the default collation order and character classification in the new database.
localeProvider _string_	Maps to the *LOCALE_PROVIDER* parameter of *CREATE DATABASE*. This setting cannot be changed. This option sets the locale provider for databases created in the new cluster. Available from PostgreSQL 16.
localeCollate _string_	Maps to the *LC_COLLATE* parameter of *CREATE DATABASE*. This setting cannot be changed.
localeCType _string_	Maps to the *LC_CTYPE* parameter of *CREATE DATABASE*. This setting cannot be changed.
icuLocale _string_	Maps to the *ICU_LOCALE* parameter of *CREATE DATABASE*. This setting cannot be changed. Specifies the ICU locale when the ICU provider is used. This option requires *localeProvider* to be set to *icu*. Available from PostgreSQL 15.
icuRules _string_	Maps to the *ICU_RULES* parameter of *CREATE DATABASE*. This setting cannot be changed. Specifies additional collation rules to customize the behavior of the default collation. This option requires *localeProvider* to be set to *icu*. Available from PostgreSQL 16.
builtinLocale _string_	Maps to the *BUILTIN_LOCALE* parameter of *CREATE DATABASE*. This setting cannot be changed. Specifies the locale name when the builtin provider is used. This option requires *localeProvider* to be set to *builtin*. Available from PostgreSQL 17.
collationVersion _string_	Maps to the *COLLATION_VERSION* parameter of *CREATE DATABASE*. This setting cannot be changed.
isTemplate _bool_	Maps to the *IS_TEMPLATE* parameter of *CREATE DATABASE* and *ALTER DATABASE*. If true, this database is considered a template and can be cloned by any user with *CREATEDB* privileges.
allowConnections _bool_	Maps to the *ALLOW_CONNECTIONS* parameter of *CREATE DATABASE* and *ALTER DATABASE*. If false then no one can connect to this database.
connectionLimit _int_	Maps to the *CONNECTION LIMIT* clause of *CREATE DATABASE* and *ALTER DATABASE*. How many concurrent connections can be made to this database. -1 (the default) means no limit.
tablespace _string_	Maps to the *TABLESPACE* parameter of *CREATE DATABASE*. Maps to the *SET TABLESPACE* command of *ALTER DATABASE*. The name of the tablespace (in PostgreSQL) that will be associated with the new database. This tablespace will be the default tablespace used for objects created in this database.
databaseReclaimPolicy _DatabaseReclaimPolicy_	The policy for end-of-life maintenance of this database.
schemas [_[]SchemaSpec_](SchemaSpec)	The list of schemas to be managed in the database

continues on next page

Table 35 – continued from previous page

Field	Description
extensions [_[]ExtensionSpec_](ExtensionSpec)	The list of extensions to be managed in the database
fdws [_[]FDWSpec_](FDWSpec)	The list of foreign data wrappers to be managed in the database

55.42 DatabaseStatus

Appears in:

- *Configure the application database*

DatabaseStatus defines the observed state of Database

Field	Description
observedGeneration _int64_	A sequence number representing the latest desired state that was synchronized
applied _bool_	Applied is true if the database was reconciled correctly
message _string_	Message is the reconciliation output message
schemas [_[]DatabaseObjectStatus_](DatabaseObjectStatus)	Schemas is the status of the managed schemas
extensions [_[]DatabaseObjectStatus_](DatabaseObjectStatus)	Extensions is the status of the managed extensions
fdws [_[]DatabaseObjectStatus_](DatabaseObjectStatus)	FDWs is the status of the managed FDWs

55.43 EmbeddedObjectMetadata

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

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

Field	Description
labels _map[string]string_	No description provided.
annotations _map[string]string_	No description provided.

55.44 EnsureOption

(Alias of `string`)

Appears in:

- DatabaseObjectSpec {postgresql-cnpg-io-v1-DatabaseObjectSpec}

EnsureOption represents whether we should enforce the presence or absence of a Role in a PostgreSQL instance

55.45 EphemeralVolumesSizeLimitConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

EphemeralVolumesSizeLimitConfiguration contains the configuration of the ephemeral storage

Field	Description
shm _k8s.io/apimachinery/pkg/api/resource.Quantity_	Shm is the size limit of the shared memory volume
temporaryData _k8s.io/apimachinery/pkg/api/resource.Quantity_	TemporaryData is the size limit of the temporary data volume

55.46 ExtensionConfiguration

Appears in:

- PostgresConfiguration {postgresql-cnpg-io-v1-PostgresConfiguration}

ExtensionConfiguration is the configuration used to add PostgreSQL extensions to the Cluster.

Field	Description
name **[Required]** _string_	The name of the extension, required
image **[Required]** _core/v1.ImageVolumeSource_	The image containing the extension, required
extension_control_path _[]string_	The list of directories inside the image which should be added to extension_control_path. If not defined, defaults to "/share".
dynamic_library_path _[]string_	The list of directories inside the image which should be added to dynamic_library_path. If not defined, defaults to "/lib".
ld_library_path _[]string_	The list of directories inside the image which should be added to ld_library_path.

55.47 ExtensionSpec

Appears in:

- DatabaseSpec {postgresql-cnpg-io-v1-DatabaseSpec}

ExtensionSpec configures an extension in a database

Field	Description
DatabaseObjectSpec *DatabaseObjectSpec*	(Members of *DatabaseObjectSpec* are embedded into this type.) Common fields
version **[Required]** _string_	The version of the extension to install. If empty, the operator will install the default version (whatever is specified in the extension's control file)
schema **[Required]** _string_	The name of the schema in which to install the extension's objects, in case the extension allows its contents to be relocated. If not specified (default), and the extension's control file does not specify a schema either, the current default object creation schema is used.

55.48 ExternalCluster

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

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

Field	Description
name **[Required]** _string_	The server name, required
connectionParameters _map[string]string_	The list of connection parameters, such as dbname, host, username, etc
sslCert _core/v1.SecretKeySelector_	The reference to an SSL certificate to be used to connect to this instance
sslKey _core/v1.SecretKeySelector_	The reference to an SSL private key to be used to connect to this instance
sslRootCert _core/v1.SecretKeySelector_	The reference to an SSL CA public key to be used to connect to this instance
password _core/v1.SecretKeySelector_	The reference to the password to be used to connect to the server. If a password is provided, CloudNativePG creates a PostgreSQL passfile at */controller/external/NAME/pass* (where "NAME" is the cluster's name). This passfile is automatically referenced in the connection string when establishing a connection to the remote PostgreSQL server from the current PostgreSQL *Cluster*. This ensures secure and efficient password management for external clusters.
barmanObjectStore _github.com/cloudnative-pg/barman-cloud	The configuration for the barman-cloud tool suite
plugin **[Required]** *_PluginConfiguration_*	The configuration of the plugin that is taking care of WAL archiving and backups for this external cluster

55.49 FDWSpec

Appears in:

- DatabaseSpec {postgresql-cnpg-io-v1-DatabaseSpec}

FDWSpec configures an Foreign Data Wrapper in a database

Field	Description
DatabaseObjectSpec *_DatabaseObjectSpec_*	(Members of *DatabaseObjectSpec* are embedded into this type.) Common fields
handler _string_	Name of the handler function (e.g., "postgres_fdw_handler"). This will be empty if no handler is specified. In that case, the default handler is registered when the FDW extension is created.
validator _string_	Name of the validator function (e.g., "postgres_fdw_validator"). This will be empty if no validator is specified. In that case, the default validator is registered when the FDW extension is created.
owner _string_	Owner specifies the database role that will own the Foreign Data Wrapper. The role must have superuser privileges in the target database.
options [_[]OptionSpec_](OptionSpec)	Options specifies the configuration options for the FDW (key is the option name, value is the option value).

continues on next page

Table 42 – continued from previous page

Field	Description
usage [_[]UsageSpec_](UsageSpec)	List of roles for which *USAGE* privileges on the FDW are granted or revoked.

55.50 FailoverQuorumStatus

Appears in:

- FailoverQuorum {postgresql-cnpg-io-v1-FailoverQuorum}

FailoverQuorumStatus is the latest observed status of the failover quorum of the PG cluster.

Field	Description
method _string_	Contains the latest reported Method value.
standbyNames _[]string_	StandbyNames is the list of potentially synchronous instance names.
standbyNumber _int_	StandbyNumber is the number of synchronous standbys that transactions need to wait for replies from.
primary _string_	Primary is the name of the primary instance that updated this object the latest time.

55.51 ImageCatalogRef

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ImageCatalogRef defines the reference to a major version in an ImageCatalog

Field	Description
TypedLocalObjectReference _core/v1.TypedLocalObjectReference_	(Members of *TypedLocalObjectReference* are embedded into this type.) No description provided.
major **[Required]** _int_	The major version of PostgreSQL we want to use from the ImageCatalog

55.52 ImageCatalogSpec

Appears in:

- ClusterImageCatalog {postgresql-cnpg-io-v1-ClusterImageCatalog}

ImageCatalogSpec defines the desired ImageCatalog

Field	Description
images **[Required]** [_[]CatalogImage_](CatalogImage)	List of CatalogImages available in the catalog

55.53 ImageInfo

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

ImageInfo contains the information about a PostgreSQL image

Field	Description
image **[Required]** _string_	Image is the image name
majorVersion **[Required]** _int_	MajorVersion is the major version of the image

55.54 Import

Appears in:

- BootstrapInitDB {postgresql-cnpg-io-v1-BootstrapInitDB}

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

Field	Description
source **[Required]** *_ImportSource_*	The source of the import
type **[Required]** *_SnapshotType_*	The import type. Can be *microservice* or *monolith*.
databases **[Required]** _[]string_	The databases to import
roles _[]string_	The roles to import
postImportApplicationSQL _[]string_	List of SQL queries to be executed as a superuser in the application database right after is imported- to be used with extreme care (by default empty). Only available in microservice type.
schemaOnly _bool_	When set to true, only the *pre-data* and *post-data* sections of *pg_restore* are invoked, avoiding data import. Default: *false*.
pgDumpExtraOptions _[]string_	List of custom options to pass to the *pg_dump* command. IMPORTANT: Use these options with caution and at your own risk, as the operator does not validate their content. Be aware that certain options may conflict with the operator's intended functionality or design.
pgRestoreExtraOptions _[]string_	List of custom options to pass to the *pg_restore* command. IMPORTANT: Use these options with caution and at your own risk, as the operator does not validate their content. Be aware that certain options may conflict with the operator's intended functionality or design.

55.55 ImportSource

Appears in:

- Import {postgresql-cnpg-io-v1-Import}

ImportSource describes the source for the logical snapshot

Field	Description
externalCluster **[Required]** _string_	The name of the externalCluster used for import

55.56 InstanceID

Appears in:

- BackupStatus {postgresql-cnpg-io-v1-BackupStatus}

InstanceID contains the information to identify an instance

Field	Description
podName _string_	The pod name
ContainerID _string_	The container ID

55.57 InstanceReportedState

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

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

Field	Description
isPrimary **[Required]** _bool_	indicates if an instance is the primary one
timeLineID _int_	indicates on which TimelineId the instance is
ip **[Required]** _string_	IP address of the instance

55.58 IsolationCheckConfiguration

Appears in:

- LivenessProbe {postgresql-cnpg-io-v1-LivenessProbe}

IsolationCheckConfiguration contains the configuration for the isolation check functionality in the liveness probe

Field	Description
enabled _bool_	Whether primary isolation checking is enabled for the liveness probe
requestTimeout _int_	Timeout in milliseconds for requests during the primary isolation check
connectionTimeout _int_	Timeout in milliseconds for connections during the primary isolation check

55.59 LDAPBindAsAuth

Appears in:

- LDAPConfig {postgresql-cnpg-io-v1-LDAPConfig}

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

Field	Description
prefix _string_	Prefix for the bind authentication option

continues on next page

Table 52 – continued from previous page

Field	Description
suffix _string_	Suffix for the bind authentication option

55.60 LDAPBindSearchAuth

Appears in:

- LDAPConfig {postgresql-cnpg-io-v1-LDAPConfig}

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

Field	Description
baseDN _string_	Root DN to begin the user search
bindDN _string_	DN of the user to bind to the directory
bindPassword _core/v1.SecretKeySelector_	Secret with the password for the user to bind to the directory
searchAttribute _string_	Attribute to match against the username
searchFilter _string_	Search filter to use when doing the search+bind authentication

55.61 LDAPConfig

Appears in:

- PostgresConfiguration {postgresql-cnpg-io-v1-PostgresConfiguration}

LDAPConfig contains the parameters needed for LDAP authentication

Field	Description
server _string_	LDAP hostname or IP address
port _int_	LDAP server port
scheme _LDAPScheme_	LDAP schema to be used, possible options are *ldap* and *ldaps*
bindAsAuth _LDAPBindAsAuth_	Bind as authentication configuration
bindSearchAuth _LDAPBindSearchAuth_	Bind+Search authentication configuration
tls _bool_	Set to 'true' to enable LDAP over TLS. 'false' is default

55.62 LDAPScheme

(Alias of `string`)

Appears in:

- LDAPConfig {postgresql-cnpg-io-v1-LDAPConfig}

LDAPScheme defines the possible schemes for LDAP

55.63 LivenessProbe

Appears in:

- ProbesConfiguration {postgresql-cnpg-io-v1-ProbesConfiguration}

LivenessProbe is the configuration of the liveness probe

Field	Description
Probe _Probe_	(Members of *Probe* are embedded into this type.) Probe is the standard probe configuration
isolationCheck _IsolationCheckConfiguration_	Configure the feature that extends the liveness probe for a primary instance. In addition to the basic checks, this verifies whether the primary is isolated from the Kubernetes API server and from its replicas, ensuring that it can be safely shut down if network partition or API unavailability is detected. Enabled by default.

55.64 ManagedConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ManagedConfiguration represents the portions of PostgreSQL that are managed by the instance manager

Field	Description
roles [_[]RoleConfiguration_](RoleConfiguration)	Database roles managed by the *Cluster*
services _ManagedServices_	Services roles managed by the *Cluster*

55.65 ManagedRoles

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

ManagedRoles tracks the status of a cluster's managed roles

Field	Description
byStatus _map[RoleStatus][]string_	ByStatus gives the list of roles in each state
cannotReconcile _map[string][]string_	CannotReconcile lists roles that cannot be reconciled in PostgreSQL, with an explanation of the cause
passwordStatus [_map[string]PasswordState_](PasswordState)	PasswordStatus gives the last transaction id and password secret version for each managed role

55.66 ManagedService

Appears in:

- ManagedServices {postgresql-cnpg-io-v1-ManagedServices}

ManagedService represents a specific service managed by the cluster. It includes the type of service and its associated template specification.

Field	Description
selectorType **[Required]** *_ServiceSelectorType_*	SelectorType specifies the type of selectors that the service will have. Valid values are "rw", "r", and "ro", representing read-write, read, and read-only services.
updateStrategy *_ServiceUpdateStrategy_*	UpdateStrategy describes how the service differences should be reconciled
serviceTemplate **[Required]** *_ServiceTemplateSpec_*	ServiceTemplate is the template specification for the service.

