EDB PostgresÂ® AI for CloudNativePGâ„¢ Cluster

Version 1.26.0

Built at 2025-06-09T15:17:54

1 EDB Postgres® AI for CloudNativePG™ Cluster

The EDB Postgres® AI for CloudNativePG™ Cluster operator is a fork based on CloudNativePG™. It provides additional value such as compatibility with Oracle using EDB Postgres Advanced Server and additional supported platforms such as IBM Power and OpenShift. It is designed, developed, and supported by EDB and covers the full lifecycle of a highly available Postgres database clusters with a primary/standby architecture, using native streaming replication.

EDB Postgres for Kubernetes was made generally available on February 4, 2021. Earlier versions were made available to selected customers prior to the GA release.

> **Note**
>
> The operator has been renamed from Cloud Native PostgreSQL. Existing users of Cloud Native PostgreSQL will not experience any change, as the underlying components and resources have not changed.

Key features in common with CloudNativePG™

- Kubernetes API integration for high availability
  - CloudNativePG™ uses the `postgresql.cnpg.io/v1` API version
  - EDB Postgres for Kubernetes uses the `postgresql.k8s.enterprisedb.io/v1` API version

- Self-healing through failover and automated recreation of replicas
- Capacity management with scale up/down capabilities
- Planned switchovers for scheduled maintenance
- Read-only and read-write Kubernetes services definitions
- Rolling updates for Postgres minor versions and operator upgrades
- Continuous backup and point-in-time recovery
- Connection Pooling with PgBouncer
- Integrated metrics exporter out of the box
- PostgreSQL replication across multiple Kubernetes clusters
- Separate volume for WAL files

Features unique to EDB Postgres of Kubernetes

- Long Term Support
- Support on IBM Power and z/Linux through partnership with IBM
- Oracle compatibility through EDB Postgres Advanced Sever
- Transparent Data Encryption (TDE) through EDB Postgres Advanced Server
- Cold backup support with Kasten and Velero/OADP
- Generic adapter for third-party Kubernetes backup tools

You can evaluate EDB Postgres for Kubernetes for free as part of a trial subscription. You need a valid EDB subscription to use EDB Postgres for Kubernetes in production.

> **Note**
>
> Based on the Operator Capability Levels model, users can expect a **"Level V - Auto Pilot"** set of capabilities from the EDB Postgres for Kubernetes Operator.

Long Term Support

EDB is committed to declaring a Long Term Support (LTS) version of EDB Postgres for Kubernetes annually. 1.25 is the current LTS version. 1.18 was the first. Each LTS version will receive maintenance releases and be supported for an additional 12 months beyond the last community release of CloudNativePG for the same version.

For example, the 1.22 release of CloudNativePG reached End-of-Life on July 24, 2024, for the open source community. Because it was declared an LTS version of EDB Postgres for Kubernetes, it will be supported for an additional 12 months, until July 24, 2025.

In addition, customers will always have at least 6 months to move between LTS versions. For example, the 1.25 LTS version was released on December 23 2024, giving ample time to users to migrate from the 1.22 LTS ahead of the End-of-life on July 2025.

While we encourage customers to regularly upgrade to the latest version of the operator to take advantage of new features, having LTS versions allows customers desiring additional stability to stay on the same version for 12-18 months before upgrading.

Licensing

EDB Postgres for Kubernetes works with both PostgreSQL, EDB Postgres Extended and EDB Postgres Advanced server, and is available under the EDB Limited Use License.

You can evaluate EDB Postgres for Kubernetes for free as part of a trial subscription. You need a valid EDB subscription to use EDB Postgres for Kubernetes in production.

Supported releases and Kubernetes distributions

For a list of the minor releases of EDB Postgres for Kubernetes that are supported by EDB, please refer to the "Platform Compatibility" page. Here you can also find which Kubernetes distributions and versions are supported for each of them and the EOL dates.

Multiple architectures

The EDB Postgres for Kubernetes Operator container images support the multi-arch format for the following platforms: `linux/amd64`, `linux/arm64`, `linux/ppc64le`, `linux/s390x`.

> **Warning**
>
> EDB Postgres for Kubernetes requires that all nodes in a Kubernetes cluster have the same CPU architecture, thus a hybrid CPU architecture Kubernetes cluster is not supported. Additionally, EDB supports `linux/ppc64le` and `linux/s390x` architectures on OpenShift only.

Supported Postgres versions

The following versions of Postgres are currently supported by version 1.25:

- PostgreSQL: 13 - 17
- EDB Postgres Advanced: 13 - 17
- EDB Postgres Extended: 13 - 17

PostgreSQL and EDB Postgres Advanced are available on the following platforms: `linux/amd64`, `linux/ppc64le`, `linux/s390x`.
In addition, PostgreSQL is also supported on `linux/arm64`.
EDB Postgres Extended is supported only on `linux/amd64`.
EDB supports operand images for `linux/ppc64le` and `linux/s390x` architectures on OpenShift only.

About this guide

Follow the instructions in the "Quickstart" to test EDB Postgres for Kubernetes 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" section.

> **Note**
>
> Although the guide primarily addresses Kubernetes, all concepts can be extended to OpenShift as well.

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

2 EDB CloudNativePG Cluster release notes

The EDB Postgres for Kubernetes documentation describes the major version of EDB Postgres for Kubernetes, including minor releases and patches. The release notes provide information on what is new in each release. For new functionality introduced in a minor or patch release, the content also indicates the release that introduced the feature.

Version	Release date	Upstream merges
1.26.0	23 May 2025	Upstream 1.26.0
1.25.2	23 May 2025	Upstream 1.25.2
1.25.1	04 Mar 2025	Upstream 1.25.1
1.25.0	23 Dec 2025	Upstream 1.25.0
1.24.4	23 May 2025	Upstream 1.24.4
1.24.3	04 Mar 2025	Upstream 1.24.3
1.24.2	23 Dec 2024	Upstream 1.24.2
1.24.1	18 Oct 2024	Upstream 1.24.1
1.24.0	26 Aug 2024	Upstream 1.24.0
1.23.6	23 Dec 2024	Upstream 1.23.6
1.23.5	18 Oct 2024	Upstream 1.23.5
1.23.4	26 Aug 2024	Upstream 1.23.4
1.23.3	01 Aug 2024	Upstream 1.23.3
1.23.2	13 Jun 2024	Upstream 1.23.2
1.23.1	29 Apr 2024	Upstream 1.23.1
1.23.0	24 Apr 2024	Upstream 1.23.0
1.22.10	23 May 2025	None
1.22.9	04 Mar 2025	None
1.22.8	23 Dec 2024	None
1.22.7	18 Oct 2024	None
1.22.6	26 Aug 2024	Upstream 1.22.6
1.22.5	01 Aug 2024	Upstream 1.22.5
1.22.4	13 Jun 2024	Upstream 1.22.4
1.22.3	24 Apr 2024	Upstream 1.22.3
1.22.2	22 Mar 2024	Upstream 1.22.2
1.22.1	02 Feb 2024	Upstream 1.22.1
1.22.0	22 Dec 2023	Upstream 1.22.0
1.21.6	13 Jun 2024	Upstream 1.21.6
1.21.5	24 Apr 2024	Upstream 1.21.5
1.21.4	22 Mar 2024	Upstream 1.21.4
1.21.3	02 Feb 2024	Upstream 1.21.3
1.21.2	22 Dec 2023	Upstream 1.21.2
1.21.1	08 Nov 2023	Upstream 1.21.1
1.21.0	18 Oct 2023	Upstream 1.21.0
1.20.6	02 Feb 2024	Upstream 1.20.6
1.20.5	22 Dec 2023	Upstream 1.20.5
1.20.4	08 Nov 2023	Upstream 1.20.4
1.20.3	18 Oct 2023	Upstream 1.20.3
1.20.2	27 Jul 2023	Upstream 1.20.2

Version	Release date	Upstream merges
1.20.1	13 Jun 2023	Upstream 1.20.1
1.20.0	27 Apr 2023	Upstream 1.20.0
1.19.6	08 Nov 2023	Upstream 1.19.6
1.19.5	18 Oct 2023	Upstream 1.19.5
1.19.4	27 Jul 2023	Upstream 1.19.4
1.19.3	13 Jun 2023	Upstream 1.19.3
1.19.2	27 Apr 2023	Upstream 1.19.2
1.19.1	20 Mar 2023	Upstream 1.19.1
1.19.0	14 Feb 2023	Upstream 1.19.0
1.18.13	13 Jun 2024	None
1.18.12	24 Apr 2024	None
1.18.11	22 Mar 2024	None
1.18.10	02 Feb 2024	None
1.18.9	22 Dec 2023	None
1.18.8	08 Nov 2023	None
1.18.7	18 Oct 2023	None
1.18.6	27 Jul 2023	None
1.18.5	13 Jun 2023	Upstream 1.18.5
1.18.4	27 Apr 2023	Upstream 1.18.4
1.18.3	20 Mar 2023	Upstream 1.18.3
1.18.2	14 Feb 2023	Upstream 1.18.2
1.18.1	21 Dec 2022	Upstream 1.18.1
1.18.0	14 Nov 2022	Upstream 1.18.0
1.17.5	20 Mar 2023	Upstream 1.17.5
1.17.4	14 Feb 2023	Upstream 1.17.4
1.17.3	21 Dec 2022	Upstream 1.17.3
1.17.2	14 Nov 2022	Upstream 1.17.2
1.17.1	07 Oct 2022	Upstream 1.17.1
1.17.0	06 Sep 2022	Upstream 1.17.0
1.16.5	21 Dec 2022	Upstream 1.16.4
1.16.4	14 Nov 2022	Upstream 1.16.4
1.16.3	07 Oct 2022	Upstream 1.16.3
1.16.2	06 Sep 2022	Upstream 1.16.2
1.16.1	12 Aug 2022	Upstream 1.16.1
1.16.0	07 Jul 2022	Upstream 1.16.0
1.15.5	07 Oct 2022	Upstream 1.15.5
1.15.4	06 Sep 2022	Upstream 1.15.4
1.15.3	12 Aug 2022	Upstream 1.15.3
1.15.2	07 Jul 2022	Upstream 1.15.2
1.15.1	27 May 2022	Upstream 1.15.1
1.15.0	21 Apr 2022	Upstream 1.15.0
1.14.0	25 Mar 2022	NA
1.13.0	17 Feb 2022	NA
1.12.0	11 Jan 2022	NA
1.11.0	15 Dec 2021	NA

Version	Release date	Upstream merges
1.10.0	11 Nov 2021	NA
1.9.2	15 Oct 2021	NA
1.9.1	30 Sep 2021	NA
1.9.0	28 Sep 2021	NA
1.8.0	13 Sep 2021	NA
1.7.1	11 Aug 2021	NA
1.7.0	28 Jul 2021	NA
1.6.0	12 Jul 2021	NA
1.5.1	11 Jun 2021	NA
1.5.0	17 Jun 2021	NA
1.4.0	18 May 2021	NA
1.3.0	23 Apr 2021	NA
1.2.1	06 Apr 2021	NA
1.2.0	31 Mar 2021	NA
1.1.0	03 Mar 2021	NA
1.0.0	04 Feb 2021	NA
0.8.0	29 Jan 2021	NA
0.7.0	31 Dec 2020	NA
0.6.0	04 Dec 2020	NA
0.5.0	20 Nov 2020	NA
0.4.0	05 Nov 2020	NA
0.3.0	25 Sep 2020	NA
0.2.0	11 Aug 2020	NA
0.1.0	03 Apr 2020	NA
0.0.1	05 Mar 2020	NA

> **Versions prior to 1.0.0**
>
> EDB Postgres for Kubernetes was made generally available on February 4, 2021. Earlier versions were made available to selected customers prior to the GA release.

2.1 EDB CloudNativePG Cluster 1.26.0 release notes

Released: 23 May 2025

This release of EDB CloudNativePG Cluster includes the following:

Changes

Description	Addresses
Merged with community CloudNativePG 1.26.0 upstream release. See the community release notes	

2.2 EDB CloudNativePG Cluster 1.25.2 release notes

Released: 23 May 2025

This release of EDB CloudNativePG Cluster includes the following:

Changes

Description	Addresses
Merged with community CloudNativePG 1.25.2 upstream release. See the community release notes	

2.3 EDB Postgres for Kubernetes 1.25.1 release notes

Released: 04 Mar 2025

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.25.1. See the community Release Notes.

2.4 EDB Postgres for Kubernetes 1.25.0 release notes

Released: 23 Dec 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.25.0. See the community Release Notes.

2.5 EDB CloudNativePG Cluster 1.24.4 release notes

Released: 23 May 2025

This release of EDB CloudNativePG Cluster includes the following:

Changes

Description	Addresses
Merged with community CloudNativePG 1.24.4 upstream release. See the community release notes	

2.6 EDB Postgres for Kubernetes 1.24.3 release notes

Released: 04 Mar 2025

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.24.3. See the community Release Notes.

2.7 EDB Postgres for Kubernetes 1.24.2 release notes

Released: 23 Dec 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.24.2. See the community Release Notes.

2.8 EDB Postgres for Kubernetes 1.24.1 release notes

Released: 18 Oct 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.24.1. See the community Release Notes.

2.9 EDB Postgres for Kubernetes 1.24.0 release notes

Released: 26 Aug 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.24.0. See the community Release Notes.

2.10 EDB Postgres for Kubernetes 1.23.6 release notes

Released: 23 Dec 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.23.6. See the community Release Notes.

2.11 EDB Postgres for Kubernetes 1.23.5 release notes

Released: 18 Oct 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.23.5. See the community Release Notes.

2.12 EDB Postgres for Kubernetes 1.23.4 release notes

Released: 26 Aug 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.23.4. See the community Release Notes.

2.13 EDB Postgres for Kubernetes 1.23.3 release notes

Released: 01 Aug 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.23.3. See the community Release Notes.

2.14 EDB Postgres for Kubernetes 1.23.2 release notes

Released: 13 Jun 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.23.2. See the community Release Notes.

2.15 EDB Postgres for Kubernetes 1.23.1 release notes

Released: 29 Apr 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.23.1. See the community Release Notes.

2.16 EDB Postgres for Kubernetes 1.23.0 release notes

Released: 24 Apr 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.23.0. See the community Release Notes.

2.17 EDB CloudNativePG Cluster 1.22.10 release notes

Released: 23 May 2025

EDB Postgres for Kubernetes version 1.22.10 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

> **Warning**
>
> A new LTS release, version 1.25.2, is now available! Please note that support for 1.22 will officially end on July 2025.
>
> We recommend starting your upgrade planning now to transition smoothly to the 1.25 LTS release.

This release of EDB CloudNativePG Cluster includes the following:

Enhancements

Description	Addresses
Implemented the `k8s.enterprisedb.io/validation` annotation ...enabling users to disable the validation webhook on EDB Postgres for Kubernetes-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

Security Fixes

Description	Addresses
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

Changes

Description	Addresses
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

Bug Fixes

Description	Addresses
Fixed native replication slot synchronization and logical replication failover for PostgreSQL 17 ...by appending the `dbname` parameter to `primary_conninfo` in replica configurations.	#7298
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
\`cnp\` plugin: ensured that the primary Pod is recreated during an imperative restart when \`primaryUpdateMethod\` is set to \`restart\`, aligning its definition with the replicas.	#7122

2.18 EDB Postgres for Kubernetes 1.22.9 release notes

Released: 04 Mar 2025

EDB Postgres for Kubernetes version 1.22.9 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

> **Warning**
>
> A new LTS release, version 1.25.1, is now available! Please note that support for 1.22 will officially end on July 2025.
>
> We recommend starting your upgrade planning now to transition smoothly to the 1.25 LTS release.

This release of EDB Postgres for Kubernetes includes the following:

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)
- 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)
- 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)
- Improved handling of replication-sensitive parameter reductions by ensuring timely reconciliation after primary server restarts. (#6440)
- 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)
- `cnp` plugin:
  - Restored functionality of the `promote` plugin command. (#6476)
  - Enhanced `kubectl cnp 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 EDB Postgres for Kubernetes CRDs. (#6695)

2.19 EDB Postgres for Kubernetes 1.22.8 release notes

Released: 23 Dec 2024

EDB Postgres for Kubernetes version 1.22.8 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

> **Warning**
>
> A new LTS release, version 1.25.0, is now available! Please note that support for 1.22 will officially end on July 2025.
>
> We recommend starting your upgrade planning now to transition smoothly to the 1.25 LTS release.

This release of EDB Postgres for Kubernetes includes the following:

Enhancements

- Enable customization of startup, liveness, and readiness probes through the `.spec.probes` stanza. (#6266)
- 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:
  - 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)
- 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)

2.20 EDB Postgres for Kubernetes 1.22.7 release notes

Released: 18 Oct 2024

EDB Postgres for Kubernetes version 1.22.7 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

This release of EDB Postgres for Kubernetes includes the following:

Enhancements:

- Remove the use of `pg_database_size` from the status probe, as it caused high resource utilization by scanning the entire `PGDATA` directory to compute database sizes. The `kubectl status` plugin will now rely on `du` to provide detailed size information retrieval (#5689).
- Add the ability to configure the `full_page_writes` parameter in PostgreSQL. This setting defaults to `on`, in line with PostgreSQL's recommendations (#5516).
- Plugin:
  - Add the `logs pretty` command in the `cnp` plugin to read a log stream from standard input and output a human-readable format, with options to filter log entries (#5770)
  - Enhance the `status` command by allowing multiple `-v` options to increase verbosity for more detailed output (#5765).
  - Add support for specifying a custom Docker image using the `--image` flag in the `pgadmin4` plugin command, giving users control over the Docker image used for pgAdmin4 deployments (#5515).