55.67 ManagedServices

Appears in:

- ManagedConfiguration {postgresql-cnpg-io-v1-ManagedConfiguration}

ManagedServices represents the services managed by the cluster.

Field	Description
disabledDefaultServices [_]ServiceSelectorType_](ServiceSelectorType)	DisabledDefaultServices is a list of service types that are disabled by default. Valid values are "r", and "ro", representing read, and read-only services.
additional [_]ManagedService_](ManagedService)	Additional is a list of additional managed services specified by the user.

55.68 Metadata

Appears in:

- PodTemplateSpec {postgresql-cnpg-io-v1-PodTemplateSpec}

Metadata is a structure similar to the metav1.ObjectMeta, but still parseable by controller-gen to create a suitable CRD for the user. The comment of PodTemplateSpec has an explanation of why we are not using the core data types.

Field	Description
name _string_	The name of the resource. Only supported for certain types
labels _map[string]string_	Map of string keys and values that can be used to organize and categorize (scope and select) objects. May match selectors of replication controllers and services. More info: http://kubernetes.io/docs/user-guide/labels
annotations _map[string]string_	Annotations is an unstructured key value map stored with a resource that may be set by external tools to store and retrieve arbitrary metadata. They are not queryable and should be preserved when modifying objects. More info: http://kubernetes.io/docs/user-guide/annotations

55.69 MonitoringConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

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

Field	Description
disableDefaultQueries _bool_	Whether the default queries should be injected. Set it to *true* if you don't want to inject default queries into the cluster. Default: false.
customQueriesConfigMap [_]github.com/cloudnative-pg/machinery/ //pkg.go.dev/github.com/ cloudnative-pg/machinery/pkg/api/ #ConfigMapKeySelector)	The list of config maps containing the custom queries
customQueriesSecret [_]github.com/cloudnative-pg/machinery/ //pkg.go.dev/github.com/cloudnative-pg/ machinery/pkg/api/#SecretKeySelector)	The list of secrets containing the custom queries
enablePodMonitor _bool_	Enable or disable the *PodMonitor*
tls *_ClusterMonitoringTLSConfiguration_*	Configure TLS communication for the metrics endpoint. Changing tls.enabled option will force a rollout of all instances.
podMonitorMetricRelabelings [_]github.com/prometheus-operator/prom //pkg.go.dev/github.com/ prometheus-operator/ prometheus-operator/pkg/apis/ monitoring/v1#RelabelConfig)	The list of metric relabelings for the *PodMonitor*. Applied to samples before ingestion.
podMonitorRelabelings [_]github.com/prometheus-operator/prom //pkg.go.dev/github.com/ prometheus-operator/ prometheus-operator/pkg/apis/ monitoring/v1#RelabelConfig)	The list of relabelings for the *PodMonitor*. Applied to samples before scraping.

55.70 NodeMaintenanceWindow

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

NodeMaintenanceWindow contains information that the operator will use while upgrading the underlying node.

This option is only useful when the chosen storage prevents the Pods from being freely moved across nodes.

Field	Description
reusePVC _bool_	Reuse the existing PVC (wait for the node to come up again) or not (recreate it elsewhere- when *instances* >1)

continues on next page

Table 62 – continued from previous page

Field	Description
inProgress _bool_	Is there a node maintenance activity in progress?

55.71 OnlineConfiguration

Appears in:

- BackupSpec {postgresql-cnpg-io-v1-BackupSpec}

OnlineConfiguration contains the configuration parameters for the online volume snapshot

Field	Description
waitForArchive _bool_	If false, the function will return immediately after the backup is completed, without waiting for WAL to be archived. This behavior is only useful with backup software that independently monitors WAL archiving. Otherwise, WAL required to make the backup consistent might be missing and make the backup useless. By default, or when this parameter is true, pg_backup_stop will wait for WAL to be archived when archiving is enabled. On a standby, this means that it will wait only when archive_mode = always. If write activity on the primary is low, it may be useful to run pg_switch_wal on the primary in order to trigger an immediate segment switch.
immediateCheckpoint _bool_	Control whether the I/O workload for the backup initial checkpoint will be limited, according to the *checkpoint_completion_target* setting on the PostgreSQL server. If set to true, an immediate checkpoint will be used, meaning PostgreSQL will complete the checkpoint as soon as possible. *false* by default.

55.72 OptionSpec

Appears in:

- FDWSpec {postgresql-cnpg-io-v1-FDWSpec}

OptionSpec holds the name, value and the ensure field for an option

Field	Description
name **[Required]** _string_	Name of the option
OptionSpecValue *_OptionSpecValue_*	(Members of *OptionSpecValue* are embedded into this type.) Value and ensure field of the option

55.73 OptionSpecValue

Appears in:

- OptionSpec {postgresql-cnpg-io-v1-OptionSpec}

OptionSpecValue holds the value and the ensure field for an option

Field	Description
value **[Required]** _string_	Value of the option
ensure *_EnsureOption_*	Specifies whether an option should be present or absent in the database. If set to *present*, the option will be created if it does not exist. If set to *absent*, the option will be removed if it exists.

55.74 PasswordState

Appears in:

- ManagedRoles {postgresql-cnpg-io-v1-ManagedRoles}

PasswordState represents the state of the password of a managed RoleConfiguration

Field	Description
transactionID _int64_	the last transaction ID to affect the role definition in PostgreSQL
resourceVersion _string_	the resource version of the password secret

55.75 PgBouncerIntegrationStatus

Appears in:

- PoolerIntegrations {postgresql-cnpg-io-v1-PoolerIntegrations}

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

Field	Description
secrets _[]string_	No description provided.

55.76 PgBouncerPoolMode

(Alias of `string`)

Appears in:

- PgBouncerSpec {postgresql-cnpg-io-v1-PgBouncerSpec}

PgBouncerPoolMode is the mode of PgBouncer

55.77 PgBouncerSecrets

Appears in:

- PoolerSecrets {postgresql-cnpg-io-v1-PoolerSecrets}

PgBouncerSecrets contains the versions of the secrets used by pgbouncer

Field	Description
authQuery *_SecretVersion_*	The auth query secret version

55.78 PgBouncerSpec

Appears in:

- PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec}

PgBouncerSpec defines how to configure PgBouncer

Field	Description
poolMode _PgBouncerPoolMode_	The pool mode. Default: *session*.
authQuerySecret _github.com/cloudnative-pg/machinery/pk	The credentials of the user that need to be used for the authentication query. In case it is specified, also an AuthQuery (e.g. "SELECT username, passwd FROM pg_catalog.pg_shadow WHERE usename=$1") has to be specified and no automatic CNPG Cluster integration will be triggered.
authQuery _string_	The query that will be used to download the hash of the password of a certain user. Default: "SELECT usename, passwd FROM public.user_search($1)". In case it is specified, also an AuthQuerySecret has to be specified and no automatic CNPG Cluster integration will be triggered.
parameters _map[string]string_	Additional parameters to be passed to PgBouncer- please check the CNPG documentation for a list of options you can configure
pg_hba _[]string_	PostgreSQL Host Based Authentication rules (lines to be appended to the pg_hba.conf file)
paused _bool_	When set to *true*, PgBouncer will disconnect from the PostgreSQL server, first waiting for all queries to complete, and pause all new client connections until this value is set to *false* (default). Internally, the operator calls PgBouncer's *PAUSE* and *RESUME* commands.

55.79 PluginConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

PluginConfiguration specifies a plugin that need to be loaded for this cluster to be reconciled

Field	Description
name **[Required]** _string_	Name is the plugin name
enabled _bool_	Enabled is true if this plugin will be used
isWALArchiver _bool_	Only one plugin can be declared as WALArchiver. Cannot be active if ".spec.backup.barmanObjectStore" configuration is present.
parameters _map[string]string_	Parameters is the configuration of the plugin

55.80 PluginStatus

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

PluginStatus is the status of a loaded plugin

Field	Description
name **[Required]** _string_	Name is the name of the plugin
version **[Required]** _string_	Version is the version of the plugin loaded by the latest reconciliation loop
capabilities _[]string_	Capabilities are the list of capabilities of the plugin
operatorCapabilities _[]string_	OperatorCapabilities are the list of capabilities of the plugin regarding the reconciler
walCapabilities _[]string_	WALCapabilities are the list of capabilities of the plugin regarding the WAL management
backupCapabilities _[]string_	BackupCapabilities are the list of capabilities of the plugin regarding the Backup management
restoreJobHookCapabilities _[]string_	RestoreJobHookCapabilities are the list of capabilities of the plugin regarding the RestoreJobHook management
status _string_	Status contain the status reported by the plugin through the SetStatusInCluster interface

55.81 PodTemplateSpec

Appears in:

- PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec}

PodTemplateSpec is a structure allowing the user to set a template for Pod generation.

Unfortunately we can't use the corev1.PodTemplateSpec type because the generated CRD won't have the field for the metadata section.

References: https://github.com/kubernetes-sigs/controller-tools/issues/385 https://github.com/kubernetes-sigs/controller-tools/issues/448 https://github.com/prometheus-operator/prometheus-operator/issues/3041

Field	Description
metadata _Metadata_	Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata
spec _core/v1.PodSpec_	Specification of the desired behavior of the pod. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

55.82 PodTopologyLabels

(Alias of `map[string]string`)

Appears in:

- PodTopologyLabels {postgresql-cnpg-io-v1-PodTopologyLabels}

PodTopologyLabels represent the topology of a Pod. map[labelName]labelValue

55.83 PoolerIntegrations

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

PoolerIntegrations encapsulates the needed integration for the poolers referencing the cluster

Field	Description
pgBouncerIntegration _PgBouncerIntegrationStatus_	No description provided.

55.84 PoolerMonitoringConfiguration

Appears in:

- PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec}

PoolerMonitoringConfiguration is the type containing all the monitoring configuration for a certain Pooler.

Mirrors the Cluster's MonitoringConfiguration but without the custom queries part for now.

Field	Description
enablePodMonitor _bool_	Enable or disable the *PodMonitor*
podMonitorMetricRelabelings [_]github.com/prometheus-operator/prom //pkg.go.dev/github.com/prometheus-operator/prometheus-operator/pkg/apis/monitoring/v1#RelabelConfig)	The list of metric relabelings for the *PodMonitor*. Applied to samples before ingestion.
podMonitorRelabelings [_]github.com/prometheus-operator/prom //pkg.go.dev/github.com/prometheus-operator/prometheus-operator/pkg/apis/monitoring/v1#RelabelConfig)	The list of relabelings for the *PodMonitor*. Applied to samples before scraping.

55.85 PoolerSecrets

Appears in:

- PoolerStatus {postgresql-cnpg-io-v1-PoolerStatus}

PoolerSecrets contains the versions of all the secrets used

Field	Description
serverTLS _SecretVersion_	The server TLS secret version
serverCA _SecretVersion_	The server CA secret version
clientCA _SecretVersion_	The client CA secret version
pgBouncerSecrets _PgBouncerSecrets_	The version of the secrets used by PgBouncer

55.86 PoolerSpec

Appears in:

- Pooler {postgresql-cnpg-io-v1-Pooler}

PoolerSpec defines the desired state of Pooler

Field	Description
cluster **[Required]** _github.com/cloudnative-pg/machinery/pk	This is the cluster reference on which the Pooler will work. Pooler name should never match with any cluster name within the same namespace.
type *_PoolerType_*	Type of service to forward traffic to. Default: *rw*.
instances _int32_	The number of replicas we want. Default: 1.
template *_PodTemplateSpec_*	The template of the Pod to be created
pgbouncer **[Required]** *_PgBouncerSpec_*	The PgBouncer configuration
deploymentStrategy _apps/v1.DeploymentStrategy_	The deployment strategy to use for pgbouncer to replace existing pods with new ones
monitoring *_PoolerMonitoringConfiguration_*	The configuration of the monitoring infrastructure of this pooler.
serviceTemplate *_ServiceTemplateSpec_*	Template for the Service to be created

55.87 PoolerStatus

Appears in:

- Pooler {postgresql-cnpg-io-v1-Pooler}

PoolerStatus defines the observed state of Pooler

Field	Description
secrets *_PoolerSecrets_*	The resource version of the config object
instances _int32_	The number of pods trying to be scheduled

55.88 PoolerType

(Alias of `string`)

Appears in:

- PoolerSpec {postgresql-cnpg-io-v1-PoolerSpec}

PoolerType is the type of the connection pool, meaning the service we are targeting. Allowed values are `rw` and `ro`.

55.89 PostgresConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

PostgresConfiguration defines the PostgreSQL configuration

Field	Description
parameters _map[string]string_	PostgreSQL configuration options (postgresql.conf)
synchronous *_SynchronousReplicaConfiguration_*	Configuration of the PostgreSQL synchronous replication feature

continues on next page

Table 78 – continued from previous page

Field	Description
pg_hba _[]string_	PostgreSQL Host Based Authentication rules (lines to be appended to the pg_hba.conf file)
pg_ident _[]string_	PostgreSQL User Name Maps rules (lines to be appended to the pg_ident.conf file)
syncReplicaElectionConstraint _SyncReplicaElectionConstraints_	Requirements to be met by sync replicas. This will affect how the "synchronous_standby_names" parameter will be set up.
shared_preload_libraries _[]string_	Lists of shared preload libraries to add to the default ones
ldap _LDAPConfig_	Options to specify LDAP configuration
promotionTimeout _int32_	Specifies the maximum number of seconds to wait when promoting an instance to primary. Default value is 40000000, greater than one year in seconds, big enough to simulate an infinite timeout
enableAlterSystem _bool_	If this parameter is true, the user will be able to invoke *ALTER SYSTEM* on this CloudNativePG Cluster. This should only be used for debugging and troubleshooting. Defaults to false.
extensions [_[]ExtensionConfiguration_](ExtensionConfiguration)	The configuration of the extensions to be added

55.90 PrimaryUpdateMethod

(Alias of `string`)

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

PrimaryUpdateMethod contains the method to use when upgrading the primary server of the cluster as part of rolling updates

55.91 PrimaryUpdateStrategy

(Alias of `string`)

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

PrimaryUpdateStrategy contains the strategy to follow when upgrading the primary server of the cluster as part of rolling updates

55.92 Probe

Appears in:

- LivenessProbe {postgresql-cnpg-io-v1-LivenessProbe}

Probe describes a health check to be performed against a container to determine whether it is alive or ready to receive traffic.