Fixes:

- Ensure that replica PodDisruptionBudgets (PDB) are removed when scaling down to two instances, enabling easier maintenance on the node hosting the replica (#5487).
- Prioritize full rollout over inplace restarts (#5407).
- Fix an issue that could lead to double failover in cases of lost connectivity (#5788).
- Correctly set the `TMPDIR` and `PSQL_HISTORY` environment variables for pods and jobs, improving temporary file and history management (#5503).
- Plugin:
  - Resolve a race condition in the `logs cluster` command (#5775).
  - Display the `potential` sync status in the `status` plugin (#5533).
  - Fix the issue where pods deployed by the `pgadmin4` command didn't have a writable home directory (#5800).

Supported versions

- PostgreSQL 17 (PostgreSQL 17.0 is the default image)

2.21 EDB Postgres for Kubernetes 1.22.6 release notes

Released: 26 Aug 2024

This release of EDB Postgres for Kubernetes includes the following:

Features

- **Configuration of Pod Disruption Budgets (PDB)**: Introduced the `.spec.enablePDB` field to disable PDBs on the primary instance, allowing proper eviction of the pod during maintenance operations. This is particularly useful for single-instance deployments. This feature is intended to replace the node maintenance window feature.

Enhancements

- **cnp plugin updates**:
  - Enhance the install generate command by adding a --control-plane option, allowing deployment of the operator on control-plane nodes by setting node affinity and tolerations (#5271).
  - Enhance the destroy command to delete also any job related to the target instance (#5298).

Fixes

- Synchronous replication self-healing checks now exclude terminated pods, focusing only on active and functional pods (#5210).
- The instance manager will now terminate all existing operator-related replication connections following a role change in a replica cluster (#5209).
- Allow setting smartShutdownTimeout to zero, enabling immediate fast shutdown and bypassing the smart shutdown process when required (#5347).

2.22 EDB Postgres for Kubernetes 1.22.5 release notes

Released: 01 Aug 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.22.5. See the community Release Notes.

2.23 EDB Postgres for Kubernetes 1.22.4 release notes

Released: 13 Jun 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.22.4. See the community Release Notes.

2.24 EDB Postgres for Kubernetes 1.22.3 release notes

Released: 24 Apr 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.22.3. See the community Release Notes.

2.25 EDB Postgres for Kubernetes 1.22.2 release notes

Released: 22 Mar 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.22.2. See the community Release Notes.

2.26 EDB Postgres for Kubernetes 1.22.1 release notes

Released: 02 Feb 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.22.1. See the community Release Notes.

2.27 EDB Postgres for Kubernetes 1.22.0 release notes

Released: 22 Dec 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.22.0. See the community Release Notes.

2.28 EDB Postgres for Kubernetes 1.21.6 release notes

Released: 13 Jun 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.21.6. See the community Release Notes.

2.29 EDB Postgres for Kubernetes 1.21.5 release notes

Released: 23 Apr 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.21.5. See the community Release Notes.

2.30 EDB Postgres for Kubernetes 1.21.4 release notes

Released: 22 Mar 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.21.4. See the community Release Notes.

2.31 EDB Postgres for Kubernetes 1.21.3 release notes

Released: 02 Feb 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.21.3. See the community Release Notes.

2.32 EDB Postgres for Kubernetes 1.21.2 release notes

Released: 22 Dec 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.21.2. See the community Release Notes.

2.33 EDB Postgres for Kubernetes 1.21.1 release notes

Released: 08 Nov 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.21.1. See the community Release Notes.

2.34 EDB Postgres for Kubernetes 1.21.0 release notes

Released: 18 Oct 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.21.0. See the community Release Notes.

2.35 EDB Postgres for Kubernetes 1.20.6 release notes

Released: 02 Feb 2024

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.20.6. See the community Release Notes.

2.36 EDB Postgres for Kubernetes 1.20.5 release notes

Released: 22 Dec 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.20.5. See the community Release Notes.

2.37 EDB Postgres for Kubernetes 1.20.4 release notes

Released: 08 Nov 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.20.4. See the community Release Notes.

2.38 EDB Postgres for Kubernetes 1.20.3 release notes

Released: 18 Oct 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.20.3. See the community Release Notes.

2.39 EDB Postgres for Kubernetes 1.20.2 release notes

Released: 27 Jul 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.20.2. See the community Release Notes.

2.40 EDB Postgres for Kubernetes 1.20.1 release notes

Released: 13 Jun 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.20.1. See the community Release Notes.

2.41 EDB Postgres for Kubernetes 1.20.0 release notes

Released: 27 Apr 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.20.0. See the community Release Notes.

2.42 EDB Postgres for Kubernetes 1.19.6 release notes

Released: 08 Nov 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.19.6. See the community Release Notes.

2.43 EDB Postgres for Kubernetes 1.19.5 release notes

Released: 18 Oct 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.19.5. See the community Release Notes.

2.44 EDB Postgres for Kubernetes 1.19.4 release notes

Released: 27 Jul 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.19.4. See the community Release Notes.

2.45 EDB Postgres for Kubernetes 1.19.3 release notes

Released: 13 Jun 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.19.3. See the community Release Notes.

2.46 EDB Postgres for Kubernetes 1.19.2 release notes

Released: 27 Apr 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.19.2. See the community Release Notes.

2.47 EDB Postgres for Kubernetes 1.19.1 release notes

Released: 20 Mar 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.19.1. See the community Release Notes.

2.48 EDB Postgres for Kubernetes 1.19.0 release notes

Released: 14 Feb 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.19.0. See the community Release Notes.
Feature	Support for Transparent Data Encryption (TDE) with EDB Postgres Advanced Server 15. TDE encrypts, transparently to the user, any user data stored in the database system.
Feature	New external backup adaptor to provide a generic way to integrate EDB Postgres for Kubernetes in a third-party tool for backups. See External Backup Adapter for more information.

2.49 EDB Postgres for Kubernetes 1.18.13 release notes

Released: 13 Jun 2024

EDB Postgres for Kubernetes version 1.18.13 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

> **Warning**
>
> This is expected to be the last release in the 1.18.X series. Users are encouraged to update to a newer minor version soon.

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Enhancement	Enabled the configuration of the liveness probe timeout via the `.spec.livenessProbeTimeout` option (#4719)
Change	Default operand image set to PostgreSQL 16.3 (#4584)
Change	Removed all RBAC requirements on namespace objects (#4753)
Bug fix	Prevented fenced instances from entering an unnecessary loop and consuming all available CPU (#4625)
Bug fix	Resolved an issue where the instance manager on the primary would indefinitely wait for the instance to start after encountering a failure following a stop operation (#4434)
Bug fix	Fixed a panic in the backup controller that occurred when pod container statuses were missing (#4765)
Bug fix	Prevented unnecessary shutdown of the instance manager (#4670)
Bug fix	Prevented unnecessary reloads of PostgreSQL configuration when unchanged (#4531)
Bug fix	Prevented unnecessary reloads of the ident map by ensuring a consistent and unique method of writing its content (#4648)
Bug fix	Avoided conflicts during phase registration by patching the status of the resource instead of updating it (#4637)
Bug fix	Implemented a timeout when restarting PostgreSQL and lifting fencing (#4504)
Bug fix	Ensured that a replica cluster is restarted after promotion to properly set the archive mode (#4399)
Bug fix	Ensured correct parsing of the additional rows field returned when the `pgaudit.log_rows` option was enabled, preventing audit logs from being incorrectly routed to the normal log stream (#4394)

2.50 EDB Postgres for Kubernetes 1.18.12 release notes

Released: 24 Apr 2024

EDB Postgres for Kubernetes version 1.18.12 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Enhancement	Added upgrade process from 1.18.x LTS to 1.22.x LTS
Enhancement	Documentation for Kubernetes 1.29.x or above (#3729)
Enhancement	Ensure pods with no ownership are deleted during cluster restore (#4141)
Bug fix	Properly handle LSN sorting when is empty on a replica (#4283)
Bug fix	Avoids stopping reconciliation loop when there is no instance status available (#4132)
Bug fix	Waits for elected replica to be in streaming mode before a switchover (#4288)
Bug fix	Allow backup hooks to be called while using Velero backup
Bug fix	Waits for the Restic init container to be completed
Security	Updated all Go dependencies to fix any latest security issues

2.51 EDB Postgres for Kubernetes 1.18.11 release notes

Released: 22 Mar 2024

EDB Postgres for Kubernetes version 1.18.11 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Enhancement	Allow customization of the wal_level GUC in PostgreSQL (#4020).
Enhancement	Added the `cnpg.io/skipWalArchiving` annotation to disable WAL archiving when set to `enabled` (#4055).
Enhancement	Enriched the `cnpg` plugin for `kubectl` with the `publication` and `subscription` command groups to imperatively set up PostgreSQL native logical replication (#4052).
Enhancement	Allow customization of `CERTIFICATE_DURATION` and `EXPIRING_CHECK_THRESHOLD` for automated management of TLS certificates handled by the operator (#3686).
Enhancement	Now retrieves the correct architecture's binary from the corresponding catalog in the running operator image during in-place updates, enabling the operator to inject the correct binary into any Pod with a supported architecture (#3840).
Security	Now uses `Role` instead of `ClusterRole` for operator permissions in OLM, requiring fewer privileges when installed on a per-namespace basis (#3855, #3990).
Security	Now enforces fully-qualified object names in SQL queries for the PgBouncer pooler (#4080).
Bug fix	Now properly synchronizes PVC group labels with those on the pods, a critical aspect when all pods are deleted and the operator needs to decide which Pod to recreate first (#3930).
Bug fix	Now disables `wal_sender_timeout` when cloning a replica to prevent timeout errors due to slow connections (#4080).
Bug fix	Now ensures that backups are ready before initiating recovery bootstrap procedures, preventing an error condition where recovery with incomplete backups could enter an error loop (#3663).
Bug fix	Resolve a corner case in hibernation where the instance pod has been deleted, but the cluster status still has the hibernation condition set to false (#3970).
Bug fix	Correctly detect Google Cloud capabilities for Barman Cloud (#3931).
Update	Set the default operand image to PostgreSQL 16.2 (#3823).

2.52 EDB Postgres for Kubernetes 1.18.10 release notes

Released: 02 Feb 2024

EDB Postgres for Kubernetes version 1.18.10 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Enhancement	Tailor ephemeral volume storage in a Postgres cluster using a claim template through the `ephemeralVolumeSource` option (#3678).
Enhancement	Introduce the `pgadmin4` command in the `cnp` plugin for `kubectl`, providing a straightforward method to demonstrate connecting to a given database cluster and navigate its content in a local environment such as kind - for evaluation purposes only (#3701).
Enhancement	Allow customization of PostgreSQL's ident map file via the `.spec.postgresql.pg_ident` stanza, through a list of user name maps (#3534).
Bug fix	Prevent an unrecoverable issue with `pg_rewind` failing due to `postgresql.auto.conf` being read-only on clusters where the `ALTER SYSTEM` SQL command is disabled - the default (#3728).
Bug fix	Reduce the risk of disk space shortage when using the import facility of the `initdb` bootstrap method, by disabling the durability settings in the PostgreSQL instance for the duration of the import process (#3743).
Bug fix	Avoid pod restart due to erroneous resource quantity comparisons, e.g. "1 != 1000m" (#3706).
Bug fix	Properly escape reserved characters in `pgpass` connection fields (#3713).
Bug fix	Prevent systematic rollout of pods due to considering zero and nil different values in `.spec.projectedVolumeTemplate.sources` (#3647).
Bug fix	Ensure configuration coherence by pruning from `postgresql.auto.conf` any options now incorporated into `override.conf` (#3773).

2.53 EDB Postgres for Kubernetes 1.18.9 release notes

Released: 22 Dec 2023

EDB Postgres for Kubernetes version 1.18.9 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

This release of EDB Postgres for Kubernetes includes the following:

Type	Subsystem	Description
Security		By default, TLSv1.3 is now enforced on all PostgreSQL 12 or higher installations. Additionally, users can configure the `ssl_ciphers`, `ssl_min_protocol_version`, and `ssl_max_protocol_version` GUCs (#3408).
Security		Integrated Docker image scanning with Dockle to enhance security measures.
Defaults		Default operand image is now PostgreSQL 16.1 (#3270).
Enhancement		Improved reconciliation of external clusters (#3533).
Enhancement		Introduced the ability to enable/disable the `ALTER SYSTEM` command (#3535).
Enhancement		Added support for Prometheus' dynamic relabeling through the `podMonitorMetricRelabelings` and `podMonitorRelabelings` options in the `.spec.monitoring` stanza of the `Cluster` and `Pooler` resources (#3075).
Enhancement		Eliminated the use of the `PGPASSFILE` environment variable when establishing a network connection to PostgreSQL (#3522).
Enhancement		Improved `cnp report` plugin command by collecting a cluster's PVCs (#3357).
Enhancement	Connection pooler	Scaling down instances of a `Pooler` resource to 0 is now possible (#3517).
Enhancement	Connection pooler	Added the `k8s.enterprisedb.io/podRole` label with a value of 'pooler' to every pooler deployment, differentiating them from instance pods (#3396).
Bug fix		Reconciled metadata, annotations, and labels of `PodDisruptionBudget` resources (#3312 and #3434).
Bug fix		Reconciled the metadata of the managed credential secrets (#3316).
Bug fix		Disabled wal_sender_timeout when joining through pg_basebackup (#3586).
Bug fix		Secrets labeled `cnpg.io/reload=true` and used by external clusters are now reloaded when they change (#3565).
Bug fix	Connection pooler	Ensured the controller watches all secrets owned by a `Pooler` resource (#3428).
Bug fix	Connection pooler	Reconciled `RoleBinding` for `Pooler` resources (#3391).
Bug fix	Connection pooler	Reconciled `imagePullSecret` for `Pooler` resources (#3389).
Bug fix	Connection pooler	Reconciled the service of a `Pooler` and addition of the required labels (#3349).
Bug fix	Connection pooler	Extended `Pooler` labels to the deployment as well, not just the pods (#3350).

2.54 EDB Postgres for Kubernetes 1.18.8 release notes

Released: 08 Nov 2023

EDB Postgres for Kubernetes version 1.8.8 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Enhancement	Enhanced the `status` command of the `cnp` plugin for `kubectl` with progress information on active streaming base backups.
Enhancement	Allowed the configuration of `max_prepared_statements` with the pgBouncer `Pooler` resource.
Technical Enhancement	Use extended query protocol for PostgreSQL in the instance manager.
Bug fix	Suspend WAL archiving during a switchover and resume it when it is completed.
Bug fix	Ensured that the instance manager always uses `synchronous_commit = local` when managing the PostgreSQL cluster.
Bug fix	Custom certificates for streaming replication user through `.spec.certificates.replicationTLSSecret` are now working.
Bug fix	Set the `k8s.enterprisedb.io/cluster` label to the `Pooler` pods.
Change	Stopped using the `postgresql.auto.conf` file inside PGDATA to control Postgres replication settings, and replace it with a file named `override.conf`.

2.55 EDB Postgres for Kubernetes 1.18.7 release notes

Released: 18 Oct 2023

EDB Postgres for Kubernetes version 1.8.7 is an LTS release of EDB Postgres for Kubernetes; there is no corresponding upstream release of CloudNativePG.