Field	Description
initialDelaySeconds _int32_	Number of seconds after the container has started before liveness probes are initiated. More info: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle#container-probes
timeoutSeconds _int32_	Number of seconds after which the probe times out. Defaults to 1 second. Minimum value is 1. More info: https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle#container-probes
periodSeconds _int32_	How often (in seconds) to perform the probe. Default to 10 seconds. Minimum value is 1.
successThreshold _int32_	Minimum consecutive successes for the probe to be considered successful after having failed. Defaults to 1. Must be 1 for liveness and startup. Minimum value is 1.
failureThreshold _int32_	Minimum consecutive failures for the probe to be considered failed after having succeeded. Defaults to 3. Minimum value is 1.
terminationGracePeriodSeconds _int64_	Optional duration in seconds the pod needs to terminate gracefully upon probe failure. The grace period is the duration in seconds after the processes running in the pod are sent a termination signal and the time when the processes are forcibly halted with a kill signal. Set this value longer than the expected cleanup time for your process. If this value is nil, the pod's terminationGracePeriodSeconds will be used. Otherwise, this value overrides the value provided by the pod spec. Value must be non-negative integer. The value zero indicates stop immediately via the kill signal (no opportunity to shut down). This is a beta field and requires enabling ProbeTerminationGracePeriod feature gate. Minimum value is 1. spec.terminationGracePeriodSeconds is used if unset.

55.93 ProbeStrategyType

(Alias of `string`)

Appears in:

- ProbeWithStrategy {postgresql-cnpg-io-v1-ProbeWithStrategy}

ProbeStrategyType is the type of the strategy used to declare a PostgreSQL instance ready

55.94 ProbeWithStrategy

Appears in:

- ProbesConfiguration {postgresql-cnpg-io-v1-ProbesConfiguration}

ProbeWithStrategy is the configuration of the startup probe

Field	Description
Probe *_Probe_*	(Members of *Probe* are embedded into this type.) Probe is the standard probe configuration
type *_ProbeStrategyType_*	The probe strategy
maximumLag _k8s.io/apimachinery/pkg/api/resource.Quantity_	Lag limit. Used only for *streaming* strategy

55.95 ProbesConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ProbesConfiguration represent the configuration for the probes to be injected in the PostgreSQL Pods

Field	Description
startup **[Required]** *_ProbeWithStrategy_*	The startup probe configuration
liveness **[Required]** *_LivenessProbe_*	The liveness probe configuration
readiness **[Required]** *_ProbeWithStrategy_*	The readiness probe configuration

55.96 PublicationReclaimPolicy

(Alias of `string`)

Appears in:

- PublicationSpec {postgresql-cnpg-io-v1-PublicationSpec}

PublicationReclaimPolicy defines a policy for end-of-life maintenance of Publications.

55.97 PublicationSpec

Appears in:

- Publication {postgresql-cnpg-io-v1-Publication}

PublicationSpec defines the desired state of Publication

Field	Description
cluster **[Required]** _core/v1.LocalObjectReference_	The name of the PostgreSQL cluster that identifies the "publisher"
name **[Required]** _string_	The name of the publication inside PostgreSQL
dbname **[Required]** _string_	The name of the database where the publication will be installed in the "publisher" cluster
parameters _map[string]string_	Publication parameters part of the *WITH* clause as expected by PostgreSQL *CREATE PUBLICATION* command
target **[Required]** *_PublicationTarget_*	Target of the publication as expected by PostgreSQL *CREATE PUBLICATION* command
publicationReclaimPolicy *_PublicationReclaimPolicy_*	The policy for end-of-life maintenance of this publication

55.98 PublicationStatus

Appears in:

- Publication {postgresql-cnpg-io-v1-Publication}

PublicationStatus defines the observed state of Publication

Field	Description
observedGeneration _int64_	A sequence number representing the latest desired state that was synchronized
applied _bool_	Applied is true if the publication was reconciled correctly
message _string_	Message is the reconciliation output message

55.99 PublicationTarget

Appears in:

- PublicationSpec {postgresql-cnpg-io-v1-PublicationSpec}

PublicationTarget is what this publication should publish

Field	Description
allTables _bool_	Marks the publication as one that replicates changes for all tables in the database, including tables created in the future. Corresponding to *FOR ALL TABLES* in PostgreSQL.
objects [_[]PublicationTargetObject_](PublicationTargetObject)	Just the following schema objects

55.100 PublicationTargetObject

Appears in:

- PublicationTarget {postgresql-cnpg-io-v1-PublicationTarget}

PublicationTargetObject is an object to publish

Field	Description
tablesInSchema _string_	Marks the publication as one that replicates changes for all tables in the specified list of schemas, including tables created in the future. Corresponding to *FOR TABLES IN SCHEMA* in PostgreSQL.
table *_PublicationTargetTable_*	Specifies a list of tables to add to the publication. Corresponding to *FOR TABLE* in PostgreSQL.

55.101 PublicationTargetTable

Appears in:

- PublicationTargetObject {postgresql-cnpg-io-v1-PublicationTargetObject}

PublicationTargetTable is a table to publish

Field	Description
only _bool_	Whether to limit to the table only or include all its descendants
name **[Required]** _string_	The table name
schema _string_	The schema name

continues on next page

Table 86 – continued from previous page

Field	Description
columns _[]string_	The columns to publish

55.102 RecoveryTarget

Appears in:

- BootstrapRecovery {postgresql-cnpg-io-v1-BootstrapRecovery}

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

Field	Description
backupID _string_	The ID of the backup from which to start the recovery process. If empty (default) the operator will automatically detect the backup based on targetTime or targetLSN if specified. Otherwise use the latest available backup in chronological order.
targetTLI _string_	The target timeline ("latest" or a positive integer)
targetXID _string_	The target transaction ID
targetName _string_	The target name (to be previously created with *pg_create_restore_point*)
targetLSN _string_	The target LSN (Log Sequence Number)
targetTime _string_	The target time as a timestamp in the RFC3339 standard
targetImmediate _bool_	End recovery as soon as a consistent state is reached
exclusive _bool_	Set the target to be exclusive. If omitted, defaults to false, so that in Postgres, *recovery_target_inclusive* will be true

55.103 ReplicaClusterConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ReplicaClusterConfiguration encapsulates the configuration of a replica cluster

Field	Description
self _string_	Self defines the name of this cluster. It is used to determine if this is a primary or a replica cluster, comparing it with *primary*
primary _string_	Primary defines which Cluster is defined to be the primary in the distributed PostgreSQL cluster, based on the topology specified in externalClusters
source **[Required]** _string_	The name of the external cluster which is the replication origin
enabled _bool_	If replica mode is enabled, this cluster will be a replica of an existing cluster. Replica cluster can be created from a recovery object store or via streaming through pg_basebackup. Refer to the Replica clusters page of the documentation for more information.
promotionToken _string_	A demotion token generated by an external cluster used to check if the promotion requirements are met.

continues on next page

Table 88 – continued from previous page

Field	Description
minApplyDelay _meta/v1.Duration_	When replica mode is enabled, this parameter allows you to replay transactions only when the system time is at least the configured time past the commit time. This provides an opportunity to correct data loss errors. Note that when this parameter is set, a promotion token cannot be used.

55.104 ReplicationSlotsConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ReplicationSlotsConfiguration encapsulates the configuration of replication slots

Field	Description
highAvailability *_ReplicationSlotsHA-Configuration_*	Replication slots for high availability configuration
updateInterval _int_	Standby will update the status of the local replication slots every *updateInterval* seconds (default 30).
synchronizeReplicas *_SynchronizeReplicasConfiguration_*	Configures the synchronization of the user defined physical replication slots

55.105 ReplicationSlotsHAConfiguration

Appears in:

- ReplicationSlotsConfiguration {postgresql-cnpg-io-v1-ReplicationSlotsConfiguration}

ReplicationSlotsHAConfiguration encapsulates the configuration of the replication slots that are automatically managed by the operator to control the streaming replication connections with the standby instances for high availability (HA) purposes. Replication slots are a PostgreSQL feature that makes sure that PostgreSQL automatically keeps WAL files in the primary when a streaming client (in this specific case a replica that is part of the HA cluster) gets disconnected.

Field	Description
enabled _bool_	If enabled (default), the operator will automatically manage replication slots on the primary instance and use them in streaming replication connections with all the standby instances that are part of the HA cluster. If disabled, the operator will not take advantage of replication slots in streaming connections with the replicas. This feature also controls replication slots in replica cluster, from the designated primary to its cascading replicas.
slotPrefix _string_	Prefix for replication slots managed by the operator for HA. It may only contain lower case letters, numbers, and the underscore character. This can only be set at creation time. By default set to *_cnpg_*.

continues on next page

Table 90 – continued from previous page

Field	Description
synchronizeLogicalDecoding _bool_	When enabled, the operator automatically manages synchronization of logical decoding (replication) slots across high-availability clusters. Requires one of the following conditions: *PostgreSQL version 17 or later* PostgreSQL version < 17 with pg_failover_slots extension enabled

55.106 RoleConfiguration

Appears in:

- ManagedConfiguration {postgresql-cnpg-io-v1-ManagedConfiguration}

RoleConfiguration is the representation, in Kubernetes, of a PostgreSQL role with the additional field Ensure specifying whether to ensure the presence or absence of the role in the database

The defaults of the CREATE ROLE command are applied Reference: https://www.postgresql.org/docs/current/sql-createrole.html

Field	Description
name **[Required]** _string_	Name of the role
comment _string_	Description of the role
ensure _EnsureOption_	Ensure the role is *present* or *absent*- defaults to "present"
passwordSecret _github.com/cloudnative-pg/machinery/pk	Secret containing the password of the role (if present) If null, the password will be ignored unless DisablePassword is set
connectionLimit _int64_	If the role can log in, this specifies how many concurrent connections the role can make. *-1* (the default) means no limit.
validUntil _meta/v1.Time_	Date and time after which the role's password is no longer valid. When omitted, the password will never expire (default).
inRoles _[]string_	List of one or more existing roles to which this role will be immediately added as a new member. Default empty.
inherit _bool_	Whether a role "inherits" the privileges of roles it is a member of. Defaults is *true*.
disablePassword _bool_	DisablePassword indicates that a role's password should be set to NULL in Postgres
superuser _bool_	Whether the role is a *superuser* who can override all access restrictions within the database- superuser status is dangerous and should be used only when really needed. You must yourself be a superuser to create a new superuser. Defaults is *false*.
createdb _bool_	When set to *true*, the role being defined will be allowed to create new databases. Specifying *false* (default) will deny a role the ability to create databases.
createrole _bool_	Whether the role will be permitted to create, alter, drop, comment on, change the security label for, and grant or revoke membership in other roles. Default is *false*.
login _bool_	Whether the role is allowed to log in. A role having the *login* attribute can be thought of as a user. Roles without this attribute are useful for managing database privileges, but are not users in the usual sense of the word. Default is *false*.

continues on next page

Table 91 – continued from previous page

Field	Description
replication _bool_	Whether a role is a replication role. A role must have this attribute (or be a superuser) in order to be able to connect to the server in replication mode (physical or logical replication) and in order to be able to create or drop replication slots. A role having the *replication* attribute is a very highly privileged role, and should only be used on roles actually used for replication. Default is *false*.
bypassrls _bool_	Whether a role bypasses every row-level security (RLS) policy. Default is *false*.

55.107 SQLRefs

Appears in:

- BootstrapInitDB {postgresql-cnpg-io-v1-BootstrapInitDB}

SQLRefs holds references to ConfigMaps or Secrets containing SQL files. The references are processed in a specific order: first, all Secrets are processed, followed by all ConfigMaps. Within each group, the processing order follows the sequence specified in their respective arrays.

Field	Description
secretRefs [_[]github.com/cloudnative-pg/machinery/ //pkg.go.dev/github.com/cloudnative-pg/machinery/pkg/api/#SecretKeySelector)	SecretRefs holds a list of references to Secrets
configMapRefs [_[]github.com/cloudnative-pg/machinery/ //pkg.go.dev/github.com/cloudnative-pg/machinery/pkg/api/#ConfigMapKeySelector)	ConfigMapRefs holds a list of references to ConfigMaps

55.108 ScheduledBackupSpec

Appears in:

- ScheduledBackup {postgresql-cnpg-io-v1-ScheduledBackup}

ScheduledBackupSpec defines the desired state of ScheduledBackup

Field	Description
suspend _bool_	If this backup is suspended or not
immediate _bool_	If the first backup has to be immediately start after creation or not
schedule **[Required]** _string_	The schedule does not follow the same format used in Kubernetes CronJobs as it includes an additional seconds specifier, see https://pkg.go.dev/github.com/robfig/cron#hdr-CRON_Expression_Format
cluster **[Required]** _github.com/cloudnative-pg/machinery/pk	The cluster to backup

continues on next page

Table 93 – continued from previous page

Field	Description
backupOwnerReference _string_	Indicates which ownerReference should be put inside the created backup resources. *none: no owner reference for created backup objects (same behavior as before the field was introduced)* self: sets the Scheduled backup object as owner of the backup * cluster: set the cluster as owner of the backup
target *_BackupTarget_*	The policy to decide which instance should perform this backup. If empty, it defaults to *cluster.spec.backup.target*. Available options are empty string, *primary* and *prefer-standby*. *primary* to have backups run always on primary instances, *prefer-standby* to have backups run preferably on the most updated standby, if available.
method *_BackupMethod_*	The backup method to be used, possible options are *barmanObjectStore*, *volumeSnapshot* or *plugin*. Defaults to: *barmanObjectStore*.
pluginConfiguration *_BackupPluginConfiguration_*	Configuration parameters passed to the plugin managing this backup
online _bool_	Whether the default type of backup with volume snapshots is online/hot (*true*, default) or offline/cold (*false*) Overrides the default setting specified in the cluster field '.spec.backup.volumeSnapshot.online'
onlineConfiguration *_OnlineConfiguration_*	Configuration parameters to control the online/hot backup with volume snapshots Overrides the default settings specified in the cluster '.backup.volumeSnapshot.onlineConfiguration' stanza

55.109 ScheduledBackupStatus

Appears in:

- ScheduledBackup {postgresql-cnpg-io-v1-ScheduledBackup}

ScheduledBackupStatus defines the observed state of ScheduledBackup

Field	Description
lastCheckTime _meta/v1.Time_	The latest time the schedule
lastScheduleTime _meta/v1.Time_	Information when was the last time that backup was successfully scheduled.
nextScheduleTime _meta/v1.Time_	Next time we will run a backup

55.110 SchemaSpec

Appears in:

- DatabaseSpec {postgresql-cnpg-io-v1-DatabaseSpec}

SchemaSpec configures a schema in a database

Field	Description
DatabaseObjectSpec *_DatabaseObjectSpec_*	(Members of *DatabaseObjectSpec* are embedded into this type.) Common fields

continues on next page

Table 95 – continued from previous page

Field	Description
owner **[Required]** _string_	The role name of the user who owns the schema inside PostgreSQL. It maps to the *AUTHORIZATION* parameter of *CREATE SCHEMA* and the *OWNER TO* command of *ALTER SCHEMA*.