Highlights of EDB Postgres for Kubernetes 1.8.7

- Changed the default value of `stopDelay` to 1800 seconds instead of 30 seconds
- Introduced a new parameter, called `smartShutdownTimeout`, to control the window of time reserved for the smart shutdown of Postgres to complete; the general formula to compute the overall timeout to stop Postgres is `max(stopDelay - smartShutdownTimeout, 30)`
- Changed the default value of `startDelay` to 3600, instead of 30 seconds
- Replaced the livenessProbe initial delay with a more proper Kubernetes startup probe to deal with the start of a Postgres server
- Changed the default value of `switchoverDelay` to 3600 seconds instead of 40000000 seconds

Additionally, this release of EDB Postgres for Kubernetes includes the following:

Type	Description
Security fix	Added a default `seccompProfile` to the operator deployment.
Enhancement	Introduced the `k8s.enterprisedb.io/coredumpFilter` annotation to control the content of a core dump generated in the unlikely event of a PostgreSQL crash, by default set to exclude shared memory segments from the dump.
Enhancement	Allowed configuration of ephemeral-storage limits for the shared memory and temporary data ephemeral volumes.
Enhancement	Validation of resource limits and requests through the webhook.
Enhancement	Ensure that PostgreSQL's `shared_buffers` are coherent with the pods' allocated memory resources.
Enhancement	Added `uri` and `jdbc-uri` fields in the credential secrets to facilitate developers when connecting their applications to the database.
Enhancement	Added a new phase, `Waiting for the instances to become active`, for finer control of a cluster's state waiting for the replicas to be ready.
Enhancement	Improved detection of Pod rollout conditions through the `podSpec` annotation.
Enhancement	Added primary timestamp and uptime to the kubectl plugin's `status` command.
Technical enhancement	Replaced `k8s-api-docgen` with `gen-crd-api-reference-docs` to automatically build the API reference documentation.
Bug fix	Ensure that the primary instance is always recreated first by prioritizing ready PVCs with a primary role.
Bug fix	Honor the `k8s.enterprisedb.io/skipEmptyWalArchiveCheck` annotation during recovery to bypass the check for an empty WAL archive.
Bug fix	prevent a cluster from being stuck when the PostgreSQL server is down but the pod is up on the primary.
Bug fix	Avoid treating the designated primary in a replica cluster as a regular HA replica when replication slots are enabled.
Bug fix	Reconcile services every time the selectors change or when labels/annotations need to be changed.
Bug fix	Default to `app` for both the owner and database during recovery bootstrap.
Bug fix	Avoid write-read concurrency on cached cluster.
Bug fix	Remove empty items, make them unique and sort in the `ResourceName` sections of the generated roles.
Bug fix	Ensure that the `ContinuousArchiving` condition is properly set to 'failed' in case of errors.
Bug fix	Reconcile PodMonitor `labels` and `annotations`.
Bug fix	Fixed backup failure due to missing RBAC `resourceNames` on the `Role` object.
Observability	Added TCP port label to default `pg_stat_replication` metric.
Observability	Fixed the `pg_wal_stat` default metric for Prometheus.
Observability	Improved the `pg_replication` default metric for Prometheus

Type	Description
Observability	Used `alertInstanceLabelFilter` instead of `alertName` in the provided Grafana dashboard
Observability	Enforce `standard_conforming_strings` in metric collection.
Change	Set the default operand image to PostgreSQL 16.0.
Change	Fencing now uses PostgreSQL's fast shutdown instead of smart shutdown to halt an instance.
Change	Rename webhooks from kb.io to k8s.enterprisedb.io group.
Change	Added the `k8s.enterprisedb.io/instanceRole` label and deprecated the existing `role` label.

2.56 EDB Postgres for Kubernetes 1.18.6 release notes

Released: 27 Jul 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Enhancement	Added a metric and status field to monitor node usage by an EDB Postgres for Kubernetes cluster.
Enhancement	Added troubleshooting instructions relating to hugepages to the documentation.
Enhancement	Extended the FAQs page in the documentation.
Enhancement	Added a check at the start of the restore process to ensure it can proceed; give improved error diagnostics if it cannot.
Bug fix	Ensured the logic of setting the recovery target matches that of Postgres.
Bug fix	Prevented taking over service accounts not owned by the cluster by setting ownerMetadata only during service account creation.
Bug fix	Prevented a possible crash of the instance manager during the configuration reload.
Bug fix	Prevented the LastFailedArchiveTime alert from triggering if a new backup has been successful after the failed ones.
Security fix	Updated all project dependencies to the latest versions

2.57 EDB Postgres for Kubernetes 1.18.5 release notes

Released: 13 Jun 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.18.5. See the community Release Notes.

2.58 EDB Postgres for Kubernetes 1.18.4 release notes

Released: 27 Apr 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.18.4. See the community Release Notes.

2.59 EDB Postgres for Kubernetes 1.18.3 release notes

Released: 20 Mar 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.18.3. See the community Release Notes.

2.60 EDB Postgres for Kubernetes 1.18.2 release notes

Released: 14 Feb 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.18.2. See the community Release Notes.
Feature	Support for Transparent Data Encryption (TDE) with EDB Postgtres Advanced Server 15. TDE encrypts, transparently to the user, any user data stored in the database system.
Feature	New external backup adaptor to provide a generic way to integrate EDB Postgres for Kubernetes in a third-party tool for backups. See External Backup Adapter for more information.

2.61 EDB Postgres for Kubernetes 1.18.1 release notes

Released: 21 Dec 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.18.1. See the community Release Notes.

2.62 EDB Postgres for Kubernetes 1.18.0 release notes

Released: 14 Nov 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.18. See the community Release Notes.

2.63 EDB Postgres for Kubernetes 1.17.5 release notes

Released: 20 Mar 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.17.5. See the community Release Notes.

2.64 EDB Postgres for Kubernetes 1.17.4 release notes

Released: 14 Feb 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.17.4. See the community Release Notes.

2.65 EDB Postgres for Kubernetes 1.17.3 release notes

Released: 21 Dec 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.17.2. See the community Release Notes.

2.66 EDB Postgres for Kubernetes 1.17.2 release notes

Released: 14 Nov 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.17.2. See the community Release Notes.

2.67 EDB Postgres for Kubernetes 1.17.1 release notes

Released: 07 Oct 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.17.1. See the community Release Notes.
Enhancement	Introduces `leaseDuration` and `renewDeadline` parameters in the controller manager to enhance configuration of the leader election in operator deployments.
Enhancement	Improves the mechanism that checks that the backup object store is empty before archiving a WAL file for the first time. A new file called `.check-empty-wal-archive` is placed in the `PGDATA` immediately after the cluster is bootstrapped. It is removed after the first WAL file is successfully archived.
Security	Explicitly sets permissions of the instance manager binary that is copied in the `distroless/static:nonroot` container image, by using the `nonroot:nonroot` user.
Bug fix	Drops any active connection on a standby after it is promoted to primary.
Bug fix	Honors `MAPPEDMETRIC` and `DURATION` metric types conversion in the native Prometheus exporter.
Bug fix	Ensures that timestamps that are specified with microsecond precision using the PostgreSQL format are correctly parsed.

2.68 EDB Postgres for Kubernetes 1.17.0 release notes

Released: 06 Sep 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.17. See the community Release Notes.

2.69 EDB Postgres for Kubernetes 1.16.5 release notes

Released: 21 Dec 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.16.5. See the community Release Notes.

2.70 EDB Postgres for Kubernetes 1.16.4 release notes

Released: 14 Nov 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.16.4. See the community Release Notes.

2.71 EDB Postgres for Kubernetes 1.16.3 release notes

Released: 07 Oct 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.16.3. See the community Release Notes.
Enhancement	Introduces `leaseDuration` and `renewDeadline` parameters in the controller manager to enhance configuration of the leader election in operator deployments.
Enhancement	Improves the mechanism that checks that the backup object store is empty before archiving a WAL file for the first time. A new file called `.check-empty-wal-archive` is placed in the `PGDATA` immediately after the cluster is bootstrapped. It is removed after the first WAL file is successfully archived.
Security	Explicitly sets permissions of the instance manager binary that is copied in the `distroless/static:nonroot` container image, by using the `nonroot:nonroot` user.
Bug fix	Drops any active connection on a standby after it is promoted to primary.
Bug fix	Honors `MAPPEDMETRIC` and `DURATION` metric types conversion in the native Prometheus exporter.

2.72 EDB Postgres for Kubernetes 1.16.2 release notes

Released: 06 Sep 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.16.2. See the community Release Notes.

2.73 EDB Postgres for Kubernetes 1.16.1 release notes

Released: 12 Aug 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.16. See the community Release Notes.

Important changes to our upgrade policy

We are adopting a new policy to support the last two minor versions of the product, in line with the CloudNativePG community.

As a result, we are introducing the following head versions in the new OLM channels in OpenShift to manage the update streams for EDB Postgres for Kubernetes:

- `fast` : the latest available patch release for the latest available minor release
- `stable-v1.16` : the latest available patch release of the 1.16 minor release
- `stable-v1.15` : the latest available patch release of the 1.15 minor release

Prior to this release, the only channel that we were supporting was the `stable` channel. This channel is now obsolete. However, for backward compatibility it is currently set as an alias of the `stable-v1.15` channel. It will be removed once version 1.15 reaches End of Life.

Important information about upgrading to a 1.16.x operator version on Openshift

We have made a change to the way conditions are represented in the status of the operator in version 1.16.0 and onward. This change could cause an operator upgrade to hang on Openshift if one of the old conditions are set during the upgrade process because of the way the Operator Lifecycle Manager checks new CRDs against existing CRs.

Prior to installing 1.16.x on Openshift, if you are upgrading from a 1.15.x (or earlier) version of the operator, we recommend uninstalling the existing version of the operator, then deleting all of the old conditions out of the statuses of all existing EDB Postgres for Kubernetes clusters. This will have no effect on the operability of your existing EDB Postgres for Kubernetes clusters.

To remove the existing conditions run:

```
while IFS=' ' read  NS CLUSTER;
do
  kubectl -n ${NS} patch --type='json' ${CLUSTER} --subresource=status -p='[{"op": "remove", "path":
"/status/conditions"}]';
done < <(kubectl get cluster -A --no-headers=true -o jsonpath='{range .items[*]}{.metadata.namespace}{"
cluster/"}{.metadata.name}{"\n"}{end}')
```

> **Important**
>
> The kubectl command must be version 1.24 or higher. If you get the output `The request is invalid` it means that the target cluster didn't have any condition on it.

This command will remove all of the conditions from all of the EDB Postgres for Kubernetes clusters in your Openshift cluster. Once the command completes, you can safely install version 1.16.x.

If you have already tried to upgrade to 1.16.x from 1.15.x (or earlier) and the install of 1.16.x shows as "Pending" and the earlier version shows as "Cannot update", uninstall both versions of the operator and run the command that removes the statuses.

2.74 EDB Postgres for Kubernetes 1.16.0 release notes

Released: 07 Jul 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.16. See the community Release Notes.
Enhancement	New “fast” channel available for OpenShift, providing update paths to the latest available patch release in the latest available minor release of EDB Postgres for Kubernetes. See the following section for more information.
Enhancement	New “stable-v1.16” channel available for OpenShift, providing update paths to the latest available patch release in the 1.16 release branch of EDB Postgres for Kubernetes. See the following section for more information.

Important changes to our upgrade policy

We are adopting a new policy to support the last two minor versions of the product, in line with the CloudNativePG community.

As a result, we are introducing the following head versions in the new OLM channels in OpenShift to manage the update streams for EDB Postgres for Kubernetes:

- `fast` : the latest available patch release for the latest available minor release
- `stable-v1.16` : the latest available patch release of the 1.16 minor release
- `stable-v1.15` : the latest available patch release of the 1.15 minor release

Prior to this release, the only channel that we were supporting was the `stable` channel. This channel is now obsolete. However, for backward compatibility it is currently set as an alias of the `stable-v1.15` channel. It will be removed once version 1.15 reaches End of Life.

Important information about upgrading to a 1.16.x operator version on Openshift

We have made a change to the way conditions are represented in the status of the operator in version 1.16.0 and onward. This change could cause an operator upgrade to hang on Openshift if one of the old conditions are set during the upgrade process because of the way the Operator Lifecycle Manager checks new CRDs against existing CRs.

Prior to installing 1.16.x on Openshift, if you are upgrading from a 1.15.x (or earlier) version of the operator, we recommend uninstalling the existing version of the operator, then deleting all of the old conditions out of the statuses of all existing EDB Postgres for Kubernetes clusters. This will have no effect on the operability of your existing EDB Postgres for Kubernetes clusters.

To remove the existing conditions run:

```
while IFS=' ' read  NS CLUSTER;
do
  kubectl -n ${NS} patch --type='json' ${CLUSTER} --subresource=status -p='[{"op": "remove", "path":
"/status/conditions"}]';
done < <(kubectl get cluster -A --no-headers=true -o jsonpath='{range .items[*]}{.metadata.namespace}{"
cluster/"}{.metadata.name}{"\n"}{end}')
```

> **Important**
>
> The kubectl command must be version 1.24 or higher. If you get the output `The request is invalid` it means that the target cluster didn't have any condition on it.

This command will remove all of the conditions from all of the EDB Postgres for Kubernetes clusters in your Openshift cluster. Once the command completes, you can safely install version 1.16.x.

If you have already tried to upgrade to 1.16.x from 1.15.x (or earlier) and the install of 1.16.x shows as "Pending" and the earlier version shows as "Cannot update", uninstall both versions of the operator and run the command that removes the statuses.

2.75 EDB Postgres for Kubernetes 1.15.5 release notes

Released: 07 Oct 2022

> **Warning**
>
> Version 1.15 has reached End-of-Life (EOL). Version 1.15.5 is the last release for the 1.15 minor version.

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.15.5. See the community Release Notes.
Enhancement	Introduces `leaseDuration` and `renewDeadline` parameters in the controller manager to enhance configuration of the leader election in operator deployments.
Enhancement	Improves the mechanism that checks that the backup object store is empty before archiving a WAL file for the first time. A new file called `.check-empty-wal-archive` is placed in the `PGDATA` immediately after the cluster is bootstrapped. It is removed after the first WAL file is successfully archived.
Security	Explicitly sets permissions of the instance manager binary that is copied in the `distroless/static:nonroot` container image, by using the `nonroot:nonroot` user.
Bug fix	Makes the cluster's conditions compatible with `metav1.Conditions` struct.
Bug fix	Drops any active connection on a standby after it is promoted to primary.
Bug fix	Honors `MAPPEDMETRIC` and `DURATION` metric types conversion in the native Prometheus exporter.

2.76 EDB Postgres for Kubernetes 1.15.4 release notes

Released: 06 Sep 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.15.4. See the community Release Notes.

2.77 EDB Postgres for Kubernetes 1.15.3 release notes

Released: 12 Aug 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.15.3. See the community Release Notes.

Important changes to our upgrade policy

We are adopting a new policy to support the last two minor versions of the product beginning with 1.16, in line with the CloudNativePG community.

As a result, we are introducing the following head versions in the new OLM channels in OpenShift to manage the update streams for EDB Postgres for Kubernetes:

- `fast` : the latest available patch release for the latest available minor release
- `stable-v1.16` : the latest available patch release of the 1.16 minor release
- `stable-v1.15` : the latest available patch release of the 1.15 minor release

Prior to the release of 1.16, the only channel that we were supporting was the `stable` channel. This channel is now obsolete. However, for backward compatibility it is currently set as an alias of the `stable-v1.15` channel and it will be removed once version 1.15 goes End of Life.

2.78 EDB Postgres for Kubernetes 1.15.2 release notes

Released: 07 Jul 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.15.2. See the community Release Notes.
Enhancement	New “stable-v1.15” channel available for OpenShift, providing update paths to the latest available patch release in the 1.15 release branch of EDB Postgres for Kubernetes. The “stable” channel is a synonym of “stable-v1.15”. See the following section for more information.

Important changes to our upgrade policy

We are adopting a new policy to support the last two minor versions of the product beginning with 1.16, in line with the CloudNativePG community.

As a result, we are introducing the following head versions in the new OLM channels in OpenShift to manage the update streams for EDB Postgres for Kubernetes:

- `fast` : the latest available patch release for the latest available minor release
- `stable-v1.16` : the latest available patch release of the 1.16 minor release
- `stable-v1.15` : the latest available patch release of the 1.15 minor release

Prior to the release of 1.16, the only channel that we were supporting was the `stable` channel. This channel is now obsolete. However, for backward compatibility it is currently set as an alias of the `stable-v1.15` channel and it will be removed once version 1.15 goes End of Life.

2.79 EDB Postgres for Kubernetes 1.15.1 release notes

Released: 27 May 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.15.1. See the community Release Notes.

2.80 EDB Postgres for Kubernetes 1.15.0 release notes

Released: 21 Apr 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Upstream merge	Merged with community CloudNativePG 1.15.0. See the community Release Notes.

2.81 EDB Postgres for Kubernetes 1.14.0 release notes

Released: 25 Mar 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	Natively support Google Cloud Storage for backup and recovery, by taking advantage of the features introduced in Barman Cloud 2.19.
Feature	Improved observability of backups through the introduction of the LastBackupSucceeded condition for the Cluster object.
Feature	Support update of Hot Standby sensitive parameters: max_connections, max_prepared_transactions, max_locks_per_transaction, max_wal_senders, max_worker_processes.
Feature	Add the Online upgrade in progress phase in the Cluster object to show when an online upgrade of the operator is in progress.
Feature	Ability to inherit an AWS IAM Role as an alternative way to provide credentials for the S3 object storage.
Feature	Support for Opaque secrets for Pooler’s authQuerySecret and certificates.
Feature	Updated default PostgreSQL version to 14.2.
Feature	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 and EDB Postgres Advanced Server containers include Barman Cloud 2.19.
Security fix	Stronger RBAC enforcement for namespaced operator installations with Operator Lifecycle Manager, including OpenShift. OpenShift users are recommended to update to this version.
Bug fix	Allow the instance manager to retry an interrupted pg_rewind by preserving a copy of the original pg_control file.
Bug fix	Clean up stale PID files before running pg_rewind.
Bug fix	Force sorting by key in primary_conninfo to avoid random restarts with PostgreSQL versions prior to 13.
Bug fix	Preserve ServiceAccount changes (e.g., labels, annotations) upon reconciliation.
Bug fix	Disable enforcement of the imagePullPolicy default value.
Bug fix	Improve initDB validation for WAL segment size.
Bug fix	Properly handle the targetLSN option when recovering a cluster with the LSN specified.
Bug fix	Fix custom TLS certificates validation by allowing a certificates chain both in the server and CA certificates.

2.82 EDB Postgres for Kubernetes 1.13.0 release notes

Released: 17 Feb 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	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.
Feature	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.
Feature	Extension of the status of a Cluster with status.conditions. The condition ContinuousArchiving indicates that the Cluster has started to archive WAL files.
Feature	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.
Feature	Support the new barman-cloud-check-wal-archive command to detect a non-empty backup destination when creating a new cluster.
Feature	Add support for using a Secret to add default monitoring queries through MONITORING_QUERIES_SECRET configuration variable.
Feature	Allow the user to restrict container’s permissions using AppArmor (on Kubernetes clusters deployed with AppArmor support).
Feature	Add Windows platform support to cnp plugin for kubectl, now the plugin is available on Windows x86 and ARM.
Feature	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.
Bug fix	Fix a memory leak in code fetching status from Postgres pods.
Bug fix	Disable PostgreSQL self-restart after a crash. The instance controller handles the lifecycle of the PostgreSQL instance.
Bug fix	Prevent modification of spec.postgresUID and spec.postgresGID fields in validation webhook. Changing these fields after Cluster creation makes PostgreSQL unable to start.
Bug fix	Reduce the log verbosity from the backup and WAL archiving handling code.
Bug fix	Correct a bug resulting in a Cluster being marked as Healthy when not initialized yet.
Bug fix	Allows standby servers in clusters with a very high WAL production rate to switch to streaming once they are aligned.
Bug fix	Fix a race condition during the startup of a PostgreSQL pod that could seldom lead to a crash.
Bug fix	Fix a race condition that could lead to a failure initializing the first PVC in a Cluster.
Bug fix	Remove an extra restart of a just demoted primary Pod before joining the Cluster as a replica.
Bug fix	Correctly handle replication-sensitive PostgreSQL configuration parameters when recovering from a backup.
Bug fix	Fix missing validation of PostgreSQL configurations during Cluster creation.

2.83 EDB Postgres for Kubernetes 1.12.0 release notes

Released: 11 Jan 2022

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	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.
Feature	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.
Feature	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.
Feature	Reduce the number of queries executed when calculating the status of an instance.
Feature	Add a readiness probe for PgBouncer.
Feature	Add support for custom Certification Authority of the endpoint of Barman’s backup object store when using Azure protocol.
Bug fix	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.
Bug fix	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.
Bug fix	Fix support for custom Certification Authority of the endpoint of Barman’s backup object store in replica clusters source.
Bug fix	Use a fixed name for default monitoring config map in the cluster namespace.
Bug fix	If the defaulting webhook is not working for any reason, the operator now updates the Cluster with the defaults also during the reconciliation cycle.
Bug fix	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.
Bug fix	Improve log messages from webhooks to also include the object namespace.
Bug fix	Stop logging a “default” message at the start of every reconciliation loop.
Bug fix	Stop logging a PodMonitor deletion on every reconciliation cycle if enablePodMonitor is false.
Bug fix	Do not complain about possible architecture mismatch if a pod is not reachable.

2.84 EDB Postgres for Kubernetes 1.11.0 release notes

Released: 15 Dec 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	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.
Feature	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.
Feature	disableDefaultQueries option (by default set to false).
Feature	Introduce the enablePodMonitor option in the monitoring section of a cluster to automatically manage a PodMonitor resource and seamlessly integrate with Prometheus.
Feature	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.
Feature	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, Cloud Native PostgreSQL waits indefinitely to privilege data durability).
Feature	Handle changes to resource requests and limits for a PostgreSQL Cluster by issuing a rolling update.
Feature	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.
Feature	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.
Feature	Propagate labels and annotations defined at cluster level to the associated resources, including pods (deletions are not supported).
Feature	Automatically remove pods that have been evicted by the Kubelet.
Feature	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).
Feature	Introduce thek8s.enterprisedb.io/reconciliationLoop annotation that, when set to disabled on a given Postgres cluster, prevents the reconciliation loop from running.
Feature	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.
Feature	Support for EDB Postgres Advanced 14.2.
Bug fix	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.
Bug fix	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.
Bug fix	Reduce client-side throttling from Postgres pods (e.g. Waited for 1.182388649s due to client-side throttling, not priority and fairness, request: GET).
Bug fix	Disable Public Key Infrastructure (PKI) initialization on OpenShift and OLM installations, by using the provided one.
Bug fix	When changing configuration parameters that require a restart, always leave the primary as last.
Bug fix	Mark a PVC to be ready only after a job has been completed successfully, preventing a race condition in PVC initialization.
Bug fix	Use the correct public key when renewing the expired webhook TLS secret.
Bug fix	Fix an overflow when parsing an LSN.
Bug fix	Remove stale PID files at startup.
Bug fix	Let the Pooler resource inherit the imagePullSecret defined in the operator, if exists.

2.85 EDB Postgres for Kubernetes 1.10.0 release notes

Released: 11 Nov 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	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.
Feature	Backup Retention Policies: support definition of recovery window retention policies for backups (e.g. ‘30d’ to ensure a recovery window of 30 days).
Feature	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).
Feature	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.
Feature	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.
Feature	New default metrics added to the instance Prometheus exporter: Postgres version, cluster name, and first point of recoverability according to the backup catalog.
Feature	Retry taking a backup after a failure.
Feature	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.
Feature	Integrate the output of the status command of the cnp plugin with information about the backup.
Feature	Introduce a new annotation that reports the status of a PVC (being initialized or ready).
Feature	Set the cluster name in the k8s.enterprisedb.io/cluster label for every object generated in a Cluster, including Backup objects.
Feature	Drop support for deprecated API version postgresql.k8s.enterprisedb.io/v1alpha1 on the Cluster, Backup, and ScheduledBackup kinds.
Feature	Set default operand image to PostgreSQL 14.2.
Security fix	Set allowPrivilegeEscalation to false for the operator containers securityContext.
Bug fix	Disable primary PodDisruptionBudget during maintenance in single-instance clusters.
Bug fix	Use the correct certificate certification authority (CA) during recovery operations.
Bug fix	Prevent Postgres connection leaking when checking WAL archiving status before taking a backup.
Bug fix	Let WAL archive/restore sleep for 100ms following transient errors that would flood logs otherwise.

2.86 EDB Postgres for Kubernetes 1.9.2 release notes

Released: 15 Oct 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	Enhance JSON log with two new loggers: wal-archive for PostgreSQL's archive_command, and wal-restore for restore_command in a standby.
Bug fix	Enable WAL archiving during the standby promotion (prevented .history files from being archived).
Bug fix	Pass the --cloud-provider option to Barman Cloud tools only when using Barman 2.13 or higher to avoid errors with older operands.
Bug fix	Wait for the pod of the primary to be ready before triggering a backup.

2.87 EDB Postgres for Kubernetes 1.9.1 release notes

Released: 30 Sep 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	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.
Bug fix	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.
Bug fix	Properly retry requesting a new backup in case of temporary communication issues with the instance manager.

2.88 EDB Postgres for Kubernetes 1.9.0 release notes

Released: 28 Sep 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	Add Kubernetes 1.22 to the list of supported Kubernetes distributions, and remove 1.16.
Feature	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/EPAS 13+).
Feature	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.
Feature	Expose a default metric for the Prometheus exporter that estimates the number of pages in the pg_catalog.pg_largeobject table in each database.
Feature	Enhance the performance of WAL archiving and fetching, through local in-memory cache.
Bug fix	Explicitly set the postgres user when invoking pg_isready - required by restricted SCC in OpenShift.
Bug fix	Properly update the FirstRecoverabilityPoint in the status.
Bug fix	Set archive_mode = always on the designated primary if backup is requested.

2.89 EDB Postgres for Kubernetes 1.8.0 release notes

Released: 13 Sep 2023

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	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.
Feature	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.
Feature	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.
Feature	Enhance the default Prometheus exporter for a PostgreSQL instance by exposing the following new metrics: number of WAL files and computed total size on disk, number of WAL files and computed total size on disk, number of .ready and .done files in the archive status folder, flag for replica mode, number of requested minimum/maximum synchronous replicas, as well as the expected and actually observed ones.
Feature	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.
Feature	Natively support Azure Blob Storage for backup and recovery, by taking advantage of the feature introduced in Barman 2.13 for Barman Cloud.
Feature	Rely on pg_isready for the liveness probe.
Feature	Support RFC3339 format for timestamp specification in recovery target times.
Feature	Introduce .spec.imagePullPolicy to control the pull policy of image containers for all pods and jobs created for a cluster.
Feature	Add support for OpenShift 4.8, which replaces OpenShift 4.5.
Feature	Support PostgreSQL 14 (beta).
Feature	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).
Feature	Introduce logLevel option to the cluster's spec to specify one of the following levels: error, info, debug or trace.
Security fix	Introduce .spec.enableSuperuserAccess to enable/disable network access with the postgres user through password authentication.
Security fix	Enable specification of a license key in a secret with spec.licenseKeySecret.
Bug fix	Properly inform users when a cluster enters an unrecoverable state and requires human intervention.

2.90 EDB Postgres for Kubernetes 1.7.1 release notes

Released: 11 Aug 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	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.
Feature	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.
Bug fix	Allow the operator to failover when the primary is not ready (bug introduced in 1.7.0).
Bug fix	Execute administrative queries using the LOCAL synchronous commit level.
Bug fix	Correctly parse multi-line log entries in PGAudit.

2.91 EDB Postgres for Kubernetes 1.7.0 release notes

Released: 28 Jul 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	Add native support to PGAudit with a new type of logger called pgaudit directly available in the JSON output.
Feature	Native support for the pg_stat_statements and auto_explain extensions.
Feature	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).
Feature	Exposure of the manual_switchover_required metric to promptly report whether a cluster with primaryUpdateStrategy set to supervised requires a manual switchover.
Feature	Transparently handle shared_preload_libraries for pg_audit, auto_explain and pg_stat_statements.
Feature	Automatic configuration of shared_preload_libraries for PostgreSQL when pg_stat_statements, pgaudit or auto_explain options are added to the postgresql parameters section.
Feature	Support the k8s.enterprisedb.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.
Feature	Add the reload command to the cnp plugin for kubectl to trigger a reconciliation loop on the instances.
Feature	Improve control of pod affinity and anti-affinity configurations through additionalPodAffinity and additionalPodAntiAffinity.
Feature	Introduce a separate PodDisruptionBudget for primary instances, by requiring at least a primary instance to run at any time.
Security fix	Add the .spec.certificates.clientCASecret and spec.certificates.replicationTLSSecret options to define custom client Certification.
Security fix	Authority and certificate for the PostgreSQL server, to be used to authenticate client certificates and secure communication between PostgreSQL nodes.
Security fix	Add the .spec.backup.barmanObjectStore.endpointCA option to define the custom Certification Authority bundle of the endpoint of Barman's backup object store.
Bug fix	Correctly parse histograms in the Prometheus exporter.
Bug fix	Reconcile services created by the operator for a cluster.

2.92 EDB Postgres for Kubernetes 1.6.0 release notes

Released: 12 Jul 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	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.
Feature	EDB Audit support on EDB Postgres Advanced Server images.
Feature	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.
Feature	Add the .spec.affinity.podAntiAffinityType parameter. It can be set to preferred (default), resulting in preferredDuringSchedulingIgnoredDuringExecution being used, or to required, resulting in requiredDuringSchedulingIgnoredDuringExecution.
Security fix	Prevent license keys from appearing in the logs.
Change	Fixed a race condition when deleting a PVC and a pod which prevented the operator from creating a new pod.
Change	Fixed a race condition preventing the manager from detecting the need for a PostgreSQL restart on a configuration change.
Change	Fixed a panic in kubectl-cnp on clusters without annotations.
Change	Lowered the level of some log messages to debug.
Change	E2E tests for server CA and TLS injection.

2.93 EDB Postgres for Kubernetes 1.5.1 release notes

Released: 11 Jun 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Change	Fix a bug with CRD validation preventing auto-update with Operator Deployments on Red Hat OpenShift.
Change	Allow passing operator's configuration using a Secret.

2.94 EDB Postgres for Kubernetes 1.5.0 release notes

Released: 17 Jun 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	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.
Feature	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.
Feature	Enable specification of a digest to an image name, through the `<image>:<tag>@sha256:<digestValue>` format, for more deterministic and repeatable deployments.
Security fix	Customize TLS certificates to authenticate the PostgreSQL server by defining secrets for the server certificate and the related Certification Authority that signed it.
Security fix	Raise the sslmode for the WAL receiver process of internal and automatically managed streaming replicas from require to verify-ca.
Change	Enhance the promote subcommand of the cnp plugin for kubectl to accept just the node number rather than the whole name of the pod.
Change	Adopt DNS-1035 validation scheme for cluster names (from which service names are inherited).
Change	Enforce streaming replication connection when cloning a standby instance or when bootstrapping using the pg_basebackup method.
Change	Integrate the Backup resource with beginWal, endWal, beginLSN, endLSN, startedAt and stoppedAt regarding the physical base backup.
Documentation fix	Provide a list of ports exposed by the operator and the operand container.
Documentation fix	Introduce the cnp-bench helm charts and guidelines for benchmarking the storage and PostgreSQL for database workloads.
E2E test fix	Test Kubernetes 1.21.
E2E test fix	Add test for High Availability of the operator.
E2E test fix	Add test for node draining.
Bug fix	Timeout to pg_ctl start during recovery operations too short.
Bug fix	Operator not watching over direct events on PVCs.
Bug fix	Fix handling of immediateCheckpoint and jobs parameter in barmanObjectStore backups.
Bug fix	Empty logs when recovering from a backup.

2.95 EDB Postgres for Kubernetes 1.4.0 release notes

Released: 18 May 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	Standard output logging of PostgreSQL error messages in JSON format.
Feature	Provide a basic set of PostgreSQL metrics for the Prometheus exporter.
Feature	Add the restart command to the cnp plugin for kubectl to restart the pods of a given PostgreSQL cluster in a rollout fashion.
Security fix	Set readOnlyRootFilesystem security context for pods.
Change	IMPORTANT: If you have previously deployed the Cloud Native PostgreSQL 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 Cloud Native PostgreSQL documentation for additional detail on upgrading to 1.4.0.
Change	Fix the labels that are automatically defined by the operator, renaming them from control-plane: controller-manager to app.kubernetes.io/name: cloud-native-postgresql.
Change	Assign the metrics name to the TCP port for the Prometheus exporter.
Change	Set cnp_metrics_exporter as the application_name to the metrics exporter connection in PostgreSQL.
Change	When available, use the application database for monitoring queries of the Prometheus exporter instead of the postgres database.
Documentation fixes	Customization of monitoring queries.
Documentation fixes	Operator upgrade instructions.
Bug fix	Avoid using -R when calling pg_basebackup.
Bug fix	Remove stack trace from error log when getting the status.

2.96 EDB Postgres for Kubernetes 1.3.0 release notes

Released: 23 Apr 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	Inheritance of labels and annotations.
Feature	Set resource limits for every container.
Security fix	Support for restricted security context constraint on Red Hat OpenShift to limit pod execution to a namespace allocated UID and SELinux context.
Security fix	Pod security contexts explicitly defined by the operator to run as non-root, non-privileged and without privilege escalation.
Change	Prometheus exporter endpoint listening on port 9187 (port 8000 is now reserved to instance coordination with API server).

2.97 EDB Postgres for Kubernetes 1.2.1 release notes

Released: 06 Apr 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	ScheduledBackup are no longer owners of the Backups, meaning that backups are not removed when ScheduledBackup objects are deleted.
Security fix	Update on ubi8-minimal image to solve RHSA-2021:1024 (Security Advisory: Important).

2.98 EDB Postgres for Kubernetes 1.2.0 release notes

Released: 31 Mar 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	Introduce experimental support for custom monitoring queries as ConfigMap and Secret objects using a compatible syntax with postgres_exporter for Prometheus.
Feature	Support Operator Lifecycle Manager (OLM) deployments, with the subsequent presence on OperatorHub.io.
Feature	Expand license key support for company-wide usage (previous restrictions limited only to a single cluster namespace).
Feature	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.

2.99 EDB Postgres for Kubernetes 1.1 release notes

Released: 03 Mar 2021

This release of EDB Postgres for Kubernetes includes the following:

Type	Description
Feature	Add kubectl cnp status to pretty-print the status of a cluster, including JSON and YAML output.
Feature	Add kubectl cnp certificate to enable TLS authentication for client applications.
Feature	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.
Feature	Rollback scaling down a cluster to a value lower than maxSyncReplicas.
Feature	Request a checkpoint before demoting a former primary.
Feature	Send SIGINT signal (fast shutdown) to PostgreSQL process on SIGTERM.