55.111 SecretVersion

Appears in:

- PgBouncerSecrets {postgresql-cnpg-io-v1-PgBouncerSecrets}

SecretVersion contains a secret name and its ResourceVersion

Field	Description
name _string_	The name of the secret
version _string_	The ResourceVersion of the secret

55.112 SecretsResourceVersion

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

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

Field	Description
superuserSecretVersion _string_	The resource version of the "postgres" user secret
replicationSecretVersion _string_	The resource version of the "streaming_replica" user secret
applicationSecretVersion _string_	The resource version of the "app" user secret
managedRoleSecretVersion _map[string]string_	The resource versions of the managed roles secrets
caSecretVersion _string_	Unused. Retained for compatibility with old versions.
clientCaSecretVersion _string_	The resource version of the PostgreSQL client-side CA secret version
serverCaSecretVersion _string_	The resource version of the PostgreSQL server-side CA secret version
serverSecretVersion _string_	The resource version of the PostgreSQL server-side secret version
barmanEndpointCA _string_	The resource version of the Barman Endpoint CA if provided
externalClusterSecretVersion _map[string]string_	The resource versions of the external cluster secrets
metrics _map[string]string_	A map with the versions of all the secrets used to pass metrics. Map keys are the secret names, map values are the versions

55.113 ServiceAccountTemplate

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

ServiceAccountTemplate contains the template needed to generate the service accounts

Field	Description
metadata **[Required]** *_Metadata_*	Metadata are the metadata to be used for the generated service account

55.114 ServiceSelectorType

(Alias of `string`)

Appears in:

- ManagedService {postgresql-cnpg-io-v1-ManagedService}

ServiceSelectorType describes a valid value for generating the service selectors. It indicates which type of service the selector applies to, such as read-write, read, or read-only

55.115 ServiceTemplateSpec

Appears in:

- ManagedService {postgresql-cnpg-io-v1-ManagedService}

ServiceTemplateSpec is a structure allowing the user to set a template for Service generation.

Field	Description
metadata *_Metadata_*	Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata
spec _core/v1.ServiceSpec_	Specification of the desired behavior of the service. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

55.116 ServiceUpdateStrategy

(Alias of `string`)

Appears in:

- ManagedService {postgresql-cnpg-io-v1-ManagedService}

ServiceUpdateStrategy describes how the changes to the managed service should be handled

55.117 SnapshotOwnerReference

(Alias of `string`)

Appears in:

- VolumeSnapshotConfiguration {postgresql-cnpg-io-v1-VolumeSnapshotConfiguration}

SnapshotOwnerReference defines the reference type for the owner of the snapshot. This specifies which owner the processed resources should relate to.

55.118 SnapshotType

(Alias of `string`)

Appears in:

- Import {postgresql-cnpg-io-v1-Import}

SnapshotType is a type of allowed import

55.119 StorageConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

StorageConfiguration is the configuration used to create and reconcile PVCs, usable for WAL volumes, PGDATA volumes, or tablespaces

Field	Description
storageClass _string_	StorageClass to use for PVCs. Applied after evaluating the PVC template, if available. If not specified, the generated PVCs will use the default storage class
size _string_	Size of the storage. Required if not already specified in the PVC template. Changes to this field are automatically reapplied to the created PVCs. Size cannot be decreased.
resizeInUseVolumes _bool_	Resize existent PVCs, defaults to true
pvcTemplate _core/v1.PersistentVolumeClaimSpec_	Template to be used to generate the Persistent Volume Claim

55.120 SubscriptionReclaimPolicy

(Alias of `string`)

Appears in:

- SubscriptionSpec {postgresql-cnpg-io-v1-SubscriptionSpec}

SubscriptionReclaimPolicy describes a policy for end-of-life maintenance of Subscriptions.

55.121 SubscriptionSpec

Appears in:

- Subscription {postgresql-cnpg-io-v1-Subscription}

SubscriptionSpec defines the desired state of Subscription

Field	Description
cluster **[Required]** _core/v1.LocalObjectReference_	The name of the PostgreSQL cluster that identifies the "subscriber"
name **[Required]** _string_	The name of the subscription inside PostgreSQL

continues on next page

Table 101 – continued from previous page

Field	Description
dbname **[Required]** _string_	The name of the database where the publication will be installed in the "subscriber" cluster
parameters _map[string]string_	Subscription parameters included in the *WITH* clause of the PostgreSQL *CREATE SUBSCRIPTION* command. Most parameters cannot be changed after the subscription is created and will be ignored if modified later, except for a limited set documented at: https://www.postgresql.org/docs/current/sql-altersubscription.html#SQL-ALTERSUBSCRIPTION-PARAMS-SET
publicationName **[Required]** _string_	The name of the publication inside the PostgreSQL database in the "publisher"
publicationDBName _string_	The name of the database containing the publication on the external cluster. Defaults to the one in the external cluster definition.
externalClusterName **[Required]** _string_	The name of the external cluster with the publication ("publisher")
subscriptionReclaimPolicy *_SubscriptionReclaimPolicy_*	The policy for end-of-life maintenance of this subscription

55.122 SubscriptionStatus

Appears in:

- Subscription {postgresql-cnpg-io-v1-Subscription}

SubscriptionStatus defines the observed state of Subscription

Field	Description
observedGeneration _int64_	A sequence number representing the latest desired state that was synchronized
applied _bool_	Applied is true if the subscription was reconciled correctly
message _string_	Message is the reconciliation output message

55.123 SwitchReplicaClusterStatus

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

SwitchReplicaClusterStatus contains all the statuses regarding the switch of a cluster to a replica cluster

Field	Description
inProgress _bool_	InProgress indicates if there is an ongoing procedure of switching a cluster to a replica cluster.

55.124 SyncReplicaElectionConstraints

Appears in:

- PostgresConfiguration {postgresql-cnpg-io-v1-PostgresConfiguration}

SyncReplicaElectionConstraints contains the constraints for sync replicas election.

For anti-affinity parameters two instances are considered in the same location if all the labels values match.

In future synchronous replica election restriction by name will be supported.

Field	Description
nodeLabelsAntiAffinity _[]string_	A list of node labels values to extract and compare to evaluate if the pods reside in the same topology or not
enabled **[Required]** _bool_	This flag enables the constraints for sync replicas

55.125 SynchronizeReplicasConfiguration

Appears in:

- ReplicationSlotsConfiguration {postgresql-cnpg-io-v1-ReplicationSlotsConfiguration}

SynchronizeReplicasConfiguration contains the configuration for the synchronization of user defined physical replication slots

Field	Description
enabled **[Required]** _bool_	When set to true, every replication slot that is on the primary is synchronized on each standby
excludePatterns _[]string_	List of regular expression patterns to match the names of replication slots to be excluded (by default empty)

55.126 SynchronousReplicaConfiguration

Appears in:

- PostgresConfiguration {postgresql-cnpg-io-v1-PostgresConfiguration}

SynchronousReplicaConfiguration contains the configuration of the PostgreSQL synchronous replication feature. Important: at this moment, also `.spec.minSyncReplicas` and `.spec.maxSyncReplicas` need to be considered.

Field	Description
method **[Required]** *archive_timeout* as values.	
number **[Required]** _int_	Specifies the number of synchronous standby servers that transactions must wait for responses from.
maxStandbyNamesFromCluster _int_	Specifies the maximum number of local cluster pods that can be automatically included in the *synchronous_standby_names* option in PostgreSQL.
standbyNamesPre _[]string_	A user-defined list of application names to be added to *synchronous_standby_names* before local cluster pods (the order is only useful for priority-based synchronous replication).
standbyNamesPost _[]string_	A user-defined list of application names to be added to *synchronous_standby_names* after local cluster pods (the order is only useful for priority-based synchronous replication).

continues on next page

Table 106 – continued from previous page

Field	Description
dataDurability _DataDurabilityLevel_	If set to "required", data durability is strictly enforced. Write operations with synchronous commit settings (*on*, *remote_write*, or *remote_apply*) will block if there are insufficient healthy replicas, ensuring data persistence. If set to "preferred", data durability is maintained when healthy replicas are available, but the required number of instances will adjust dynamically if replicas become unavailable. This setting relaxes strict durability enforcement to allow for operational continuity. This setting is only applicable if both *standbyNamesPre* and *standbyNamesPost* are unset (empty).

55.127 SynchronousReplicaConfigurationMethod

(Alias of `string`)

Appears in:

- SynchronousReplicaConfiguration {postgresql-cnpg-io-v1-SynchronousReplicaConfiguration}

SynchronousReplicaConfigurationMethod configures whether to use quorum based replication or a priority list

55.128 TablespaceConfiguration

Appears in:

- ClusterSpec {postgresql-cnpg-io-v1-ClusterSpec}

TablespaceConfiguration is the configuration of a tablespace, and includes the storage specification for the tablespace

Field	Description
name **[Required]** _string_	The name of the tablespace
storage **[Required]** _StorageConfiguration_	The storage configuration for the tablespace
owner _DatabaseRoleRef_	Owner is the PostgreSQL user owning the tablespace
temporary _bool_	When set to true, the tablespace will be added as a *temp_tablespaces* entry in PostgreSQL, and will be available to automatically house temp database objects, or other temporary files. Please refer to PostgreSQL documentation for more information on the *temp_tablespaces* GUC.

55.129 TablespaceState

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

TablespaceState represents the state of a tablespace in a cluster

Field	Description
name **[Required]** _string_	Name is the name of the tablespace

continues on next page

Table 108 – continued from previous page

Field	Description
owner _string_	Owner is the PostgreSQL user owning the tablespace
state **[Required]** *_TablespaceStatus_*	State is the latest reconciliation state
error _string_	Error is the reconciliation error, if any

55.130 TablespaceStatus

(Alias of `string`)

Appears in:

- TablespaceState {postgresql-cnpg-io-v1-TablespaceState}

TablespaceStatus represents the status of a tablespace in the cluster

55.131 Topology

Appears in:

- ClusterStatus {postgresql-cnpg-io-v1-ClusterStatus}

Topology contains the cluster topology

Field	Description
instances [_map[PodName]PodTopologyLabels_](PodTopologyLabels)	Instances contains the pod topology of the instances
nodesUsed _int32_	NodesUsed represents the count of distinct nodes accommodating the instances. A value of '1' suggests that all instances are hosted on a single node, implying the absence of High Availability (HA). Ideally, this value should be the same as the number of instances in the Postgres HA cluster, implying shared nothing architecture on the compute side.
successfullyExtracted _bool_	SuccessfullyExtracted indicates if the topology data was extract. It is useful to enact fallback behaviors in synchronous replica election in case of failures

55.132 UsageSpec

Appears in:

- FDWSpec {postgresql-cnpg-io-v1-FDWSpec}

UsageSpec configures a usage for a foreign data wrapper

Field	Description
name **[Required]** _string_	Name of the usage
type _string_	The type of usage

55.133 VolumeSnapshotConfiguration

Appears in:

- BackupConfiguration {postgresql-cnpg-io-v1-BackupConfiguration}

VolumeSnapshotConfiguration represents the configuration for the execution of snapshot backups.

Field	Description
labels _map[string]string_	Labels are key-value pairs that will be added to .metadata.labels snapshot resources.
annotations _map[string]string_	Annotations key-value pairs that will be added to .metadata.annotations snapshot resources.
className _string_	ClassName specifies the Snapshot Class to be used for PG_DATA PersistentVolumeClaim. It is the default class for the other types if no specific class is present
walClassName _string_	WalClassName specifies the Snapshot Class to be used for the PG_WAL PersistentVolumeClaim.
tablespaceClassName _map[string]string_	TablespaceClassName specifies the Snapshot Class to be used for the tablespaces. defaults to the PGDATA Snapshot Class, if set
snapshotOwnerReference *_SnapshotOwnerReference_*	SnapshotOwnerReference indicates the type of owner reference the snapshot should have
online _bool_	Whether the default type of backup with volume snapshots is online/hot (*true*, default) or offline/cold (*false*)
onlineConfiguration *_OnlineConfiguration_*	Configuration parameters to control the online/hot backup with volume snapshots

CHAPTER 56

Supported releases

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

We are committed to providing support for the latest minor release, with a dedication to launching a new minor release every three months. Each release remains fully supported until reaching its designated "End of Life" date, as outlined in the *Support status of CloudNativePG releases* . This includes an additional 3-month assistance window to facilitate seamless upgrade planning.

Supported releases of CloudNativePG include releases that are in the active maintenance window and are patched for security and bug fixes.

Subsequent patch releases on a minor release contain backward-compatible changes only.

- *Support Policy*
- *Naming Scheme*
- *Support status of CloudNativePG releases*
- *What we mean by support*

56.1 Support Policy

CloudNativePG produces new builds for each commit.

Approximately every two months, we create a minor release that undergoes several additional tests and a thorough release qualification process. We release patch versions for issues found in supported minor releases.

Before an official release, at least one Release Candidate (RC) is built for *preview testing* . Additional release candidates may be issued if new bugs are discovered. The Release Candidates are announced on the Slack channel to encourage community testing before the final release. The maintainers provide 1-2 weeks for community testing, and if no objections are raised, the final release is announced.

Different types of releases represent varying levels of product quality and assistance from the CloudNativePG community. For details on the support provided by the community, see *What we mean by support* .

Type	Support level	Quality and recommended Use
Development Build	No support	Dangerous, might not be fully reliable. Useful to experiment with.
Release Candidate	No support	*archive_timeout* . Released for experimentation and testing.
Minor Release	Support provided until 3 months after the N+1 minor release (ex. 1.23 supported until 3 months after 1.24.0 is released)	
Patch	Same as the corresponding minor release	Users are encouraged to adopt patch releases as soon as they are available for a given release.
Security Patch	Same as a patch, however, it doesn't contain any additional code other than the security fix from the previous patch	Given the nature of security fixes, users are **strongly** encouraged to adopt security patches after release.

You can find available releases on the releases page .

You can find high-level more information for each minor and patch release in the *Release notes for 1.14.0 and earlier* .

Sure, here's an improved version of the naming scheme section:

56.2 Naming Scheme

Our naming scheme follows Semantic Versioning 2.0.0 and is structured as follows:

```
<major>.<minor>.<patch>
```

- `<minor>` is incremented for each release.
- `<patch>` counts the number of patches for the current `<minor>` release, representing small changes relative to the `<minor>` release.

Release candidates are indicated by an additional `-<pre-release>` identifier following the patch version, as specified in Semantic Versioning 2.0.0 - item #9 .

Git tags for versions are prefixed with `v` .

56.3 Support status of CloudNativePG releases

Version	Currently supported	Release date	End of life	Supported Kubernetes versions	Tested, but not supported	Supported Postgres versions
1.27.x	Yes	12 Aug 2025	~ Feb 2026	1.31, 1.32, 1.33	1.30, 1.29	13 - 17
1.26.x	Yes	May 23, 2025	12 Nov 2025	1.30, 1.31, 1.32, 1.33	1.29	13 - 17

continues on next page

Table 2 – continued from previous page

Version	Currently supported	Release date	End of life	Supported Kubernetes versions	Tested, but not supported	Supported Postgres versions
1.25.x	Yes	Dec 23, 2024	22 Aug 2025	1.29, 1.30, 1.31, 1.32	1.33	13 - 17
main	No, development only					13 - 17

1 *PostgreSQL 13 will be supported until November 13, 2025.*

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

Currently, the CloudNativePG community does not officially support or test any Kubernetes distributions beyond the standard/vanilla one - such as Red Hat OpenShift. This may change in the future, and if it does, the CloudNativePG maintainers will update the official policy accordingly.

If you plan to deploy CloudNativePG on Red Hat OpenShift, you can use the certified operator provided by EDB , which comes with full support from EDB.

56.3.1 Supported PostgreSQL versions

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

See the PostgreSQL Versioning Policy

page for more information about supported versions.

> **Note**
>
> Starting from November 14, 2024, Postgres 12 is no longer supported .

We also recommend that you regularly update your PostgreSQL operand images and use the latest minor release for the major version you have in use, as not upgrading is riskier than upgrading. As a result, when opening an issue with an older minor version of PostgreSQL, we might not be able to help you.

56.4 Upcoming releases

Version	Release date	End of life
1.28.0	~ Nov, 2025	~ May, 2026
1.29.0	~ Feb, 2026	~ Aug, 2026
1.30.0	~ May, 2026	~ Nov, 2026

> **Note**

Feature freeze occurs 1-2 weeks before the release, at which point a release candidate version is built and distributed for testing, as described earlier.

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.

56.5 Old releases

Version	Release date	End of life	Compatible Kubernetes versions
1.24.x	Aug 22, 2024	May 23, 2025	1.28, 1.29, 1.30, 1.31
1.23.x	April 24, 2024	November 24, 2024	1.27, 1.28, 1.29
1.22.x	December 21, 2023	July 24, 2024	1.26, 1.27, 1.28
1.21.x	October 12, 2023	Jun 12, 2024	1.25, 1.26, 1.27, 1.28
1.20.x	April 27, 2023	January 21, 2024	1.24, 1.25, 1.26, 1.27
1.19.x	February 14, 2023	November 3, 2023	1.23, 1.24, 1.25, 1.26
1.18.x	Nov 10, 2022	June 12, 2023	1.23, 1.24, 1.25, 1.26, 1.27
1.17.x	September 6, 2022	March 20, 2023	1.22, 1.23, 1.24
1.16.x	July 7, 2022	December 21, 2022	1.22, 1.23, 1.24
1.15.x	April 21, 2022	October 6, 2022	1.21, 1.22, 1.23

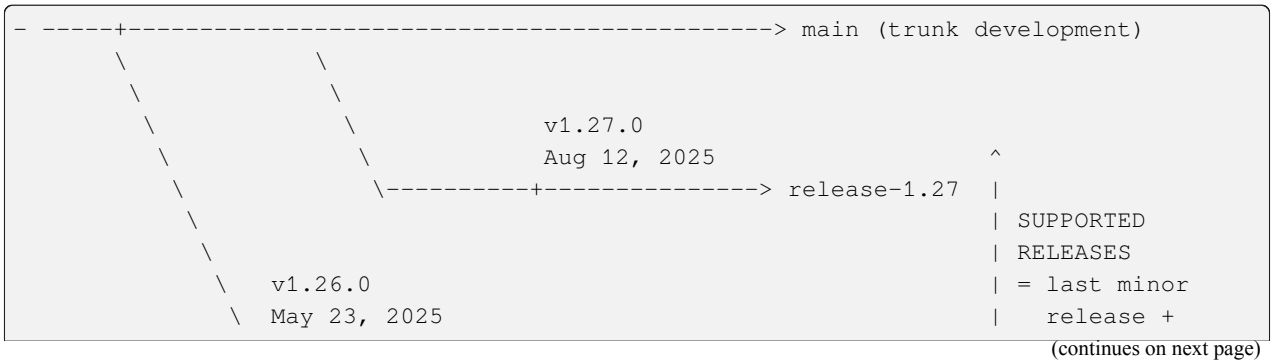
56.6 What we mean by support

Our support window is roughly five/six months for each release branch (latest minor release, plus 3 additional months), given that we produce a new final release every two/three months.

In the following diagram, `release-1.27` is an example of a release branch.

For example, if the latest release is `v1.27.0`, you can expect a supplementary 3-month support period for the preceding release, `v1.26.x`.

Only the last patch release of each branch is supported.



(continues on next page)

(continued from previous page)

```
                +-------------------+---------------> release-1.26  |   3 months
                                                                     v
```

We offer two types of support:

Technical support : Technical assistance is offered on a best-effort basis and is limited to supported releases only. For help, you can reach out to the community via the #cloudnativepg-users

channel on the CNCF Slack workspace (if you're not yet a member, you can join the workspace). Alternatively, you can post your questions in the GitHub Discussions

section of the CloudNativePG repository.

Security and bug fixes : We backport important bug fixes — including security fixes - to all currently supported releases. Before backporting a patch, we ask ourselves: *"Does this backport improve ``CloudNativePG``, bearing in mind that we really value stability for already-released versions?"*

If you're looking for professional support, please refer to the Support page on our website . The vendors listed there may offer service level agreements (SLA), including extended support periods and additional services.

CHAPTER 57

Preview Versions

CloudNativePG candidate releases are pre-release versions made available for testing before the community issues a new generally available (GA) release. These versions are feature-frozen, meaning no new features are added, and are intended for public testing prior to the final release.

> **Important**
>
> CloudNativePG release candidates are not intended for use in production systems.

57.1 Purpose of Release Candidates

Release candidates are provided to the community for extensive testing before the official release. While a release candidate aims to be identical to the initial release of a new minor version of CloudNativePG, additional changes may be implemented before the GA release.

57.2 Community Involvement

The stability of each CloudNativePG minor release significantly depends on the community's efforts to test the upcoming version with their workloads and tools. Identifying bugs and regressions through user testing is crucial in determining when we can finalize the release.

57.3 Usage Advisory

The CloudNativePG Community strongly advises against using preview versions of CloudNativePG in production environments or active development projects. Although CloudNativePG undergoes extensive automated and manual testing, beta releases may contain serious bugs. Features in preview versions may change in ways that are not backwards compatible and could be removed entirely.

57.4 Current Preview Version

There are currently no preview versions available.

<!– The current preview version is **1.28.0-rc1**.

For more information on the current preview version and how to test, please view the links below:

- Announcement
- Documentation
- ->

CHAPTER 58

Release Note

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

- *Release notes for CloudNativePG 1.27*
- *Release notes for CloudNativePG 1.26*
- *Release notes for CloudNativePG 1.25*

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

Older releases:

- Release notes for CloudNativePG 1.24
- Release notes for CloudNativePG 1.23
- Release notes for CloudNativePG 1.22
- Release notes for CloudNativePG 1.21
- Release notes for CloudNativePG 1.20
- Release notes for CloudNativePG 1.19
- Release notes for CloudNativePG 1.18
- Release notes for CloudNativePG 1.17
- Release notes for CloudNativePG 1.16
- Release notes for CloudNativePG 1.15

We also keep record of all the *release notes from 1.0.0 to 1.14.0 of Cloud Native PostgreSQL by EDB* , the predecessor of CloudNativePG.

58.1 Release notes for CloudNativePG 1.27

History of user-visible changes in the 1.27 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

58.1.1 Version 1.27.0

Release date: Aug 12, 2025

Important changes:

- The default behavior of the *Liveness Probe* has been updated. An *isolated primary is now forcibly shut down*

within the configured `livenessProbeTimeout` (default: 30 seconds).

Features:

- **Dynamic loading of PostgreSQL extensions**: Introduced the `.spec.postgresql.extensions` stanza for mounting PostgreSQL extensions, packaged as OCI-compliant container images, as read-only and immutable volumes inside instance pods. This allows dynamic extension management without rebuilding base images. (#7991).
- **Logical decoding slot synchronization in HA clusters**: Added the `synchronizeLogicalDecoding` field under `spec.replicationSlots.highAvailability` to enable automatic synchronization of logical decoding slots across high-availability clusters, ensuring logical replication subscribers continue seamlessly after a publisher failover (#7931).
- **Primary Isolation Check**: Promoted to stable the liveness pinger experimental feature introduced in 1.26, adding the `.spec.probes.liveness.isolationCheck` section to enable primary isolation checks in the liveness probe by default. This improves the detection and handling of primary connectivity issues in Kubernetes environments (#7845).

Enhancements:

- Introduced an opt-in experimental feature that enables quorum-based failover to improve safety and data durability during failover events. This feature, also called failover quorum, can be activated via the `alpha.cnpg.io/failoverQuorum` annotation. (#7572).
- Added support for user maps for predefined users such as `streaming_replica` , allowing the use of self-managed client certificates with different Common Names in environments with strict policies or shared CAs, while still enabling replicas to join clusters using the `streaming_replica` role (#7725).
- Added a new `PhaseFailurePlugin` phase in the `Cluster` status to improve observability of plugin-related failures (#7988).
- Made the `Backup.spec` field immutable after creation, ensuring consistency and predictability in backup operations (#7904).
- Added `fqdn-uri` and `fqdn-jdbc-uri` fields in the user secret to simplify the retrieval of fully qualified domain name-based connection strings (#7852).
- CNPG-I:
  - Added `Postgres` interface support to the CNPG-I operator, continuing the transition toward a plugin-based architecture (#7179).
  - Added `metrics` capabilities to the CNPG-I instance webserver, enabling metrics exposure directly from the instance for better observability (#8033).

Fixes

- Unblocked rollouts when migrating to the `barman-cloud` plugin using the `switchover` strategy. Former primary Pods now restart correctly after WAL archiving fails due to missing plugin support. (#8236)

Supported versions

- Kubernetes 1.33, 1.32, and 1.31
- PostgreSQL 17, 16, 15, 14, and 13 - PostgreSQL 17.5 is the default image - PostgreSQL 13 support ends on November 12, 2025

58.2 Release notes for CloudNativePG 1.26

History of user-visible changes in the 1.26 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

58.2.1 Version 1.26.1

Release date: Jul 25, 2025

- In memory of DJ Walker-Morgan .*

Important Changes

CloudNativePG is moving toward a plugin-based, backup and recovery agnostic architecture (initiated in 1.26.0 with Barman Cloud). As part of this transition, the following fields in the `.status` section of the `Cluster` resource are now deprecated:

- `firstRecoverabilityPoint`
- `firstRecoverabilityPointByMethod`
- `lastSuccessfulBackup`
- `lastSuccessfulBackupByMethod`
- `lastFailedBackup`

Additionally, the following Prometheus metrics are deprecated:

- `cnpg_collector_first_recoverability_point`
- `cnpg_collector_last_available_backup_timestamp`
- `cnpg_collector_last_failed_backup_timestamp`

These fields and metrics will no longer update when using plugin-based backups (e.g., Barman Cloud via CNPG-I). They remain functional for users still using in-core Barman Cloud and volume snapshot backups.

> **Note:** We, as maintainers, are sorry for any inconvenience caused by not > highlighting this change during the 1.26.0 release. As we transition to a > plugin-based backup and recovery architecture, we encourage you to **test > your chosen plugin thoroughly in a staging environment before moving to > production** to ensure your workflows and observability integration continue > to meet your needs. Thank you for your understanding and for working with us > as CloudNativePG evolves to provide a more modular and robust experience.

Changes

- Removed `386` and ARM (v5/v6/v7) architectures from the `cnpg` plugin build matrix, reducing the number of published binaries (#7648).

Enhancements

- Improved validation of `shared_buffers` by correctly considering `HugePages` settings, ensuring accurate memory configuration checks (#7864).
- Set `oom_score_adj` for PostgreSQL worker processes to improve prioritization during out-of-memory situations (#7891).
- Added `fqdn-uri` and `fqdn-jdbc-uri` fields in user secrets to simplify application connection string management and align with DNS-based connection best practices (#7852).
- Added the `systemID` field and related condition in the `Cluster` status to track the PostgreSQL system identifier. (#7717).

Fixes

- Added a mutex in the connection pooler to protect concurrent access to the connections map, improving stability in high-concurrency environments (#7804).
- Fixed replica cluster instance ordering by correctly detecting the designated primary, improving replica cluster stability and switchover operations (#8108).
- Added support for reconciling `VolumeAttributesClass` for PVCs, enhancing storage compatibility and lifecycle management (#7885).
- Made the internal webserver routines non-blocking to improve responsiveness under load (#8071).
- Fixed an issue where the `ensureReplicationClientLeafCertificate` error did not display the correct `secretName` in the not found message (#8086).
- Prevented invalid `ALTER SUBSCRIPTION` statements by updating only PostgreSQL supported parameters; unsupported options like `copy_data` are ignored to avoid reconciliation failures (7844).
- Fixed an issue where the `bootstrap-controller` in the connection pooler did not apply `resources` settings correctly (#7922).
- Ensured online backups fail cleanly if the `targetPod` becomes unhealthy during backup, preventing partial or misleading backups (#7944).
- Ensured the Backup resource status is set properly after a failure, improving observability and scripting reliability (#7898).
- Improved liveness probe handling to avoid unnecessary timeouts when they are not required (#7902).

58.2.2 Version 1.26.0

Release date: May 23, 2025

Important Changes

- **CloudNativePG is now officially a CNCF project**: CloudNativePG has been accepted into the Cloud Native Computing Foundation (CNCF), marking a significant milestone in its evolution. As part of this transition, the project is now governed under **CloudNativePG, a Series of LF Projects, LLC**, ensuring long-term sustainability and community-driven innovation. (#7203)

- **Deprecation of Native Barman Cloud Support**: Native support for Barman Cloud backups and recovery is now deprecated and will be fully removed in CloudNativePG version 1.28.0. Although still available in the current release, users are strongly encouraged to begin migrating their existing clusters to the new Barman Cloud Plugin to ensure a smooth and seamless transition. The plugin should also be used for all new deployments. This change marks the first step toward making CloudNativePG a backup-agnostic solution, a goal that will be fully realized when volume snapshot support is also moved to a plugin-based architecture. (#6876)
- **End of Support for Barman 3.4 and Earlier**: CloudNativePG no longer supports Barman versions 3.4 and earlier, including the capability detection framework. Users running older operand versions (from before April 2023) must update their operand before upgrading the operator to avoid compatibility issues. (#7220)
- **Hibernation Command Changes**: The `hibernate on` and `hibernate off` commands in the `cnpg` plugin for `kubectl` now serve as shortcuts for declarative hibernation. The previous imperative approach has been removed in favor of this method. Additionally, the `hibernate status` command has been removed, as its functionality is now covered by the standard `status` command. **Warning:** Do not upgrade to version 1.26 of both the plugin and the operator unless you are prepared to migrate to the declarative hibernation method. (#7155)

Features

- **Declarative Offline In-Place Major Upgrades of PostgreSQL**: Introduced support for offline in-place major upgrades when a new operand container image with a higher PostgreSQL major version is applied to a cluster. During the upgrade, all cluster pods are shut down to ensure data consistency. A new job is created to validate upgrade conditions, run `pg_upgrade` , and set up new directories for `PGDATA` , WAL files, and tablespaces as needed. Once the upgrade is complete, replicas are re-created. Failed upgrades can be rolled back declaratively. (#6664)
- **Improved Startup and Readiness Probes for Replicas**: Enhanced support for Kubernetes startup and readiness probes in PostgreSQL instances, providing greater control over replicas based on the streaming lag. (#6623)
- **Declarative management of extensions and schemas**: Introduced the `extensions` and `schemas` stanzas in the Database resource to declaratively create, modify, and drop PostgreSQL extensions and schemas within a database. (#7062)