2.100 EDB Postgres for Kubernetes 1.0 release notes

Released: 04 Feb 2021

The first major release implements Cluster, Backup and ScheduledBackup in the API group postgresql.k8s.enterprisedb.io/v1. It uses these resources to create and manage PostgreSQL clusters inside Kubernetes with the following main capabilities:

Type	Description
Feature	Direct integration with Kubernetes API server for High Availability, without requiring an external tool.
Feature	Failover of the primary instance by promoting the most aligned replica.
Feature	Automated recreation of a replica.
Feature	Planned switchover of the primary instance by promoting a selected replica.
Feature	Scale up/down capabilities.
Feature	Definition of an arbitrary number of instances (minimum 1 - one primary server).
Feature	Definition of the read-write service to connect your applications to the only primary server of the cluster.
Feature	Definition of the read service to connect your applications to any of the instances for reading workloads.
Feature	Support for Local Persistent Volumes with PVC templates.
Feature	Reuse of Persistent Volumes storage in Pods.
Feature	Rolling updates for PostgreSQL minor versions and operator upgrades.
Feature	TLS connections and client certificate authentication.
Feature	Continuous backup to an S3 compatible object store.
Feature	Full recovery and point-in-time recovery from an S3 compatible object store backup.
Feature	Support for synchronous replicas.
Feature	Support for node affinity via nodeSelector property.
Feature	Standard output logging of PostgreSQL error messages.

2.101 EDB Postgres for Kubernetes 0.8 release notes

Released: 29 Jan 2021

Type	Description
Feature	Upgraded API version to v1.
Feature	Implement node affinity via `nodeSelector`.
Feature	Activate `AntiAffinity` by default.
Feature	Remove completed `Jobs` from the cluster.
Feature	Upgrade controller-runtime to v0.8.1.
Change	Build container images using schema version 2.
Change	Enhance E2E tests by covering more cases, robustness, and reliability improvements.
Fix	Bug fixes and code improvements.

2.102 EDB Postgres for Kubernetes 0.7 release notes

Released: 31 Dec 2020

Type	Description
Feature	Support `pg_rewind` if needed following a failover or switchover to let the former primary act as a standby.
Feature	Add persistent volume expansion support, if permitted by the storage class.
Enhancement	Enhance metrics exporter for PostgreSQL.
Feature	Add support for Kubernetes version 1.20.
Feature	Drop support for Kubernetes version 1.15.
Feature	Refactor E2E tests, including conversion to Github actions.
Fix	Bug fixes and code improvements.

2.103 EDB Postgres for Kubernetes 0.6 release notes

Released: 4 Dec 2020

Type	Description
Feature	Add Point-In-Time Recovery based on timestamp, target name, or transaction Id, as well as the specification of the timeline, through a new bootstrap method option called `recoveryTarget`.
Feature	Add Synchronous Streaming Replication support through the `minSyncReplicas` and `maxSyncReplicas` cluster options, defining respectively the expected minimum and maximum number of synchronous standby servers at any time (disabled by default)
Feature	Support EDB Postgres Advanced Server (EPAS).
Feature	Configure `initdb` options for the bootstrap of an empty cluster (`initDb`).
Feature	Enable/Disable Redwood compatibility level with EPAS.
Feature	Extend the instance manager with a new framework for the export of metrics for Prometheus - currently supporting pg_stat_archiver only.
Feature	Use Kubernetes jobs instead of init containers to perform cluster initialization procedures (including standby creation and recovery) and improve their observability.
Feature	Record Kubernetes events to be used by `kubectl describe` and `kubectl get events`.
Feature	Introduce Kubernetes expectations for Pods, PVCs, and Jobs to prevent race conditions.
Feature	Set `application_name` in PostgreSQL to the name of the Pod/instance.
Feature	The `fullRecovery` bootstrap mode has been renamed to `recovery` to address also Point-In-Time Recovery
Fix	Bug fixes and code improvements.

2.104 EDB Postgres for Kubernetes 0.5 release notes

Released: 20 Nov 2020

Type	Description
Feature	Automated provisioning of an independent Certification Authority (CA) for each PostgreSQL cluster.
Feature	Transparent and native support for TLS/SSL connections to encrypt client/server communications.
Enhancement	Improve the security of the standby streaming replication channel through a dedicated and fixed database user called `streaming_replica` with sole `REPLICATION` privileges and II. an automatically managed X.509 TLS certificate signed by the cluster Certification Authority to authenticate the `streaming_replica` user.
Enhancement	Improve the security of the standby streaming replication channel through an automatically managed X.509 TLS certificate signed by the cluster Certification Authority to authenticate the `streaming_replica` user.
Enhancement	Improve PostgreSQL configuration capability through a mutating webhook that prevents users from changing those parameters that are directly managed by the operator.
Enhancement	Improve PostgreSQL configuration capability through a defaulting webhook that integrates the users' supplied configuration options with default values in the cluster state.
Enhancement	Improve PostgreSQL configuration capability through automated management of the PostgreSQL instances reload and restart.
Feature	Enable custom and independent configuration of a PostgreSQL cluster following a recovery from a backup.
Feature	Convey the current status of the cluster (i.e. healthy, failover in progress, switchover in progress).
Feature	API change from k8s.2ndq.io to k8s.enterprisedb.io.

2.105 EDB Postgres for Kubernetes 0.4 release notes

Released: 5 Nov 2020

Type	Description
Feature	Support for full recovery from a backup.
Feature	Add `bootstrap` section to configure how to initiate a cluster: `initDB` or `fullRecovery` .
Feature	Introduce defaulting and validating webhooks.
Feature	Simplify configuration (convention over configuration).
Feature	Constrain rolling upgrades to the same PostgreSQL major version.
Feature	End-to-end tests run on GKE, AKS and GKS.
Enhancement	Documentation improvements.
Enhancement	Bug fixes and minor improvements.

2.106 EDB Postgres for Kubernetes 0.3 release notes

Released: 25 Sep 2020

Type	Description
Feature	Support for PostgreSQL 13.
Feature	Remove `emptyDir` volume storage support.
Feature	Node maintenance support for PostgreSQL clusters with local storage through the `nodeMaintenanceWindow` parameter.
Enhancement	Improve PostgreSQL configuration management through the usage of a dictionary supporting default, required and fixed values.
Feature	Use MD5 as authentication method for inter-cluster communication, with automated password creation in a secret.
Feature	Remove need for "trust" authentication method.
Feature	`spec.postgresql.pg_hba` is now optional, defaulting to MD5 authentication required for communication between Pods and `peer` for in-Pod communication.
Feature	Remove "unusable" annotation support, now PVC can just be removed followed by a deletion of the corresponding Pod.
Feature	Support for license keys, including implicit 30 day trial version.
Fix	Bug fixes and minor improvements.
Feature	Update from an earlier version not supported.

2.107 EDB Postgres for Kubernetes 0.2 release notes

Released: 11 Aug 2020

Type	Description
Feature	PostgreSQL 10 and 11 are now supported, in addition to PostgreSQL 12.
Feature	Usage of UBI as base image of the operator (OpenShift support).
Feature	New Backup and ScheduledBackup CRD, allowing users to take a physical backup of the cluster in an object store that complies with the S3 protocol (such as AWS S3, MinIO, MinIO Gateway).
Feature	Support for WAL archiving to an object store that complies with the S3 protocol.
Enhancement	Improvements in the E2e tests infrastructure.
Fix	Bug fixes and minor improvements.

2.108 EDB Postgres for Kubernetes 0.1 release notes

Released: 3 Apr 2020

Type	Description
Feature	Reuse of existing persistent storage with Pods (required for rolling updates).
Feature	Independence between operator container image and PostgreSQL container image (required by rolling updates) - this enables usage of Community PostgreSQL images.
Feature	Rolling updates for the update of the operator and the entire cluster to a new version of PostgreSQL, by updating all the replicas first; switchover can be entirely managed by Kubernetes once replicas have been updated (`unsupervised` option), or manually triggered by the user (`supervised` option).
Feature	Check framework for update-in-progress of a Pod.
Enhancement	Improvements in cluster status information.
Enhancement	E2E tests for Kubernetes 1.18.
Enhancement	E2E performance tests for failover (< 5 seconds).
Enhancement	Improvements in E2E tests for switchover.
Feature	Support for PostgreSQL's `cluster_name` GUC.
Feature	New documentation section `Exposing Postgres services` .
Feature	New documentation section `Security` .

2.109 EDB Postgres for Kubernetes 0.0.1 release notes

Released: 5 Mar 2020

Type	Description
Feature	PostgreSQL 12.2 container image.
Feature	Self-Healing capability, through failover of the primary instance, by promoting the most aligned replica.
Feature	Self-Healing capability, through automated recreation of a replica.
Feature	Planned switchover of the primary instance, by promoting a selected replica.
Feature	Scale up/down capabilities.
Feature	Definition of an arbitrary number of instances (minimum 1 - one primary server).
Feature	Definition of the *read-write* service, to connect your applications to the only primary server of the cluster.
Feature	Definition of the *read-only* service, to connect your applications to any of the instances for read workloads.
Feature	Support for Local Persistent Volumes with PVC templates.
Feature	Standard output logging of PostgreSQL error messages.

3 Before You Start

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

Kubernetes terminology

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

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 EDB Postgres for Kubernetes. 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.

EDB Postgres for Kubernetes requires a Kubernetes version supported by the community. Please refer to the "Supported releases" page for details.

PostgreSQL terminology

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

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

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

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

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

Replica Cluster : A EDB Postgres for Kubernetes `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, EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes' 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.

Cloud terminology

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

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

What to do next

Now that you have familiarized with the terminology, you can decide to test EDB Postgres for Kubernetes on your laptop using a local cluster before deploying the operator in your selected cloud environment.

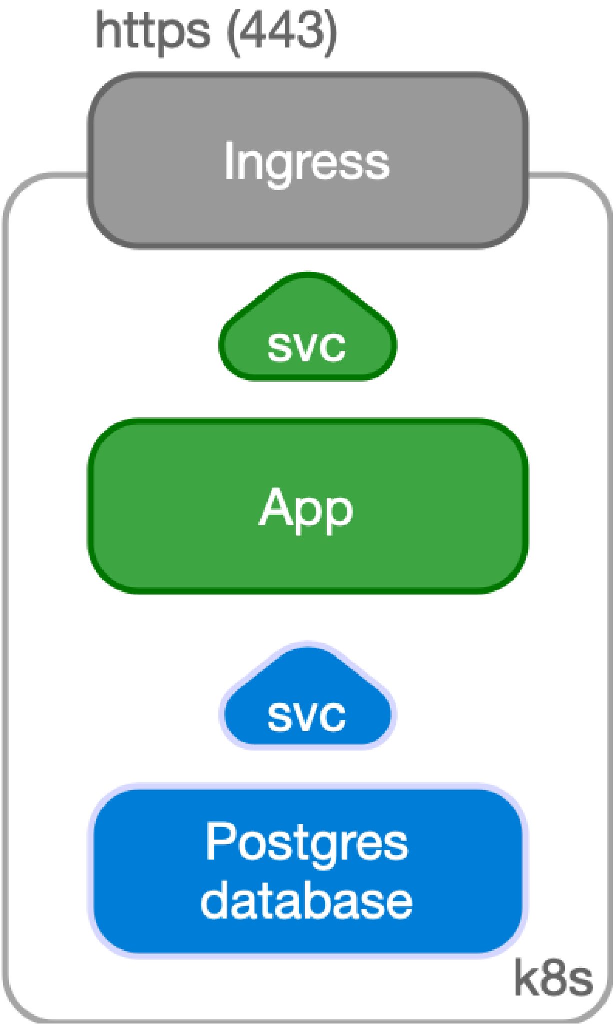
4 Use cases

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

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

Case 1: Applications inside Kubernetes

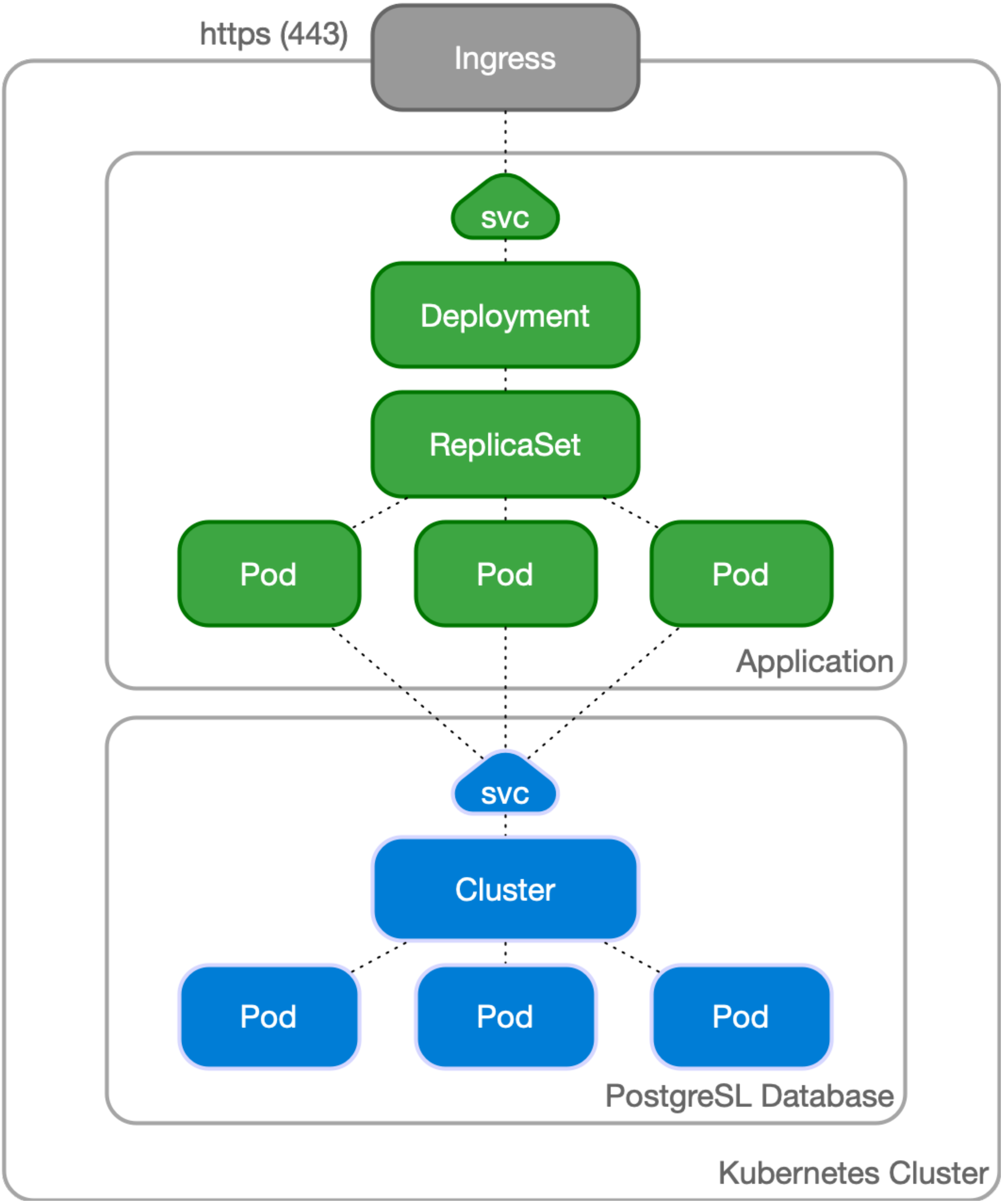
In a typical situation, the application and the database run in the same namespace inside a Kubernetes cluster.



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

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

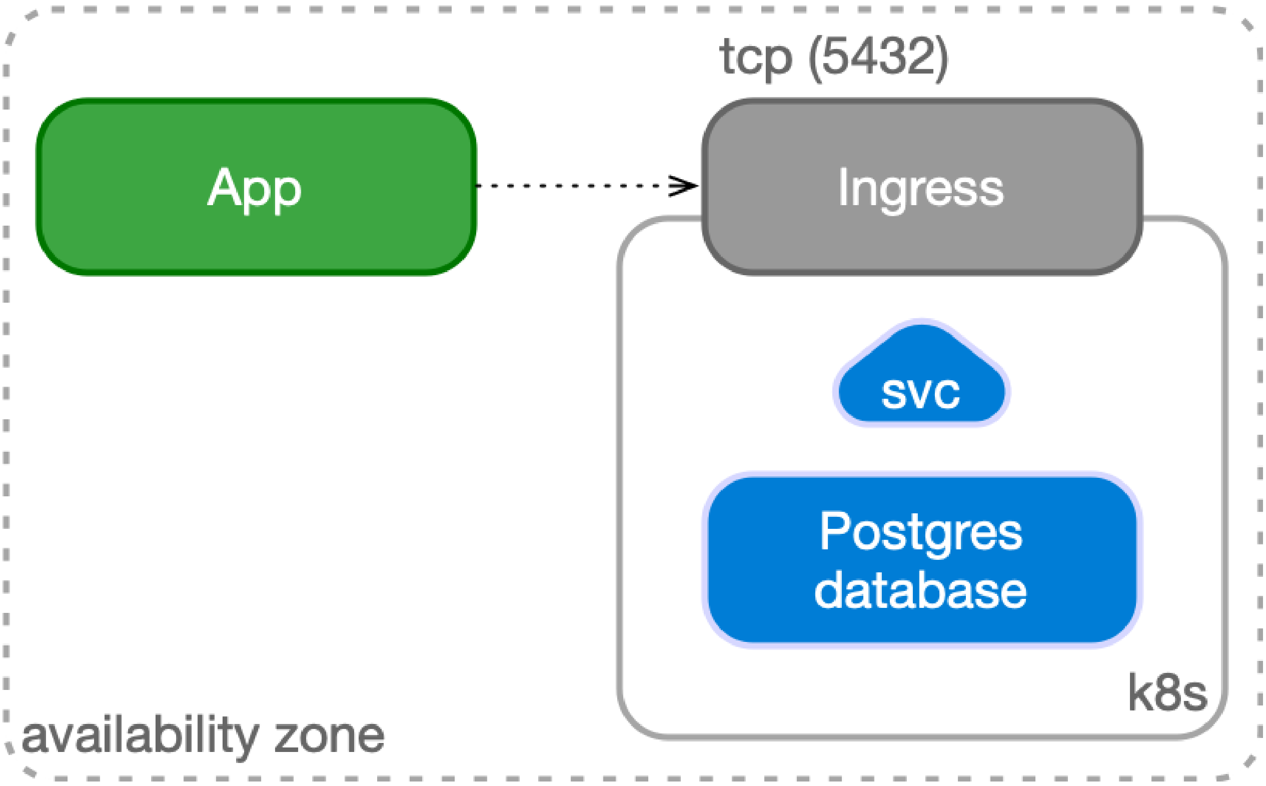
The application uses the backend PostgreSQL database to keep track of the state in a reliable and persistent way. The application refers to the read-write service exposed by the `Cluster` resource defined by EDB Postgres for Kubernetes, which points to the current primary instance, through a TLS connection.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


https (443)
Ingress
svc
Deployment
ReplicaSet
Pod
Pod
Pod
Application
svc
Cluster
Pod
Pod
Pod
PostgreSL Database
Kubernetes Cluster

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.



5 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:

- **Deployments in *stretched* vs. *non-stretched* clusters**: Evaluating the differences between deploying in stretched clusters (across 3 or more availability zones) versus non-stretched clusters (within a single availability zone).
- **Reservation of `postgres` worker nodes**: Isolating PostgreSQL workloads by dedicating specific worker nodes to `postgres` tasks, ensuring optimal performance and minimizing interference from other workloads.
- **PostgreSQL architectures within a single Kubernetes cluster**: Designing effective PostgreSQL deployments within a single Kubernetes cluster to meet high availability and performance requirements.
- **PostgreSQL architectures across Kubernetes clusters for disaster recovery**: Planning and implementing PostgreSQL architectures that span multiple Kubernetes clusters to provide comprehensive disaster recovery capabilities.

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

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes allows you to adopt that strategy. For more information, please refer to the talk given by Chris Milsted and Gabriele Bartolini at KubeCon NA 2022 entitled "Data On Kubernetes, Deploying And Running PostgreSQL And Patterns For Databases In a Kubernetes Cluster" where this topic was covered in detail.

Kubernetes architecture

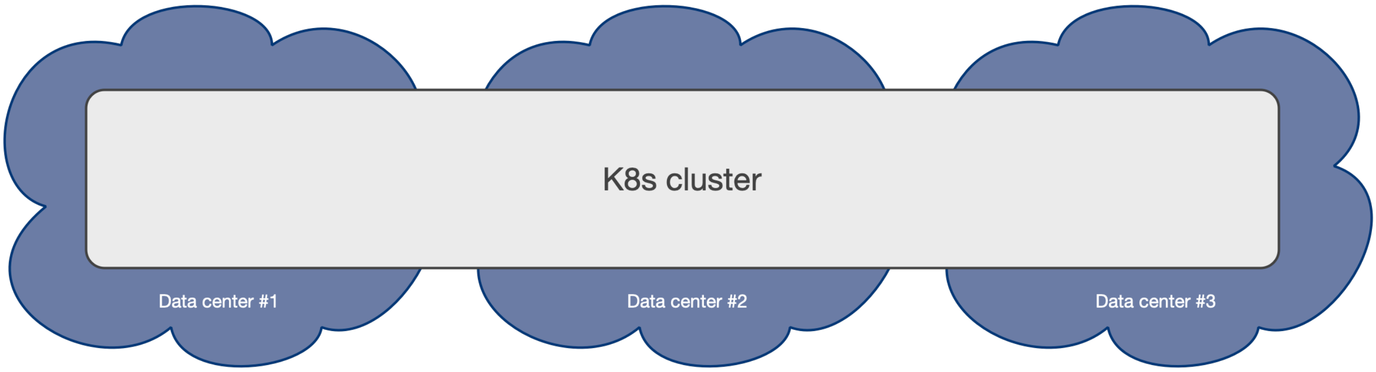
Kubernetes natively provides the possibility to span separate physical locations - also known as data centers, failure zones, or more frequently **availability zones** - connected to each other via redundant, low-latency, private network connectivity.

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

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

Multi-availability zone Kubernetes clusters

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

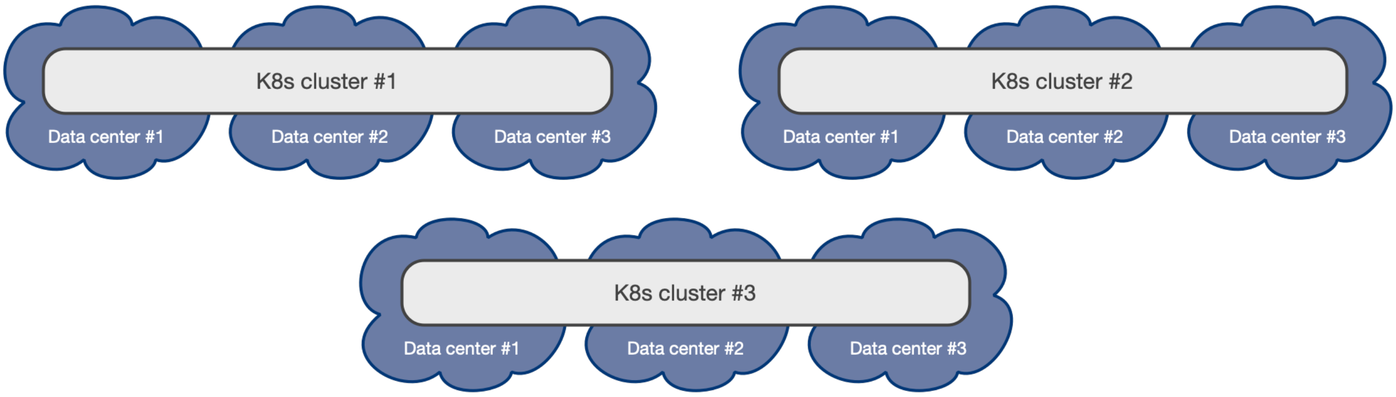


Such an architecture enables the EDB Postgres for Kubernetes operator to control the full lifecycle of a `Cluster` resource across the zones within a single Kubernetes cluster, by treating all the availability zones as active: this includes, among other operations, scheduling the workloads in a declarative 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 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.

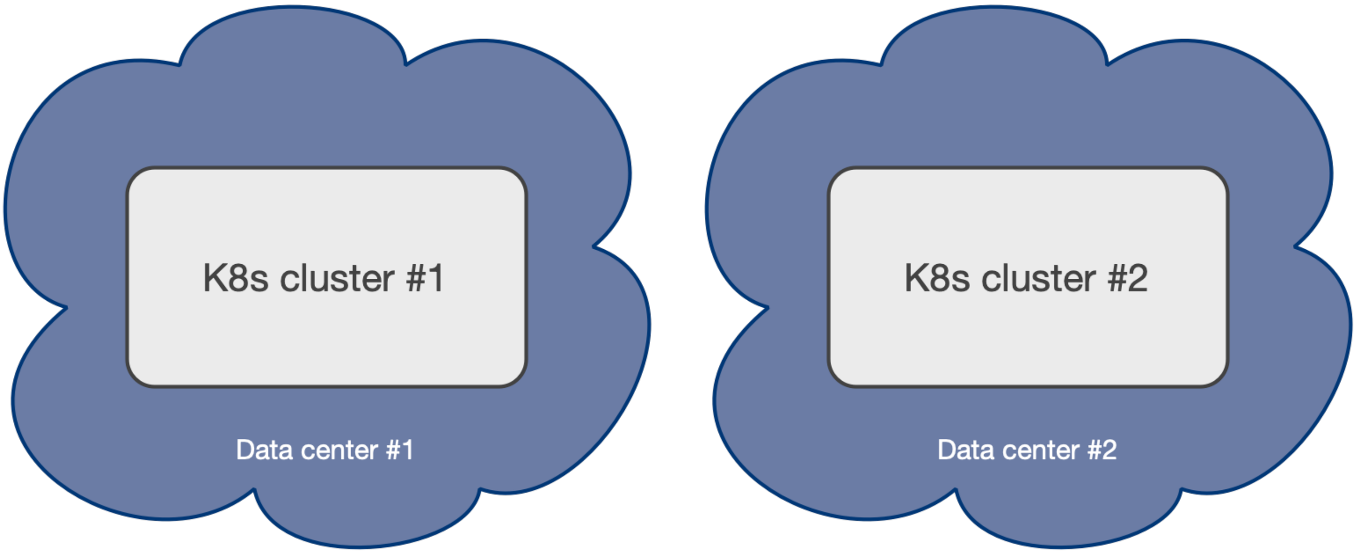


Single availability zone Kubernetes clusters

If your Kubernetes cluster has only one availability zone, EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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" 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 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 EDB Postgres for Kubernetes' scope, as the operator can only function within a single Kubernetes cluster.

> **Important**
>
> EDB Postgres for Kubernetes 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.

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` .

Proposed node label

EDB Postgres for Kubernetes 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: ""
```

Proposed node taint

EDB Postgres for Kubernetes 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:

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

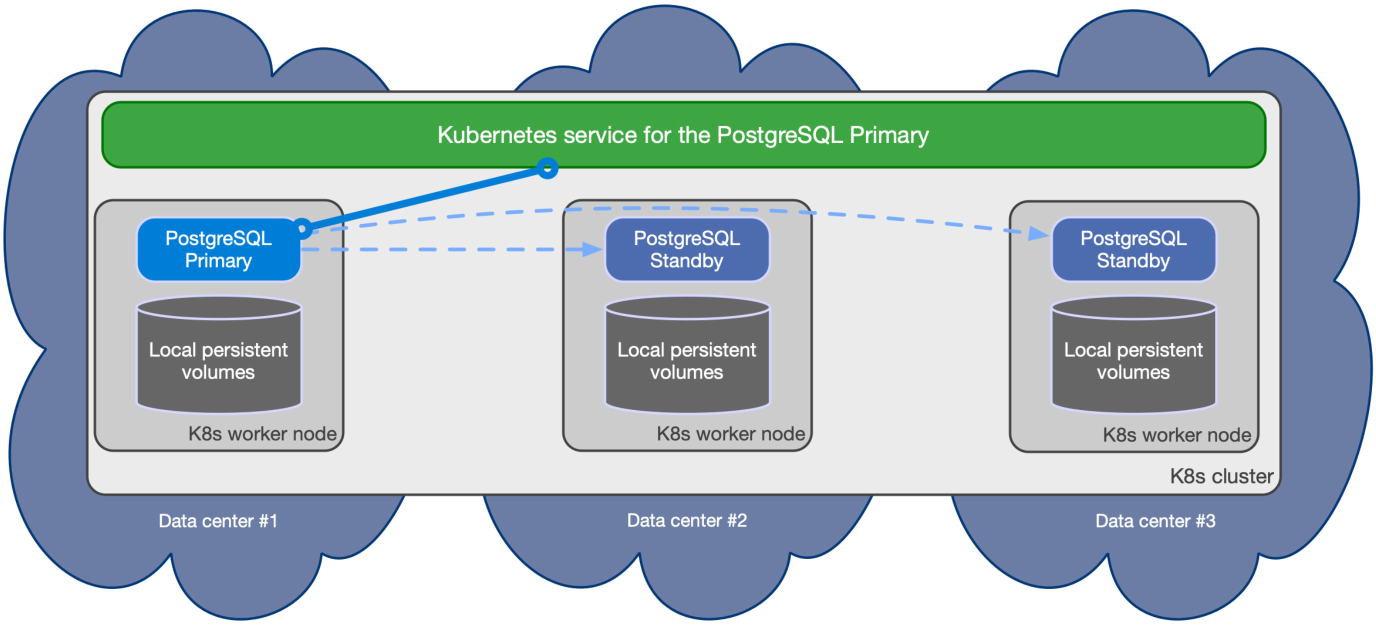
PostgreSQL architecture

EDB Postgres for Kubernetes 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" section 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.



EDB Postgres for Kubernetes 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.

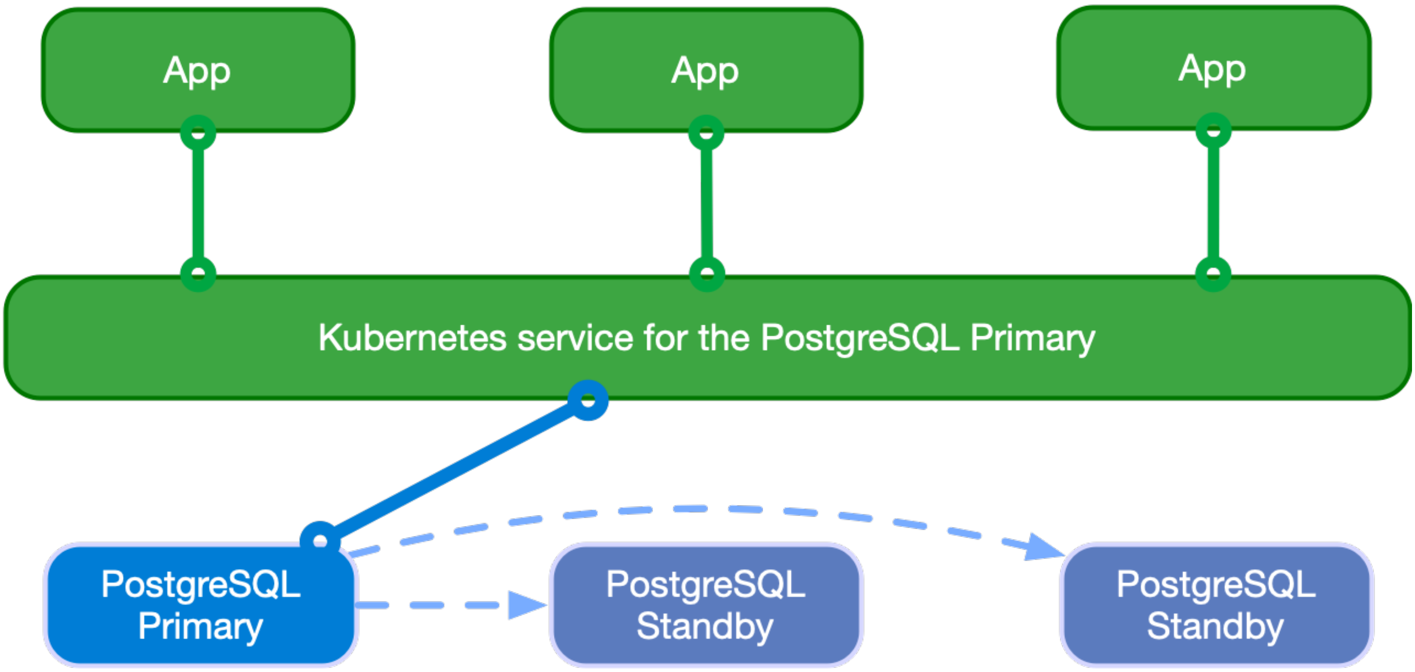
> **Replication**
>
> Please refer to the "Replication" section for more information about how EDB Postgres for Kubernetes relies on PostgreSQL replication, including synchronous settings.

> **Connecting from an application**
>
> Please refer to the "Connecting from an application" section for information about how to connect to EDB Postgres for Kubernetes from a stateless application within the same Kubernetes cluster.

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

Read-write workloads

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



Applications can use the `-rw` suffix service.

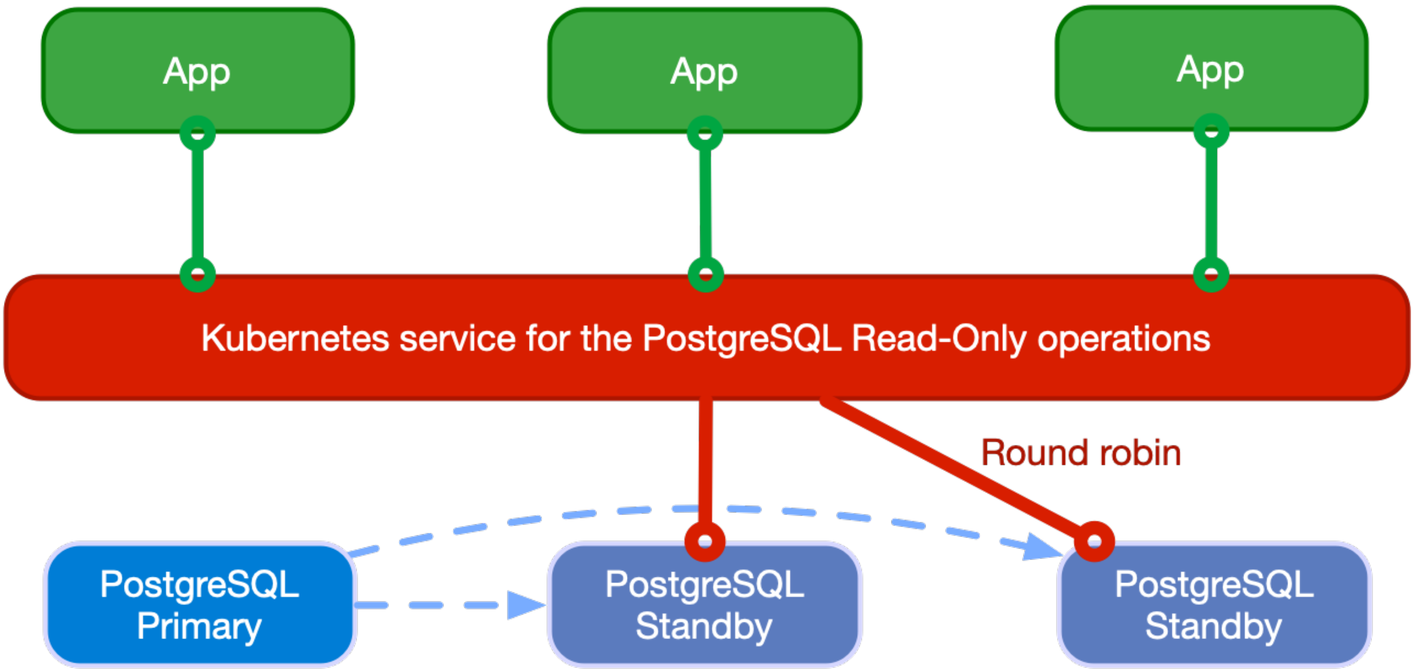
In case of temporary or permanent unavailability of the primary, for High Availability purposes EDB Postgres for Kubernetes will trigger a failover, pointing the `-rw` service to another instance of the cluster.

Read-only workloads

> **Important**
>
> Applications must be aware of the limitations that Hot Standby presents and familiar with the way PostgreSQL operates when dealing with these workloads.

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

The following diagram shows the architecture:



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

Deployments across Kubernetes clusters

> **Info**
>
> EDB Postgres for Kubernetes 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.

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

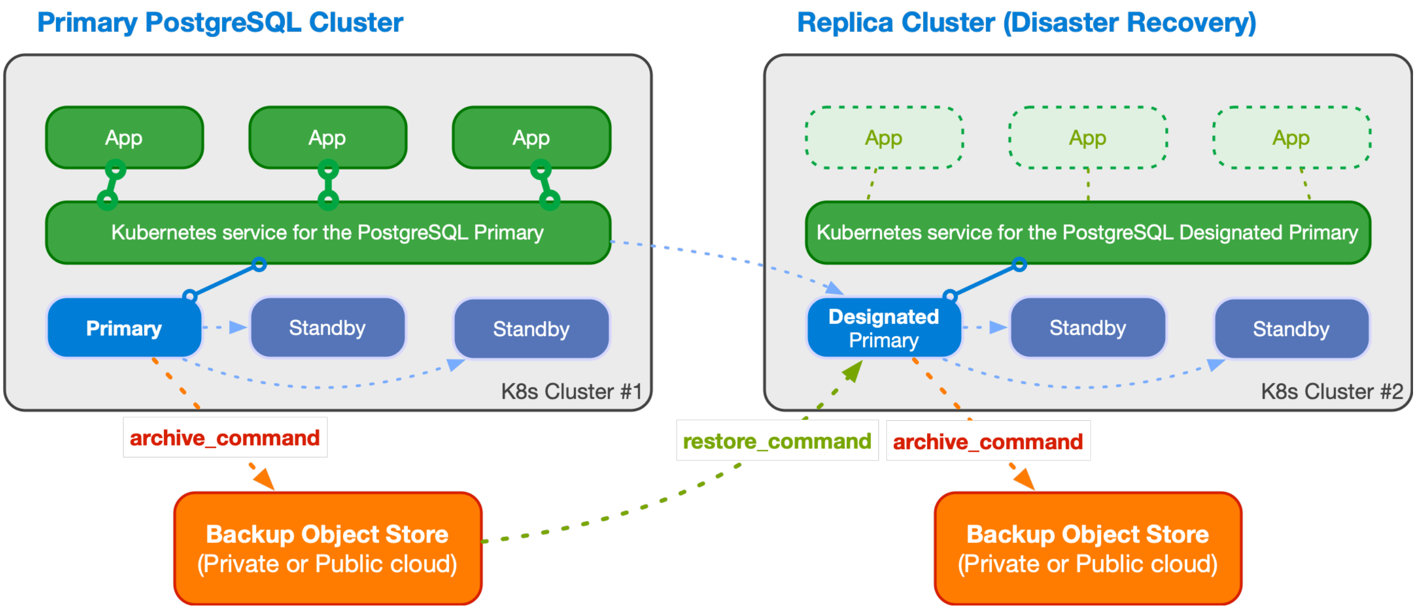
However, for business continuity objectives it is fundamental to:

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

In order to address the above concerns, EDB Postgres for Kubernetes 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.



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 EDB Postgres for Kubernetes.
- **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
>
> EDB Postgres for Kubernetes 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
>
> EDB Postgres for Kubernetes 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`.

EDB Postgres for Kubernetes 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.

> Replica clusters
>
> Please refer to the "Replica Clusters" section 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.

6 Installation and upgrades

> **OpenShift**
>
> For instructions on how to install Cloud Native PostgreSQL on Red Hat OpenShift Container Platform, please refer to the "OpenShift" section.

> **Warning**
>
> OLM (via operatorhub.io is no longer supported as an installation method for EDB Postgres for Kubernetes.

Installation on Kubernetes

Obtaining an EDB subscription token

> **Important**
>
> You must obtain an EDB subscription token to install EDB Postgres for Kubernetes. Without a token, you will not be able to access the EDB private software repositories.

Installing EDB Postgres for Kubernetes requires an EDB Repos 2.0 token to gain access to the EDB private software repositories.

You can obtain the token by visiting your EDB Account Profile. You will have to sign in if you are not already logged in.

Your account profile page displays the token to use next to **Repos 2.0 Token** label. By default, the token is obscured, click the "Show" button (an eye icon) to reveal it.

Your token entitles you to access one of two repositories: standard or enterprise.

- `standard` - Includes the operator and the EDB Postgres Extended operand images.
- `enterprise` - Includes the operator and the EDB Postgres Advanced and EDB Postgres Extended operand images.

Set the relevant value, determined by your subscription, as an environment variable `EDB_SUBSCRIPTION_PLAN`.