Enhancements

- Introduced an opt-in experimental feature to enhance the liveness probe with network isolation detection for primary instances. This feature can be activated via the `alpha.cnpg.io/livenessPinger` annotation (#7466).
- Introduced the `STANDBY_TCP_USER_TIMEOUT` operator configuration setting, allowing users to specify the `tcp_user_timeout` parameter on all standby instances managed by the operator. (#7036)
- Introduced the `DRAIN_TAINTS` operator configuration option, enabling users to customize which node taints indicate a node is being drained. This replaces the previous fixed behavior of only recognizing `node.kubernetes.io/unschedulable` as a drain signal. (#6928)
- Added a new field in the `status` of the `Cluster` resource to track the latest known Pod IP (#7546).
- Added the `pg_extensions` metric, providing information about installed PostgreSQL extensions and their latest available versions. (#7195)
- Added the `KUBERNETES_CLUSTER_DOMAIN` configuration option to the operator, allowing users to specify the domain suffix for fully qualified domain names (FQDNs) generated within the Kubernetes cluster. If not set, it defaults to `cluster.local` . (#6989)
- Implemented the `cnpg.io/validation` annotation, enabling users to disable the validation webhook on CloudNativePG-managed resources. Use with caution, as this allows unrestricted changes. (#7196)

- Added support for patching PostgreSQL instance pods using the `cnpg.io/podPatch` annotation with a JSON Patch. This may introduce discrepancies between the operator's expectations and Kubernetes behavior, so it should be used with caution. (#6323)
- Added support for collecting `pg_stat_wal` metrics in PostgreSQL 18. (#7005)
- Removed the `ENABLE_AZURE_PVC_UPDATES` configuration, as it is no longer required to resize Azure volumes correctly. The Azure CSI driver includes the necessary fix as of version 1.11.0 . (#7297)
- The `.spec.backup.barmanObjectStore` and `.spec.backup.retentionPolicy` fields are now deprecated in favor of the external Barman Cloud Plugin, and a warning is now emitted by the admission webhook when these fields are used in the `Cluster` specification (#7500).
- Added support for LZ4, XZ, and Zstandard compression methods when archiving WAL files via Barman Cloud (*deprecated*). (#7151)
- CloudNativePG Interface (CNPG-I):
  - A plugin can now trigger instance rollouts by implementing the `EVALUATE` verb, ensuring that plugin-induced changes are properly reconciled. (#7126)
  - Introduced support for WAL recovery via CNPG-I plugins during snapshot restore. (#7284)

Security

- Set `imagePullPolicy` to `Always` for the operator deployment to ensure that images are always pulled from the registry, reducing the risk of using outdated or potentially unsafe local images. (#7250)

Fixes

- Fixed native replication slot synchronization and logical replication failover for PostgreSQL 17 by appending the `dbname` parameter to `primary_conninfo` in replica configurations (#7298).
- Fixed a regression in WAL restore operations that prevented fallback to the in-tree `barmanObjectStore` configuration defined in the `externalCluster` source when a plugin failed to locate a WAL file (#7507).
- Improved backup efficiency by introducing a fail-fast mechanism in WAL archiving, allowing quicker detection of unexpected primary demotion and avoiding unnecessary retries (#7483).
- Fixed an off-by-one error in parallel WAL archiving that could cause one extra worker process to be spawned beyond the requested number (#7389).
- Resolved a race condition that caused the operator to perform two switchovers when updating the PostgreSQL configuration. (#6991)
- Corrected the `PodMonitor` configuration by adjusting the `matchLabels` scope for the targeted pooler and cluster pods. Previously, the `matchLabels` were too broad, inadvertently inheriting labels from the cluster and leading to data collection from unintended targets. (#7063)
- Added a webhook warning for clusters with a missing unit (e.g., MB, GB) in the `shared_buffers` configuration. This will become an error in future releases. Users should update their configurations to include explicit units (e.g., `512MB` instead of `512`). (#7160)
- Treated timeout errors during volume snapshot creation as retryable to prevent unnecessary backup failures. (#7010)
- Moved the defaulting logic for `.spec.postgresql.synchronous.dataDurability` from the CRD to the webhook to avoid UI issues with OLM. (#7600)
- CloudNativePG Interface (CNPG-I):
  - Implemented automatic reloading of TLS certificates for plugins when they change. (#7029)

  - Ensured the operator properly closes the plugin connection when performing a backup using the plugin. (#7095, #7096)
  - Fixed an issue that prevented WALs from being archived on a former primary node when using a plugin. (#6964)
  - Improved performance and resilience of CNPG-I by removing timeouts for local plugin operations, avoiding failures during longer backup or WAL archiving executions (#7496).

- `cnpg` plugin:
  - Increased the buffer size in the `logs pretty` command to better handle larger log output (#7281).
  - Ensured the `plugin-name` parameter is required for plugin-based backups and disallowed for non-plugin backup methods (#7506).
  - Ensured that the primary Pod is recreated during an imperative restart when `primaryUpdateMethod` is set to `restart` , aligning its definition with the replicas. (#7122)

Changes

- Updated the default PostgreSQL version to 17.5 for new cluster definitions. (#7556)
- Updated the default PgBouncer version to **1.24.1** for new `Pooler` deployments (#7399).

Supported versions

- Kubernetes 1.33, 1.32, 1.31, and 1.30
- PostgreSQL 17, 16, 15, 14, and 13 - PostgreSQL 17.5 is the default image - PostgreSQL 13 support ends on November 12, 2025

58.3 Release notes for CloudNativePG 1.25

History of user-visible changes in the 1.25 minor release of CloudNativePG.

For a complete list of changes, please refer to the commits

on the release branch in GitHub.

58.3.1 Version 1.25.3

Release date: Jul 25, 2025

- In memory of DJ Walker-Morgan .*

Changes

- Removed `386` and ARM (v5/v6/v7) architectures from the `cnpg` plugin build matrix, reducing the number of published binaries (#7648).

Enhancements

- Improved validation of `shared_buffers` by correctly considering `HugePages` settings, ensuring accurate memory configuration checks (#7864).
- Set `oom_score_adj` for PostgreSQL worker processes to improve prioritization during out-of-memory situations (#7891).
- Added `fqdn-uri` and `fqdn-jdbc-uri` fields in user secrets to simplify application connection string management and align with DNS-based connection best practices (#7852).

- Added the `systemID` field and related condition in the `Cluster` status to track the PostgreSQL system identifier. (#7717).

Fixes

- Added a mutex in the connection pooler to protect concurrent access to the connections map, improving stability in high-concurrency environments (#7804).
- Fixed replica cluster instance ordering by correctly detecting the designated primary, improving replica cluster stability and switchover operations (#8108).
- Added support for reconciling `VolumeAttributesClass` for PVCs, enhancing storage compatibility and lifecycle management (#7885).
- Made the internal webserver routines non-blocking to improve responsiveness under load (#8071).
- Fixed an issue where the `ensureReplicationClientLeafCertificate` error did not display the correct `secretName` in the not found message (#8086).
- Prevented invalid `ALTER SUBSCRIPTION` statements by updating only PostgreSQL supported parameters; unsupported options like `copy_data` are ignored to avoid reconciliation failures (7844).
- Fixed an issue where the `bootstrap-controller` in the connection pooler did not apply `resources` settings correctly (#7922).
- Ensured online backups fail cleanly if the `targetPod` becomes unhealthy during backup, preventing partial or misleading backups (#7944).
- Ensured the Backup resource status is set properly after a failure, improving observability and scripting reliability (#7898).

58.3.2 Version 1.25.2

Release date: May 23, 2025

Important Changes

- **CloudNativePG is now officially a CNCF project**: CloudNativePG has been accepted into the Cloud Native Computing Foundation (CNCF), marking a significant milestone in its evolution. As part of this transition, the project is now governed under **CloudNativePG, a Series of LF Projects, LLC**, ensuring long-term sustainability and community-driven innovation. (#7203)

Enhancements

- Added the `KUBERNETES_CLUSTER_DOMAIN` configuration option to the operator, allowing users to specify the domain suffix for fully qualified domain names (FQDNs) generated within the Kubernetes cluster. If not set, it defaults to `cluster.local` . (#6989)
- Implemented the `cnpg.io/validation` annotation, enabling users to disable the validation webhook on CloudNativePG-managed resources. Use with caution, as this allows unrestricted changes. (#7196)
- Added support for collecting `pg_stat_wal` metrics in PostgreSQL 18. (#7005)
- Added support for LZ4, XZ, and Zstandard compression methods when archiving WAL files via Barman Cloud (*deprecated*). (#7151)
- CloudNativePG Interface (CNPG-I):
- A plugin can now trigger instance rollouts by implementing the `EVALUATE` verb, ensuring that plugin-induced changes are properly reconciled. (#7126)
- Introduced support for WAL recovery via CNPG-I plugins during snapshot restore. (#7284)

Security

- Set `imagePullPolicy` to `Always` for the operator deployment to ensure that images are always pulled from the registry, reducing the risk of using outdated or potentially unsafe local images. (#7250)

Fixes

- Fixed native replication slot synchronization and logical replication failover for PostgreSQL 17 by appending the `dbname` parameter to `primary_conninfo` in replica configurations (#7298).
- Fixed a regression in WAL restore operations that prevented fallback to the in-tree `barmanObjectStore` configuration defined in the `externalCluster` source when a plugin failed to locate a WAL file (#7507).
- Improved backup efficiency by introducing a fail-fast mechanism in WAL archiving, allowing quicker detection of unexpected primary demotion and avoiding unnecessary retries (#7483).
- Fixed an off-by-one error in parallel WAL archiving that could cause one extra worker process to be spawned beyond the requested number (#7389).
- Resolved a race condition that caused the operator to perform two switchovers when updating the PostgreSQL configuration. (#6991)
- Corrected the `PodMonitor` configuration by adjusting the `matchLabels` scope for the targeted pooler and cluster pods. Previously, the `matchLabels` were too broad, inadvertently inheriting labels from the cluster and leading to data collection from unintended targets. (#7063)
- Added a webhook warning for clusters with a missing unit (e.g., MB, GB) in the `shared_buffers` configuration. This will become an error in future releases. Users should update their configurations to include explicit units (e.g., `512MB` instead of `512`). (#7160)
- Treated timeout errors during volume snapshot creation as retryable to prevent unnecessary backup failures. (#7010)
- Moved the defaulting logic for `.spec.postgresql.synchronous.dataDurability` from the CRD to the webhook to avoid UI issues with OLM. (#7600)
- CloudNativePG Interface (CNPG-I):
- Implemented automatic reloading of TLS certificates for plugins when they change. (#7029)
- Ensured the operator properly closes the plugin connection when performing a backup using the plugin. (#7095, #7096)
- Improved performance and resilience of CNPG-I by removing timeouts for local plugin operations, avoiding failures during longer backup or WAL archiving executions (#7496).
- `cnpg` plugin:
- Increased the buffer size in the `logs pretty` command to better handle larger log output (#7281).
- Ensured the `plugin-name` parameter is required for plugin-based backups and disallowed for non-plugin backup methods (#7506).
- Ensured that the primary Pod is recreated during an imperative restart when `primaryUpdateMethod` is set to `restart` , aligning its definition with the replicas. (#7122)

Changes

- Updated the default PostgreSQL version to 17.5 for new cluster definitions. (#7556)
- Updated the default PgBouncer version to **1.24.1** for new `Pooler` deployments (#7399).

58.3.3 Version 1.25.1

Release Date: February 28, 2025

Enhancements

- Introduced a startup probe for the operator to enhance reliability and prevent premature liveness probe failures during initialization. (#7008)
- Added support for using the `-r` service with the Pooler. (#6868)
- Introduced an optional `--ttl` flag for the `pgbench` plugin, enabling automatic deletion of completed jobs after a user-defined duration. (#6701)
- Marked known error messages from the Azure CSI Driver for volume snapshots as retryable, improving resilience. (#6906)
- Updated the default PostgreSQL version to 17.4 for new cluster definitions. (#6960)

Security

- The operator image build process has been enhanced to strengthen security and transparency. Images are now signed with `cosign` , and OCI attestations are generated, incorporating the Software Bill of Materials (SBOM) and provenance data. Additionally, OCI annotations have been added to improve traceability and ensure the integrity of the images.

Bug Fixes

- Fixed inconsistent behavior in default probe knob values when `.spec.probes` is defined, ensuring users can override all settings, including `failureThreshold` . If unspecified in the startup probe, `failureThreshold` is now correctly derived from `.spec.startupDelay / periodSeconds` (default: `10` , now overridable). The same logic applies to liveness probes via `.spec.livenessProbeTimeout` . (#6656)
- Managed service ports now take precedence over default operator-defined ports. (#6474)
- Fixed an issue where WAL metrics were unavailable after an instance restart until a configuration change was applied. (#6816)
- Fixed an issue in monolithic database import where role import was skipped if no roles were specified. (#6646)
- Added support for new metrics introduced in PgBouncer 1.24. (#6630)
- Resolved an issue where `Database` , `Publication` , and `Subscription` CRDs became stuck in `cluster resource has been deleted, skipping reconciliation` after cluster rehydration. This patch forces `status.observedGeneration` to zero, ensuring proper reconciliation. (#6607)
- Improved handling of replication-sensitive parameter reductions by ensuring timely reconciliation after primary server restarts. (#6440)
- Introduced a new `isWALArchiver` flag in the CNPG-I plugin configuration, allowing users to designate a plugin as a WAL archiver. This enables seamless migration from in-tree Barman Cloud support to the plugin while maintaining WAL archive consistency. (#6593)
- Ensured `override.conf` is consistently included in `postgresql.conf` during replica cluster bootstrapping, preventing replication failures due to missing configuration settings. (#6808)
- Ensured `override.conf` is correctly initialized before invoking `pg_rewind` to prevent failures during primary role changes. (#6670)
- Enhanced webhook responses to return both warnings and errors when applicable, improving diagnostic accuracy. (#6579)
- Ensured the operator version is correctly reconciled. (#6496)

- Improved PostgreSQL version detection by using a more precise check of the data directory. (#6659)
- Volume Snapshot Backups:
  - Fixed an issue where unused backup connections were not properly cleaned up. (#6882)
  - Ensured the instance manager closes stale PostgreSQL connections left by failed volume snapshot backups. (#6879)
  - Prevented the operator from starting a new volume snapshot backup while another is already in progress. (#6890)
- `cnpg` plugin:
  - Restored functionality of the `promote` plugin command. (#6476)
  - Enhanced `kubectl cnpg report --logs <cluster>` to collect logs from all containers, including sidecars. (#6636)
  - Ensured `pgbench` jobs can run when a `Cluster` uses an `ImageCatalog` . (#6868)

Technical Enhancements

- Added support for Kubernetes `client-gen` , enabling automated generation of Go clients for all CloudNativePG CRDs. (#6695)

58.3.4 Version 1.25.0

Release Date: December 23, 2024

Features

- **Declarative Database Management**: Introduce the `Database` Custom Resource Definition (CRD), enabling users to create and manage PostgreSQL databases declaratively within a cluster. (#5325)
- **Logical Replication Management**: Add `Publication` and `Subscription` CRDs for declarative management of PostgreSQL logical replication. These simplify replication setup and facilitate online migrations to CloudNativePG. (#5329)
- **Experimental Support for CNPG-I**: Introducing CNPG-I (CloudNativePG Interface), a standardized framework designed to extend CloudNativePG functionality through third-party plugins and foster the growth of the CNPG ecosystem. The Barman Cloud Plugin serves as a live example, illustrating how plugins can be developed to enhance backup and recovery workflows. Although CNPG-I support is currently experimental, it offers a powerful approach to extending CloudNativePG without modifying the operator's core code—akin to PostgreSQL extensions. We welcome community feedback and contributions to shape this exciting new capability.