```
EDB_SUBSCRIPTION_PLAN=enterprise
```

then set the Repos 2.0 token to an environment variable `EDB_SUBSCRIPTION_TOKEN`.

```
EDB_SUBSCRIPTION_TOKEN=<your-token>
```

> **Warning**
>
> The token is sensitive information. Please ensure that you don't expose it to unauthorized users.

You can now proceed with the installation.

Using the Helm Chart

The operator can be installed using the provided Helm chart.

Directly using the operator manifest

Install the EDB pull secret

Before installing EDB Postgres for Kubernetes, you need to create a pull secret for EDB software in the `postgresql-operator-system` namespace.

The pull secret needs to be saved in the namespace where the operator will reside. Create the `postgresql-operator-system` namespace using this command:

```
kubectl create namespace postgresql-operator-system
```

To create the pull secret itself, run the following command:

```
kubectl create secret -n postgresql-operator-system docker-registry edb-pull-secret \
  --docker-server=docker.enterprisedb.com \
  --docker-username=k8s_$EDB_SUBSCRIPTION_PLAN \
  --docker-password=$EDB_SUBSCRIPTION_TOKEN
```

Install the operator

Now that the pull-secret has been added to the namespace, the operator can be installed like any other resource in Kubernetes, through a YAML manifest applied via `kubectl`.

There are two different manifests available depending on your subscription plan:

- Standard: The latest standard operator manifest.
- Enterprise: The latest enterprise operator manifest.

You can install the manifest for the latest version of the operator by running:

```
kubectl apply --server-side -f \
  https://get.enterprisedb.io/pg4k/pg4k-$EDB_SUBSCRIPTION_PLAN-1.26.0.yaml
```

You can verify that with:

```
kubectl rollout status deployment
\
  -n postgresql-operator-system postgresql-operator-controller-
manager
```

Using the `cnp` plugin for `kubectl`

You can use the `cnp` 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 cnp install generate \
  --watch-namespace "specific-namespace" \
  > cnp_for_specific_namespace.yaml
```

See the "`cnp` plugin" documentation for a more comprehensive example.

> **Warning**
>
> If you are deploying EDB Postgres for Kubernetes 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.

Details about the deployment

In Kubernetes, the operator is by default installed in the `postgresql-operator-system` namespace as a Kubernetes `Deployment`. The name of this deployment depends on the installation method. When installed through the manifest or the `cnp` plugin, by default, it is called `postgresql-operator-controller-manager`. When installed via Helm, by default, the deployment name is derived from the helm release name, appended with the suffix `-edb-postgres-for-kubernetes` (e.g., `<name>-edb-postgres-for-kubernetes`).

> **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 postgresql-operator-
system
NAME                 READY   UP-TO-DATE   AVAILABLE
AGE
<deployment-name>    1/1     1            1
18m
```

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

As with any Deployment, it sits on top of a ReplicaSet and supports rolling upgrades. The default configuration of the EDB Postgres for Kubernetes 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).

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

Upgrades

> **Important**
>
> Please carefully read the release notes before performing an upgrade as some versions might require extra steps.

Upgrading EDB Postgres for Kubernetes 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 `cnp` plugin for `kubectl`.

> **Rolling updates**
>
> 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.

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. EDB Postgres for Kubernetes 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`).

In-place updates of the instance manager

By default, EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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.

Compatibility among versions

EDB Postgres for Kubernetes 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.

EDB Postgres for Kubernetes 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 page contains a detailed list of the changes introduced in every released version of EDB Postgres for Kubernetes, and it must be read before upgrading to a newer version of the software.

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

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

Upgrading to 1.26.0 or 1.25.2

> **Important**
>
> We strongly recommend that all EDB Postgres for Kubernetes users upgrade to version 1.26.0 or at least to the latest stable version of the minor release you are currently using (namely 1.25.x).

In this release, the `cnp` 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.

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.

Upgrading to 1.24 from a previous minor version

From Replica Clusters to Distributed Topology

One of the key enhancements in EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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 "Distributed Topology" section for replica clusters.

7 Quickstart

This section guides you through testing a PostgreSQL cluster on your local machine by deploying EDB Postgres for Kubernetes on a local Kubernetes cluster using either Kind or Minikube.

Red Hat OpenShift Container Platform users can test the certified operator for EDB Postgres for Kubernetes on the Red Hat OpenShift Local (formerly Red Hat CodeReady Containers).

> **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, EDB Postgres for Kubernetes is deployed using regular manifests written in YAML.

By following these instructions you should be able to start a PostgreSQL cluster on your local Kubernetes/Openshift installation and experiment with it.

> **Important**
>
> Make sure that you have `kubectl` installed on your machine in order to connect to the Kubernetes cluster, or `oc` if using OpenShift Local. Please follow the Kubernetes documentation on how to install `kubectl` or the Openshift documentation on how to install `oc`.

> **Note**
>
> If you are running Openshift, use `oc` every time `kubectl` is mentioned in this documentation. `kubectl` commands are compatible with `oc` ones.

Part 1 - Setup the local Kubernetes/Openshift Local playground

The first part is about installing Minikube, Kind, or OpenShift Local. 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

Minikube

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

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

```
minikube start
```

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

```
kubectl get
nodes
```

You will see one node called `minikube`.

Kind

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

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

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

OpenShift Local (formerly CodeReady Containers (CRC))

1. Download OpenShift Local and move the binary inside a directory in your `PATH`.

2. Run the following commands:

   ```
   crc setup
   crc start
   ```

   The `crc start` output will explain how to proceed.

3. Execute the output of the `crc oc-env` command.

4. Log in as `kubeadmin` with the printed `oc login` command. You can also open the web console running `crc console`. User and password are the same as for the `oc login` command.

5. OpenShift Local doesn't come with a StorageClass, so one has to be configured. Follow the Dynamic volume provisioning wiki page and install `rancher/local-path-provisioner`.

Part 2: Install EDB Postgres for Kubernetes

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

Unless specified in a cluster configuration file, EDB Postgres for Kubernetes will currently deploy Community Postgresql operands by default. See the section Deploying EDB Postgres servers for more information.

Refer to the "Installation" section and then proceed with the deployment of a PostgreSQL cluster.

Part 3: Deploy a PostgreSQL cluster

Unless specified in a cluster configuration file, EDB Postgres for Kubernetes will currently deploy Community Postgresql operands by default. See the section Deploying EDB Postgres servers for more information.

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

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

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  instances: 3

  storage:
    size:
1Gi
```

> **There's more**
>
> For more detailed information about the available options, please refer to the"API Reference" section.

In order to create the 3-node Community 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 `k8s.enterprisedb.io/cluster` label on all objects relevant to a particular cluster. For example:

```
kubectl get pods -l
k8s.enterprisedb.io/cluster=<CLUSTER>
```

> **Important**
>
> Note that we are using `k8s.enterprisedb.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 `k8s.enterprisedb.io/cluster`.

Deploying EDB Postgres servers

By default, the operator will install the latest available minor version of the latest major version of Community 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 EDB Postgres Advanced 16.4 you can use:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
   #
[...]
spec:
   #
[...]
     imageName: docker.enterprisedb.com/k8s_enterprise/edb-postgres-advanced:16
   #
[...]
```

And to install EDB Postgres Extended 16 you can use:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  #
[...]
spec:
  #
[...]
    imageName: docker.enterprisedb.com/k8s_enterprise/edb-postgres-extended:16
  #
[...]
```

> **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.

> **There's more**
>
> There are some examples cluster configurations bundled with the operator. Please refer to the "Examples" section.

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 EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes clusters.

Installation

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

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

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/EnterpriseDB/docs/main/product_docs/docs/postgres_for_kubernetes/1/samples/
itoring/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.

> **Seealso**
>
> 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
```

Viewing with Prometheus

At this point, a EDB Postgres for Kubernetes cluster deployed with monitoring activated would be observable via Prometheus.

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

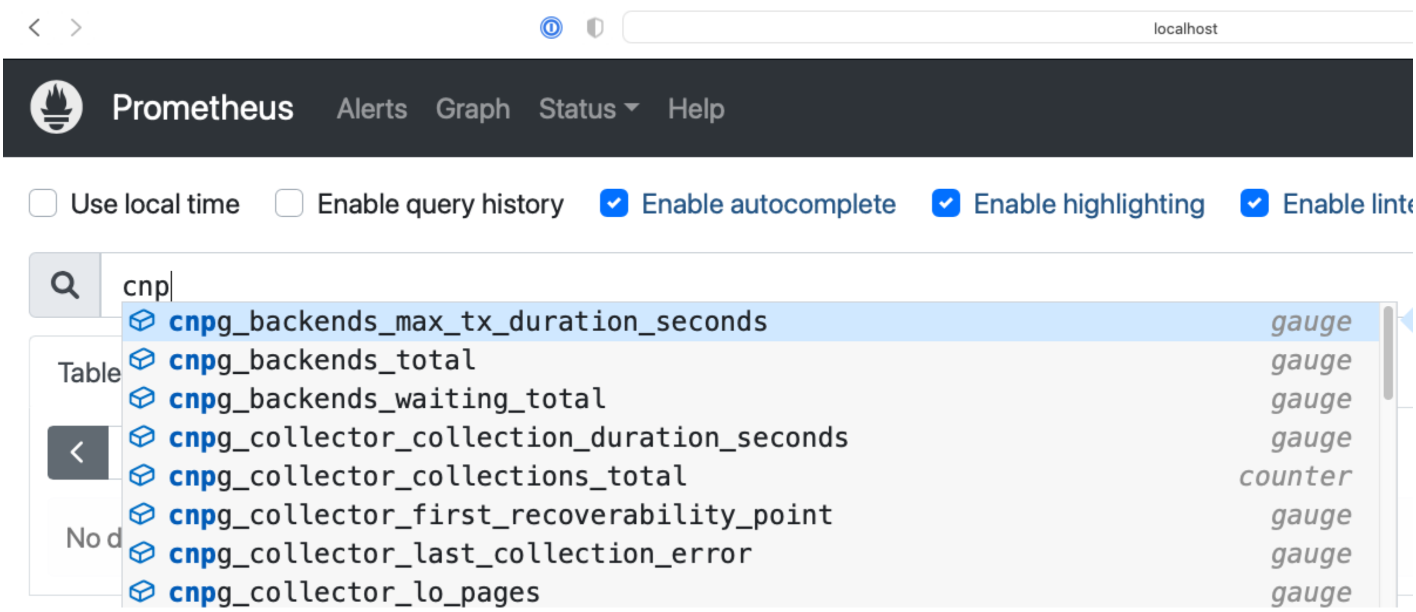

```
kubectl apply -f - <<EOF
---
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-with-
metrics
spec:
  instances:
3


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 EDB Postgres for Kubernetes clusters. Please refer to the monitoring section for more information.