Enhancements

- Add the `dataDurability` option to the `.spec.postgresql.synchronous` stanza, allowing users to choose between `required` (default) or `preferred` durability in synchronous replication. (#5878)
- Enable customization of startup, liveness, and readiness probes through the `.spec.probes` stanza. (#6266)
- Support additional `pg_dump` and `pg_restore` options to enhance database import flexibility. (#6214)
- Add support for `maxConcurrentReconciles` in the CloudNativePG controller and set the default to 10, improving the operator's ability to efficiently manage larger deployments out of the box. (#5678)
- Add the `cnpg.io/userType` label to secrets generated for predefined users, specifically `superuser` and `app` . (#4392)

- Improved validation for the `spec.schedule` field in ScheduledBackups, raising warnings for potential misconfigurations. (#5396)
- `cnpg` plugin:
  - Enhance the `backup` command to support plugins. (#6045)
  - Honor the `User-Agent` header in HTTP requests with the API server. (#6153)

Bug Fixes

- Ensure the former primary flushes its WAL file queue to the archive before re-synchronizing as a replica, reducing recovery times and enhancing data consistency during failovers. (#6141)
- Clean the WAL volume along with the `PGDATA` volume during bootstrap. (#6265)
- Update the operator to set the cluster phase to `Unrecoverable` when all previously generated `PersistentVolumeClaims` are missing. (#6170)
- Fix the parsing of the `synchronous_standby_names` GUC when `.spec.postgresql.synchronous.method` is set to `first` . (#5955)
- Resolved a potential race condition when patching certain conditions in CRD statuses, improving reliability in concurrent updates. (#6328)
- Correct role changes to apply at the transaction level instead of the database context. (#6064)
- Remove the `primary_slot_name` definition from the `override.conf` file on the primary to ensure it is always empty. (#6219)
- Configure libpq environment variables, including `PGHOST` , in PgBouncer pods to enable seamless access to the `pgbouncer` virtual database using `psql` from within the container. (#6247)
- Remove unnecessary updates to the Cluster status when verifying changes in the image catalog. (#6277)
- Prevent panic during recovery from an external server without proper backup configuration. (#6300)
- Resolved a key collision issue in structured logs, where the name field was inconsistently used to log two distinct values. (#6324)
- Ensure proper quoting of the inRoles field in SQL statements to prevent syntax errors in generated SQL during role management. (#6346)
- `cnpg` plugin:
  - Ensure the `kubectl` context is properly passed in the `psql` command. (#6257)
  - Avoid displaying physical backups block when empty with `status` command. (#5998)

Supported Versions

- **Kubernetes**: 1.32, 1.31, 1.30, and 1.29
- **PostgreSQL**: 17, 16, 15, 14, and 13 - Default image: PostgreSQL 17.2 - Officially dropped support for PostgreSQL 12 - PostgreSQL 13 support ends on November 12, 2025

58.4 Release notes for 1.14.0 and earlier

The first public release of CloudNativePG is version 1.15.0. Before that, the product was entirely owned by EDB and distributed under the name of Cloud Native PostgreSQL .

The list of changes in this page is only for informative purposes, to demonstrate the history of the product on top of commits. **None of the versions listed here exists for CloudNativePG.**

58.4.1 Version 1.14.0

Release date: 25 March 2022

Features:

- Natively support Google Cloud Storage for backup and recovery, by taking advantage of the features introduced in Barman Cloud 2.19
- Improved observability of backups through the introduction of the `LastBackupSucceeded` condition for the `Cluster` object
- Support update of Hot Standby sensitive parameters: `max_connections` , `max_prepared_transactions` , `max_locks_per_transaction` , `max_wal_senders` , `max_worker_processes`
- Add the `Online upgrade in progress` phase in the `Cluster` object to show when an online upgrade of the operator is in progress
- Ability to inherit an AWS IAM Role as an alternative way to provide credentials for the S3 object storage
- Support for Opaque secrets for Pooler's authQuerySecret and certificates
- Updated default PostgreSQL version to 14.2
- Add a new command to `kubectl cnp` plugin named `maintenance` to set maintenance window to cluster(s) in one or all namespaces across the Kubernetes cluster

Container Images:

- Latest PostgreSQL containers include Barman Cloud 2.19

Security Enhancements:

- Stronger RBAC enforcement for namespaced operator installations with Operator Lifecycle Manager, including OpenShift. OpenShift users are recommended to update to this version.

Fixes:

- Allow the instance manager to retry an interrupted `pg_rewind` by preserving a copy of the original `pg_control` file
- Clean up stale PID files before running `pg_rewind`
- Force sorting by key in `primary_conninfo` to avoid random restarts with PostgreSQL versions prior to 13
- Preserve `ServiceAccount` changes (e.g., labels, annotations) upon reconciliation
- Disable enforcement of the imagePullPolicy default value
- Improve initDB validation for WAL segment size
- Properly handle the `targetLSN` option when recovering a cluster with the LSN specified
- Fix custom TLS certificates validation by allowing a certificates chain both in the server and CA certificates

58.4.2 Version 1.13.0

Release date: 17 February 2022

Features:

- Support for Snappy compression. Snappy is a fast compression option for backups that increase the speed of uploads to the object store using a lower compression ratio
- Support for tagging files uploaded to the Barman object store. This feature requires Barman 2.18 in the operand image. of backups after Cluster deletion

- Extension of the status of a Cluster with `status.conditions` . The condition `ContinuousArchiving` indicates that the Cluster has started to archive WAL files
- Improve the status command of the `cnp` plugin for `kubectl` with additional information: add a `Cluster Summary` section showing the status of the Cluster and a `Certificates Status` section including the status of the certificates used in the Cluster along with the time left to expire
- Support the new `barman-cloud-check-wal-archive` command to detect a non-empty backup destination when creating a new cluster
- Add support for using a `Secret` to add default monitoring queries through `MONITORING_QUERIES_SECRET` configuration variable.
- Allow the user to restrict container's permissions using AppArmor (on Kubernetes clusters deployed with AppArmor support)
- Add Windows platform support to `cnp` plugin for `kubectl` , now the plugin is available on Windows x86 and ARM
- Drop support for Kubernetes 1.18 and deprecated API versions

Container Images:

- PostgreSQL containers include Barman 2.18

Security Fix:

- Add coherence check of username field inside owner and superuser secrets; previously, a malicious user could have used the secrets to change the password of any PostgreSQL user

Fixes:

- Fix a memory leak in code fetching status from Postgres pods
- Disable PostgreSQL self-restart after a crash. The instance controller handles the lifecycle of the PostgreSQL instance
- Prevent modification of `.spec.postgresUID` and `.spec.postgresGID` fields in validation webhook. Changing these fields after Cluster creation makes PostgreSQL unable to start
- Reduce the log verbosity from the backup and WAL archiving handling code
- Correct a bug resulting in a Cluster being marked as `Healthy` when not initialized yet
- Allows standby servers in clusters with a very high WAL production rate to switch to streaming once they are aligned
- Fix a race condition during the startup of a PostgreSQL pod that could seldom lead to a crash
- Fix a race condition that could lead to a failure initializing the first PVC in a Cluster
- Remove an extra restart of a just demoted primary Pod before joining the Cluster as a replica
- Correctly handle replication-sensitive PostgreSQL configuration parameters when recovering from a backup
- Fix missing validation of PostgreSQL configurations during Cluster creation

58.4.3 Version 1.12.0

Release date: 11 January 2022

Features:

- Add Kubernetes 1.23 to the list of supported Kubernetes distributions and remove end-to-end tests for 1.17, which ended support by the Kubernetes project in Dec 2020

- Improve the responsiveness of pod status checks in case of network issues by adding a connection timeout of 2 seconds and a communication timeout of 30 seconds. This change sets a limit on the time the operator waits for a pod to report its status before declaring it as failed, enhancing the robustness and predictability of a failover operation
- Introduce the `.spec.inheritedMetadata` field to the Cluster allowing the user to specify labels and annotations that will apply to all objects generated by the Cluster
- Reduce the number of queries executed when calculating the status of an instance
- Add a readiness probe for PgBouncer
- Add support for custom Certification Authority of the endpoint of Barman's backup object store when using Azure protocol

Fixes:

- During a failover, wait to select a new primary until all the WAL streaming connections are closed. The operator now sets by default `wal_sender_timeout` and `wal_receiver_timeout` to 5 seconds to make sure standby nodes will quickly notice if the primary has network issues
- Change WAL archiving strategy in replica clusters to fix rolling updates by setting "archive_mode" to "always" for any PostgreSQL instance in a replica cluster. We then restrict the upload of the WAL only from the current and target designated primary. A WAL may be uploaded twice during switchovers, which is not an issue
- Fix support for custom Certification Authority of the endpoint of Barman's backup object store in replica clusters source
- Use a fixed name for default monitoring config map in the cluster namespace
- If the defaulting webhook is not working for any reason, the operator now updates the Cluster with the defaults also during the reconciliation cycle
- Fix the comparison of resource requests and limits to fix a rare issue leading to an update of all the pods on every reconciliation cycle
- Improve log messages from webhooks to also include the object namespace
- Stop logging a "default" message at the start of every reconciliation loop
- Stop logging a PodMonitor deletion on every reconciliation cycle if `enablePodMonitor` is false
- Do not complain about possible architecture mismatch if a pod is not reachable

58.4.4 Version 1.11.0

Release date: 15 December 2021

Features:

- **Parallel WAL archiving and restore:** allow the database to keep up with WAL generation on high write systems by introducing the `backupObjectStore.maxParallel` option to set the maximum number of parallel jobs to be executed during both WAL archiving (by PostgreSQL's `archive_command`) and WAL restore (by `restore_command`). Using parallel restore option can allow newly promoted Standbys to get to a ready state faster by fetching needed WAL files to replay in parallel rather than sequentially
- **Default set of metrics for monitoring:** a new `ConfigMap` called `default-monitoring` is automatically deployed in the same namespace of the operator and, by default, added to any existing Postgres cluster. Such behavior can be changed globally by setting the `MONITORING_QUERIES_CONFIGMAP` parameter in the operator's configuration, or at cluster level through the `.spec.monitoring.disableDefaultQueries` option (by default set to `false`)

- Introduce the `enablePodMonitor` option in the monitoring section of a cluster to automatically manage a `PodMonitor` resource and seamlessly integrate with Prometheus
- Improve the PostgreSQL shutdown procedure by trying to execute a smart shutdown for the first half of the desired `stopDelay` time, and a fast shutdown for the remaining half, before the pod is killed by Kubernetes
- Add the `switchoverDelay` option to control the time given to the former primary to shut down gracefully and archive all the WAL files before promoting the new primary (by default, CloudNativePG waits indefinitely to privilege data durability)
- Handle changes to resource requests and limits for a PostgreSQL `Cluster` by issuing a rolling update
- Improve the `status` command of the `cnp` plugin for `kubectl` with additional information: streaming replication status, total size of the database, role of an instance in the cluster
- Enhance support of workloads with many parallel workers by enabling configuration of the `dynamic_shared_memory_type` and `shared_memory_type` parameters for PostgreSQL's management of shared memory
- Propagate labels and annotations defined at cluster level to the associated resources, including pods (deletions are not supported)
- Automatically remove pods that have been evicted by the Kubelet
- Manage automated resizing of persistent volumes in Azure through the `ENABLE_AZURE_PVC_UPDATES` operator configuration option, by issuing a rolling update of the cluster if needed (disabled by default)
- Introduce the`cnpg.io/reconciliationLoop` annotation that, when set to `disabled` on a given Postgres cluster, prevents the reconciliation loop from running
- Introduce the `postInitApplicationSQL` option as part of the `initdb` bootstrap method to specify a list of SQL queries to be executed on the main application database as a superuser immediately after the cluster has been created

Fixes:

- Liveness probe now correctly handles the startup process of a PostgreSQL server. This fixes an issue reported by a few customers and affects a restarted standby server that needs to recover WAL files to reach a consistent state, but it was not able to do it before the timeout of liveness probe would kick in, leaving the pods in `CrashLoopBackOff` status.
- Liveness probe now correctly handles the case of a former primary that needs to use `pg_rewind` to re-align with the current primary after a timeline diversion. This fixes the pod of the new standby from repeatedly being killed by Kubernetes.
- Reduce client-side throttling from Postgres pods (e.g. `Waited for 1.182388649s due to client-side throttling, not priority and fairness, request: GET`)
- Disable Public Key Infrastructure (PKI) initialization on OpenShift and OLM installations, by using the provided one
- When changing configuration parameters that require a restart, always leave the primary as last
- Mark a PVC to be ready only after a job has been completed successfully, preventing a race condition in PVC initialization
- Use the correct public key when renewing the expired webhook TLS secret.
- Fix an overflow when parsing an LSN
- Remove stale PID files at startup
- Let the `Pooler` resource inherit the `imagePullSecret` defined in the operator, if exists

58.4.5 Version 1.10.0

Release date: 11 November 2021

Features:

- **Connection Pooling with PgBouncer**: introduce the `Pooler` resource and controller to automatically manage a PgBouncer deployment to be used as a connection pooler for a local PostgreSQL `Cluster`. The feature includes TLS client/server connections, password authentication, High Availability, pod templates support, configuration of key PgBouncer parameters, `PAUSE`/`RESUME`, logging in JSON format, Prometheus exporter for stats, pools, and lists
- **Backup Retention Policies**: support definition of recovery window retention policies for backups (e.g. '30d' to ensure a recovery window of 30 days)
- **In-Place updates of the operator**: introduce an in-place online update of the instance manager, which removes the need to perform a rolling update of the entire cluster following an update of the operator. By default this option is disabled (please refer to the *documentation for more detailed information*)
- Limit the list of options that can be customized in the `initdb` bootstrap method to `dataChecksums`, `encoding`, `localeCollate`, `localeCType`, `walSegmentSize`. This makes the `options` array obsolete and planned to be removed in the v2 API
- Introduce the `postInitTemplateSQL` option as part of the `initdb` bootstrap method to specify a list of SQL queries to be executed on the `template1` database as a superuser immediately after the cluster has been created. This feature allows you to include default objects in all application databases created in the cluster
- New default metrics added to the instance Prometheus exporter: Postgres version, cluster name, and first point of recoverability according to the backup catalog
- Retry taking a backup after a failure
- Build awareness about Barman Cloud capabilities in order to prevent the operator from invoking recently introduced features (such as retention policies, or Azure Blob Container storage) that are not present in operand images that are not frequently updated
- Integrate the output of the `status` command of the `cnp` plugin with information about the backup
- Introduce a new annotation that reports the status of a PVC (being initialized or ready)
- Set the cluster name in the `k8s.enterprisedb.io/cluster` label for every object generated in a `Cluster`, including `Backup` objects
- Drop support for deprecated API version `postgresql.cnpg.io/v1alpha1` on the `Cluster`, `Backup`, and `ScheduledBackup` kinds
- Set default operand image to PostgreSQL 14.2

Security:

- Set allowPrivilegeEscalation to `false` for the operator containers securityContext

Fixes:

- Disable primary PodDisruptionBudget during maintenance in single-instance clusters
- Use the correct certificate certification authority (CA) during recovery operations
- Prevent Postgres connection leaking when checking WAL archiving status before taking a backup
- Let WAL archive/restore sleep for 100ms following transient errors that would flood logs otherwise

58.4.6 Version 1.9.2

Release date: 15 October 2021

Features:

- Enhance JSON log with two new loggers: `wal-archive` for PostgreSQL's `archive_command` , and `wal-restore` for `restore_command` in a standby

Fixes:

- Enable WAL archiving during the standby promotion (prevented `.history` files from being archived)
- Pass the `--cloud-provider` option to Barman Cloud tools only when using Barman 2.13 or higher to avoid errors with older operands
- Wait for the pod of the primary to be ready before triggering a backup

58.4.7 Version 1.9.1

Release date: 30 September 2021

- This release is to celebrate the launch of PostgreSQL 14

by making it the default major version when a new `Cluster` is created without defining a specific image name.*

Fixes:

- Fix issue causing `Error while getting barman endpoint CA secret` message to appear in the logs of the primary pod, which prevented the backup to work correctly
- Properly retry requesting a new backup in case of temporary communication issues with the instance manager

58.4.8 Version 1.9.0

Release date: 28 September 2021

- Version 1.9.0 is not available on OpenShift due to delays with the release process and the subsequent release of version 1.9.1.*

Features:

- Add Kubernetes 1.22 to the list of supported Kubernetes distributions, and remove 1.16
- Introduce support for the `--restore-target-wal` option in `pg_rewind` , in order to fetch WAL files from the backup archive, if necessary (available only with PostgreSQL 13+)
- Expose a default metric for the Prometheus exporter that estimates the number of pages in the `pg_catalog.pg_largeobject` table in each database
- Enhance the performance of WAL archiving and fetching, through local in-memory cache

Fixes:

- Explicitly set the `postgres` user when invoking `pg_isready` - required by restricted SCC in OpenShift
- Properly update the `FirstRecoverabilityPoint` in the status
- Set `archive_mode = always` on the designated primary if backup is requested
- Minor bug fixes

58.4.9 Version 1.8.0

Release date: 13 September 2021

Features:

- Bootstrap a new cluster via full or Point-In-Time Recovery directly from an object store defined in the external cluster section, eliminating the previous requirement to have a Backup CR defined
- Introduce the `immediate` option in scheduled backups to request a backup immediately after the first Postgres instance running, adding the capability to rewind to the very beginning of a cluster when Point-In-Time Recovery is configured
- Add the `firstRecoverabilityPoint` in the cluster status to report the oldest consistent point in time to request a recovery based on the backup object store's content
- Enhance the default Prometheus exporter for a PostgreSQL instance by exposing the following new metrics:
  1. number of WAL files and computed total size on disk
  2. number of `.ready` and `.done` files in the archive status folder
  3. flag for replica mode
  4. number of requested minimum/maximum synchronous replicas, as well as the expected and actually observed ones
- Add support for the `runonserver` option when defining custom metrics in the Prometheus exporter to limit the collection of a metric to a range of PostgreSQL versions
- Natively support Azure Blob Storage for backup and recovery, by taking advantage of the feature introduced in Barman 2.13 for Barman Cloud
- Rely on `pg_isready` for the liveness probe
- Support RFC3339 format for timestamp specification in recovery target times
- Introduce `.spec.imagePullPolicy` to control the pull policy of image containers for all pods and jobs created for a cluster
- Add support for OpenShift 4.8, which replaces OpenShift 4.5
- Support PostgreSQL 14 (beta)
- Enhance the replica cluster feature with cross-cluster replication from an object store defined in an external cluster section, without requiring a streaming connection (experimental)
- Introduce `logLevel` option to the cluster's spec to specify one of the following levels: error, info, debug or trace

Security Enhancements:

- Introduce `.spec.enableSuperuserAccess` to enable/disable network access with the `postgres` user through password authentication

Fixes:

- Properly inform users when a cluster enters an unrecoverable state and requires human intervention

58.4.10 Version 1.7.1

Release date: 11 August 2021

Features:

- Prefer self-healing over configuration with regards to synchronous replication, empowering the operator to temporarily override `minSyncReplicas` and `maxSyncReplicas` settings in case the cluster is not able to meet the requirements during self-healing operations
- Introduce the `postInitSQL` option as part of the `initdb` bootstrap method to specify a list of SQL queries to be executed as a superuser immediately after the cluster has been created

Fixes:

- Allow the operator to failover when the primary is not ready (bug introduced in 1.7.0)
- Execute administrative queries using the `LOCAL` synchronous commit level
- Correctly parse multi-line log entries in PGAudit

58.4.11 Version 1.7.0

Release date: 28 July 2021

Features:

- Add native support to PGAudit with a new type of `logger` called `pgaudit` directly available in the JSON output
- Enhance monitoring and observability capabilities through:
  - Native support for the `pg_stat_statements` and `auto_explain` extensions
  - The `target_databases` option in the Prometheus exporter to run a user-defined metric query on one or more databases (including auto-discovery of databases through shell-like pattern matching)
  - Exposure of the `manual_switchover_required` metric to promptly report whether a cluster with `primaryUpdateStrategy` set to `supervised` requires a manual switchover
- Transparently handle `shared_preload_libraries` for `pg_audit` , `auto_explain` and `pg_stat_statements`
  - Automatic configuration of `shared_preload_libraries` for PostgreSQL when `pg_stat_statements` , `pgaudit` or `auto_explain` options are added to the `postgresql` parameters section
- Support the `cnpg.io/reload` label to finely control the automated reload of config maps and secrets, including those used for custom monitoring/alerting metrics in the Prometheus exporter or to store certificates
- Add the `reload` command to the `cnp` plugin for `kubectl` to trigger a reconciliation loop on the instances
- Improve control of pod affinity and anti-affinity configurations through `additionalPodAffinity` and `additionalPodAntiAffinity`
- Introduce a separate `PodDisruptionBudget` for primary instances, by requiring at least a primary instance to run at any time

Security Enhancements:

- Add the `.spec.certificates.clientCASecret` and `.spec.certificates.replicationTLSSecret` options to define custom client Certification Authority and certificate for the PostgreSQL server, to be used to authenticate client certificates and secure communication between PostgreSQL nodes
- Add the `.spec.backup.barmanObjectStore.endpointCA` option to define the custom Certification Authority bundle of the endpoint of Barman's backup object store

Fixes:

- Correctly parse histograms in the Prometheus exporter
- Reconcile services created by the operator for a cluster

58.4.12 Version 1.6.0

Release date: 12 July 2021

Features:

- Replica mode (**EXPERIMENTAL**): allow a cluster to be created as a replica of a source cluster. A replica cluster has a *designated primary* and any number of standbys.
- Add the `.spec.postgresql.promotionTimeout` parameter to specify the maximum amount of seconds to wait when promoting an instance to primary, defaulting to 40000000 seconds.
- Add the `.spec.affinity.podAntiAffinityType` parameter. It can be set to `preferred` (default), resulting in `preferredDuringSchedulingIgnoredDuringExecution` being used, or to `required`, resulting in `requiredDuringSchedulingIgnoredDuringExecution`.

Changes:

- Fixed a race condition when deleting a PVC and a pod which prevented the operator from creating a new pod.
- Fixed a race condition preventing the manager from detecting the need for a PostgreSQL restart on a configuration change.
- Fixed a panic in `kubectl-cnp` on clusters without annotations.
- Lowered the level of some log messages to `debug`.
- E2E tests for server CA and TLS injection.

58.4.13 Version 1.5.1

Release date: 17 June 2021

Change:

- Fix a bug with CRD validation preventing auto-update with Operator Deployments on Red Hat OpenShift
- Allow passing operator's configuration using a Secret.

58.4.14 Version 1.5.0

Release date: 11 June 2021

Features:

- Introduce the `pg_basebackup` bootstrap method to create a new PostgreSQL cluster as a copy of an existing PostgreSQL instance of the same major version, even outside Kubernetes
- Add support for Kubernetes' tolerations in the `Affinity` section of the `Cluster` resource, allowing users to distribute PostgreSQL instances on Kubernetes nodes with the required taint
- Enable specification of a digest to an image name, through the `<image>:<tag>@sha256:<digestValue>` format, for more deterministic and repeatable deployments

Security Enhancements:

- Customize TLS certificates to authenticate the PostgreSQL server by defining secrets for the server certificate and the related Certification Authority that signed it
- Raise the `sslmode` for the WAL receiver process of internal and automatically managed streaming replicas from `require` to `verify-ca`

Changes:

- Enhance the `promote` subcommand of the `cnp` plugin for `kubectl` to accept just the node number rather than the whole name of the pod
- Adopt DNS-1035 validation scheme for cluster names (from which service names are inherited)
- Enforce streaming replication connection when cloning a standby instance or when bootstrapping using the `pg_basebackup` method
- Integrate the `Backup` resource with `beginWal` , `endWal` , `beginLSN` , `endLSN` , `startedAt` and `stoppedAt` regarding the physical base backup
- Documentation improvements:
  - Provide a list of ports exposed by the operator and the operand container
  - Introduce the `cnp-bench` helm charts and guidelines for benchmarking the storage and PostgreSQL for database workloads
- E2E tests enhancements:
  - Test Kubernetes 1.21
  - Add test for High Availability of the operator
  - Add test for node draining
- Minor bug fixes, including:
  - Timeout to pg_ctl start during recovery operations too short
  - Operator not watching over direct events on PVCs
  - Fix handling of `immediateCheckpoint` and `jobs` parameter in `barmanObjectStore` backups
  - Empty logs when recovering from a backup

58.4.15 Version 1.4.0

Release date: 18 May 2021

Features:

- Standard output logging of PostgreSQL error messages in JSON format
- Provide a basic set of PostgreSQL metrics for the Prometheus exporter
- Add the `restart` command to the `cnp` plugin for `kubectl` to restart the pods of a given PostgreSQL cluster in a rollout fashion

Security Enhancements:

- Set `readOnlyRootFilesystem` security context for pods

Changes:

- **IMPORTANT:** If you have previously deployed the CloudNativePG operator using the YAML manifest, you must delete the existing operator deployment before installing the new version. This is required to avoid conflicts with other Kubernetes API's due to a change in labels and label selectors being directly managed by the operator. Please refer to the CloudNativePG documentation for additional detail on upgrading to 1.4.0
- Fix the labels that are automatically defined by the operator, renaming them from `control-plane: controller-manager` to `app.kubernetes.io/name: cloudnative-pg`
- Assign the `metrics` name to the TCP port for the Prometheus exporter
- Set `cnp_metrics_exporter` as the `application_name` to the metrics exporter connection in PostgreSQL

- When available, use the application database for monitoring queries of the Prometheus exporter instead of the `postgres` database
- Documentation improvements:
  - Customization of monitoring queries
  - Operator upgrade instructions
- E2E tests enhancements
- Minor bug fixes, including:
  - Avoid using `-R` when calling `pg_basebackup`
  - Remove stack trace from error log when getting the status

58.4.16 Version 1.3.0

Release date: 23 Apr 2021

Features:

- Inheritance of labels and annotations
- Set resource limits for every container

Security Enhancements:

- Support for restricted security context constraint on Red Hat OpenShift to limit pod execution to a namespace allocated UID and SELinux context
- Pod security contexts explicitly defined by the operator to run as non-root, non-privileged and without privilege escalation

Changes:

- Prometheus exporter endpoint listening on port 9187 (port 8000 is now reserved to instance coordination with API server)
- Documentation improvements
- E2E tests enhancements, including GKE environment
- Minor bug fixes

58.4.17 Version 1.2.1

Release date: 6 Apr 2021

- ScheduledBackup are no longer owners of the Backups, meaning that backups are not removed when ScheduledBackup objects are deleted
- Update on ubi8-minimal image to solve RHSA-2021:1024 (Security Advisory: Important)

58.4.18 Version 1.2.0

Release date: 31 Mar 2021

- Introduce experimental support for custom monitoring queries as ConfigMap and Secret objects using a compatible syntax with `postgres_exporter` for Prometheus
- Support Operator Lifecycle Manager (OLM) deployments, with the subsequent presence on OperatorHub.io

- Enhance container security by applying guidelines from the US Department of Defense (DoD)'s Defense Information Systems Agency (DISA) and the Center for Internet Security (CIS) and verifying them directly in the pipeline with Dockle
- Improve E2E tests on AKS
- Minor bug fixes

58.4.19 Version 1.1.0

Release date: 3 Mar 2021

- Add `kubectl cnp status` to pretty-print the status of a cluster, including JSON and YAML output
- Add `kubectl cnp certificate` to enable TLS authentication for client applications
- Add the `-ro` service to route connections to the available hot standby replicas only, enabling offload of read-only queries from the cluster's primary instance
- Rollback scaling down a cluster to a value lower than `maxSyncReplicas`
- Request a checkpoint before demoting a former primary
- Send `SIGINT` signal (fast shutdown) to PostgreSQL process on `SIGTERM`
- Minor bug fixes

58.4.20 Version 1.0.0

Release date: 4 Feb 2021

The first major stable release of CloudNativePG implements `Cluster` , `Backup` and `ScheduledBackup` in the API group `postgresql.cnpg.io/v1` . It uses these resources to create and manage PostgreSQL clusters inside Kubernetes with the following main capabilities:

- Direct integration with Kubernetes API server for High Availability, without requiring an external tool
- Self-Healing capability, through:
  - failover of the primary instance by promoting the most aligned replica
  - automated recreation of a replica
- Planned switchover of the primary instance by promoting a selected replica
- Scale up/down capabilities
- Definition of an arbitrary number of instances (minimum 1 - one primary server)
- Definition of the *read-write* service to connect your applications to the only primary server of the cluster
- Definition of the *read* service to connect your applications to any of the instances for reading workloads
- Support for Local Persistent Volumes with PVC templates
- Reuse of Persistent Volumes storage in Pods
- Rolling updates for PostgreSQL minor versions and operator upgrades
- TLS connections and client certificate authentication
- Continuous backup to an S3 compatible object store
- Full recovery and point-in-time recovery from an S3 compatible object store backup
- Support for synchronous replicas
- Support for node affinity via `nodeSelector` property

- Standard output logging of PostgreSQL error messages