You can also monitor the EDB Postgres for Kubernetes 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/EnterpriseDB/docs/main/product_docs/docs/postgres_for_kubernetes/1/samples/
itoring/prometheusrule.yaml
```

You should see the default alerts now:


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

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

Grafana Dashboard

In our 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`).

EDB Postgres for Kubernetes provides a default dashboard for Grafana in the dedicated `grafana-dashboards` repository. 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 `EDB Postgres for Kubernetes` dashboard just created:

> **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 EDB Postgres for Kubernetes clusters deployed with the Monitoring feature enabled.

8 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 `cluster-example-custom.yaml` .

> **Warning**
>
> **OpenShift users:** due to a current limitation of the OpenShift user interface, it is possible to change PostgreSQL settings from the YAML pane only.

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"
  #
...
```

> **PostgreSQL GUCs: Grand Unified Configuration**
>
> Refer to the PostgreSQL documentation for more information on the available parameters, also known as GUC (Grand Unified Configuration). Please note that EDB Postgres for Kubernetes 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 "Replication" section.)
>
> 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.

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` operator configuration setting, 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.

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" section.

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 EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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.

EDB Postgres for Kubernetes is able to automatically manage the content of the `shared_preload_libraries` option for some of the most used PostgreSQL extensions (see the "Managed extensions" section below for details).

Specifically, as soon as the operator notices that a configuration parameter requires one of the managed libraries, it will automatically add the needed library. The operator will also remove the library as soon as no actual parameter requires it.

> **Important**
>
> Please always keep in mind that removing libraries from `shared_preload_libraries` requires a restart of all instances in the cluster in order to be effective.

You can provide additional `shared_preload_libraries` via `.spec.postgresql.shared_preload_libraries` as a list of strings: the operator will merge them with the ones that it automatically manages.

Managed extensions

As anticipated in the previous section, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes, and some functionalities might be deprecated.

Enabling `auto_explain`

The `auto_explain` extension provides a means for logging execution plans of slow statements automatically, without having to manually run `EXPLAIN` (helpful for tracking down un-optimized queries).

You can enable `auto_explain` by adding to the configuration a parameter that starts with `auto_explain.` as in the following example excerpt (which automatically logs execution plans of queries that take longer than 10 seconds to complete):

```
  #
...
  postgresql:
    parameters:
      auto_explain.log_min_duration: "10s"
  #
...
```

> **Note**
>
> Enabling auto_explain can lead to performance issues. Please refer to `the auto explain documentation`

Enabling `pg_stat_statements`

The `pg_stat_statements` extension is one of the most important capabilities available in PostgreSQL for real-time monitoring of queries.

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

```
  #
...
  postgresql:
    parameters:
      pg_stat_statements.max: "10000"
      pg_stat_statements.track:
all
  #
...
```

As explained previously, the operator will automatically add `pg_stat_statements` to `shared_preload_libraries` and run `CREATE EXTENSION IF NOT EXISTS pg_stat_statements` on each database, enabling you to run queries against the `pg_stat_statements` view.

Enabling `pgaudit`

The `pgaudit` extension provides detailed session and/or object audit logging via the standard PostgreSQL logging facility.

EDB Postgres for Kubernetes has transparent and native support for PGAudit on PostgreSQL clusters. For further information, please refer to the"PGAudit" logs section.

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` 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 EDB Postgres for Kubernetes.

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` for details on this extension.

Additionally, for each database that you intend to you use with `pg_failover_slots` you need to add an entry in the `pg_hba` section that enables each replica to connect to the primary. For example, suppose that you want to use the `app` database with `pg_failover_slots`, you need to add this entry in the `pg_hba` section:

```
  postgresql:
    pg_hba:
      - hostssl app streaming_replica all
cert
```


The `pg_hba` section

`pg_hba` is a list of PostgreSQL Host Based Authentication rules used to create the `pg_hba.conf` used by the pods.

> **Important**
>
> See the PostgreSQL documentation for more information on `pg_hba.conf`.

Since the first matching rule is used for authentication, the `pg_hba.conf` file generated by the operator can be seen as composed of four sections:

1. Fixed rules
2. User-defined rules
3. Optional LDAP section
4. Default rules

Fixed rules:

```
local all all peer

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

Default rules:

```
host all all all <default-authentication-method>
```

From PostgreSQL 14 the default value of the `password_encryption` database parameter is set to `scram-sha-256`. Because of that, the default authentication method is `scram-sha-256` from this PostgreSQL version.

PostgreSQL 13 and older will use `md5` as the default authentication method.

The resulting `pg_hba.conf` will look like this:

```
local all all peer

hostssl postgres streaming_replica all cert
hostssl replication streaming_replica all cert

<user defined rules>
<user defined LDAP>

host all all all scram-sha-256 # (or md5 for PostgreSQL version <= 13)
```

Inside the cluster manifest, `pg_hba` lines are added as list items in `.spec.postgresql.pg_hba`, as in the following excerpt:

```
  postgresql:
    pg_hba:
      - hostssl app app 10.244.0.0/16
md5
```

In the above example we are enabling access for the `app` user to the `app` database using MD5 password authentication (you can use `scram-sha-256` if you prefer) via a secure channel (`hostssl`).

LDAP Configuration

Under the `postgres` section of the cluster spec there is an optional `ldap` section available to define an LDAP configuration to be converted into a rule added into the `pg_hba.conf` file.

This will support two modes: `simple bind` mode which requires specifying a `server`, `prefix` and `suffix` in the LDAP section and the `search+bind` mode which requires specifying `server`, `baseDN`, `binDN`, and a `bindPassword` which is a secret containing the ldap password. Additionally, in `search+bind` mode you have the option to specify a `searchFilter` or `searchAttribute`. If no `searchAttribute` is specified the default one of `uid` will be used.

Additionally, both modes allow the specification of a `scheme` for ldapscheme and a `port`. Neither scheme nor port are required, however.

This section filled out for search+bind could look as follows:

```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'
```

The `pg_ident` section

`pg_ident` is a list of PostgreSQL User Name Maps that EDB Postgres for Kubernetes 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`].

The `pg_ident.conf` file written by the operator is made up of the following two sections:

1. Fixed rules
2. User-defined rules

Currently the only fixed rule, automatically generated by the operator, is:

```
local <postgres system user> postgres
```

The instance manager detects the user running the PostgreSQL instance and automatically adds a rule to map it to the `postgres` user in the database.

If the `postgres` user is not properly configured inside the container, the instance manager will allow any local user to connect and then log a warning message like the following:

```
Unable to identify the current user. Falling back to insecure mapping.
```

The resulting `pg_ident.conf` will look like this:

```
local <postgres system user> postgres

<user defined lines>
```

Inside the cluster manifest, `pg_ident` lines are added as list items in `.spec.postgresql.pg_ident`. For example:

```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`

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes the default configuration disables the use of `ALTER SYSTEM` on new Postgres clusters. This decision is rooted in the recognition of potential risks associated with this command. To enable the use of `ALTER SYSTEM`, you can explicitly set `.spec.postgresql.enableAlterSystem` to `true`.

> **Warning**
>
> Proceed with caution when utilizing `ALTER SYSTEM`. This command operates directly on the connected instance and does not undergo replication. EDB Postgres for Kubernetes 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 — a feature directly contributed by EDB Postgres for Kubernetes 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
```

Dynamic Shared Memory settings

PostgreSQL supports a few implementations for dynamic shared memory management through the `dynamic_shared_memory_type` configuration option. In EDB Postgres for Kubernetes 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 thisthread from the `pgsql-general` mailing list.

POSIX shared memory

The default setting of `posix` should be enough in most cases, considering that the operator automatically mounts a *memory-bound* *`EmptyDir`* *volume* called `shm` under `/dev/shm`. You can verify the size of such volume inside the running Postgres container with:

```
mount | grep
shm
```

You should get something similar to the following output:

```
shm on /dev/shm type tmpfs (rw,nosuid,nodev,noexec,relatime,size=******)
```

If you would like to set a maximum size for the `shm` volume, you can do so by setting the `.spec.ephemeralVolumesSizeLimit.shm` field in the `Cluster` resource. For example:

```
spec:
  ephemeralVolumesSizeLimit:
    shm:
1Gi
```

System V shared memory

In case your Kubernetes cluster has a high enough value for the `SHMMAX` and `SHMALL` parameters, you can also set:

```
dynamic_shared_memory_type: "sysv"
```

You can check the `SHMMAX` / `SHMALL` from inside a PostgreSQL container, by running:

```
ipcs -lm
```

For example:

```
------ Shared Memory Limits --------
max number of segments = 4096
max seg size (kbytes) = 18014398509465599
max total shared memory (kbytes) = 18014398509481980
min seg size (bytes) = 1
```

As you can see, the very high number of `max total shared memory` recommends setting `dynamic_shared_memory_type` to `sysv`.

An alternate method is to run:

```
cat /proc/sys/kernel/shmall
cat /proc/sys/kernel/shmmax
```

Fixed parameters

Some PostgreSQL configuration parameters should be managed exclusively by the operator. The operator prevents the user from setting them using a webhook.

Users are not allowed to set the following configuration parameters in the `postgresql` section:

- `allow_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`
- `edb_audit`
- `edb_audit_destination`
- `edb_audit_directory`
- `edb_audit_filename`
- `edb_audit_rotation_day`
- `edb_audit_rotation_seconds`
- `edb_audit_rotation_size`
- `edb_audit_tag`
- `edb_log_every_bulk_value`
- `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`

9 Operator configuration

The operator for EDB Postgres for Kubernetes 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:

- setting a company license key that is shared by all deployments managed by the operator
- 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 `postgresql-operator-system` namespace as a Kubernetes `Deployment` called `postgresql-operator-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 `postgresql-operator-controller-manager-config` as the name.

> **Important**
>
> Any change to the config's `ConfigMap` / `Secret` will not be automatically detected by the operator, - and as such, it needs to be reloaded (see below). Moreover, changes only apply to the resources created after the configuration is reloaded.

> **Important**
>
> The operator first processes the ConfigMap values and then the Secret's, in this order. As a result, if a parameter is defined in both places, the one in the Secret will be used.

Available options

The operator looks for the following environment variables to be defined in the `ConfigMap` / `Secret`:

Name	Description
`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`
`EDB_LICENSE_KEY`	Default license key (to be used only if the cluster does not define one, and preferably in the `Secret`)
`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`)
`ENABLE_REDWOOD_BY_DEFAULT`	Enable the Redwood compatibility by default when using EPAS.
`EXPIRING_CHECK_THRESHOLD`	Determines the threshold, in days, for identifying a certificate as expiring. Default is 7.

Name	Description
`EXTERNAL_BACKUP_ADDON_CONFIGURATION`	Configuration for the `external-backup-adapter` add-on. (See "Customizing the adapter" in Add-ons)
`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 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 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.

Defining an operator config map

The example below customizes the behavior of the operator, by defining a default license key (namely a company key), 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 instance manager, and by spreading upgrades.

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgresql-operator-controller-manager-
config
  namespace: postgresql-operator-
system
data:
  CLUSTERS_ROLLOUT_DELAY: '60'
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: 'true'
  EDB_LICENSE_KEY:
<YOUR_BASE64_ENCODED_EDB_LICENSE_KEY_HERE>
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload,
app
  INSTANCES_ROLLOUT_DELAY: '10'
```

Defining an operator secret

The example below customizes the behavior of the operator, by defining a default license key (namely a company key), the label/annotation names to be inherited by the resources created by any `Cluster` object that is deployed at a later time, and by enabling in-place updates for the instance manager, and by spreading upgrades.

```yaml
apiVersion: v1
kind:
Secret
metadata:
  name: postgresql-operator-controller-manager-
config
  namespace: postgresql-operator-
system
type:
Opaque
stringData:
  CLUSTERS_ROLLOUT_DELAY: '60'
  ENABLE_INSTANCE_MANAGER_INPLACE_UPDATES: 'true'
  EDB_LICENSE_KEY:
<YOUR_BASE64_ENCODED_EDB_LICENSE_KEY_HERE>
  INHERITED_ANNOTATIONS: categories
  INHERITED_LABELS: environment, workload,
app
  INSTANCES_ROLLOUT_DELAY: '10'
```

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 postgresql-operator-system \
    postgresql-operator-controller-manager
```

Otherwise, If you have installed the operator using OLM, or you are running on Openshift, run the following command specifying the namespace the operator is installed in:

```
kubectl delete pods -n [NAMESPACE_NAME_HERE] \
  -l app.kubernetes.io/name=cloud-native-postgresql
```

> **Warning**
>
> Customizations will be applied only to `Cluster` resources created after the reload of the operator deployment.

Following the above example, if the `Cluster` definition contains a `categories` annotation and any of the `environment`, `workload`, or `app` labels, these will be inherited by all the resources generated by the deployment.

pprof HTTP Server

The operator can expose a PPROF HTTP server with the following endpoints on `localhost:6060`:

- `/debug/pprof/`. Responds to a request for "/debug/pprof/" with an HTML page listing the available profiles
- `/debug/pprof/cmdline`. Responds with the running program's command line, with arguments separated by NULL bytes.
- `/debug/pprof/profile`. Responds with the pprof-formatted cpu profile. Profiling lasts for duration specified in seconds GET parameter, or for 30 seconds if not specified.
- `/debug/pprof/symbol`. Looks up the program counters listed in the request, responding with a table mapping program counters to function names.
- `/debug/pprof/trace`. Responds with the execution trace in binary form. Tracing lasts for duration specified in seconds GET parameter, or for 1 second if not specified.

To enable the operator you need to edit the operator deployment add the flag `--pprof-server=true`.

You can do this by executing these commands:

```
kubectl edit deployment -n postgresql-operator-system postgresql-operator-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=postgresql-operator-controller-manager-
config
        - --secret-name=postgresql-operator-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 postgresql-operator-system <pod name> -- bash
```

Once inside execute:

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

10 Instance pod configuration

Projected volumes

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes, 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.

```
apiVersion: postgresql.k8s.enterprisedb.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 thecluster-example-projected-volume.yaml deployment manifest.

Ephemeral volumes

EDB Postgres for Kubernetes 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.

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.k8s.enterprisedb.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.

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.

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.

EDB Postgres for Kubernetes 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.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  instances: 3

  env:
  - name: TZ
    value:
Australia/Sydney

  storage:
    size:
1Gi
```

The `envFrom` stanza can refer to ConfigMaps or secrets to use their content as environment variables:

```yaml
apiVersion: postgresql.k8s.enterprisedb.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.

11 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.

> **Reference**
>
> For a list of available options, see API reference.

Basics

Basic cluster : `cluster-example.yaml` A basic example of a cluster.

EDB Postgres Advanced Server (EPAS) cluster : `cluster-example-epas.yaml` A basic example of an EPAS cluster.

EDB Postgres Extended (PGE) cluster : `cluster-example-pge.yaml` A basic example of a PGE cluster.

Custom cluster : `cluster-example-custom.yaml` A basic cluster that uses the default storage class and custom parameters for the `postgresql.conf` and `pg_hba.conf` files.

Cluster with customized storage class : `cluster-storage-class.yaml` : A basic cluster that uses a specified storage class of `standard` .

Cluster with persistent volume claim (PVC) template configured : `cluster-pvc-template.yaml` : A basic cluster with an explicit persistent volume claim template.

Extended configuration example : `cluster-example-full.yaml` : A cluster that sets most of the available options.

Bootstrap cluster with SQL files : `cluster-example-initdb-sql-refs.yaml` : 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 : `cluster-example-pg-hba.yaml` : A basic cluster that enables the user app to authenticate using certificates.

Sample cluster with Secret and ConfigMap mounted using projected volume template : `cluster-example-projected-volume.yaml` A basic cluster with the existing `Secret` and `ConfigMap` mounted into Postgres pod using projected volume mount.

Cluster with TDE enabled : `cluster-example-tde.yaml` an EPAS 15 cluster with TDE. Note that you will need access credentials to download the image used.

Backups

Customized storage class and backups : *Prerequisites*: Bucket storage must be available. The sample config is for AWS. Change it to suit your setup. : `cluster-storage-class-with-backup.yaml` A cluster with backups configured.

Backup : *Prerequisites*: `cluster-storage-class-with-backup.yaml` applied and healthy. : `backup-example.yaml` : An example of a backup that runs against the previous sample.

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. : `cluster-example-with-backup.yaml` 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. : `cluster-example-with-backup-scaleway.yaml` A basic cluster with backups configured to work with Scaleway Object Storage..

Replica clusters

Replica cluster by way of backup from an object store : *Prerequisites*: `cluster-storage-class-with-backup.yaml` applied and healthy, and a backup `cluster-example-trigger-backup.yaml` applied and completed. : `cluster-example-replica-from-backup-simple.yaml` : A replica cluster following a cluster with backup configured.

Replica cluster by way of volume snapshot : *Prerequisites*: `cluster-example-with-volume-snapshot.yaml` applied and healthy, and a volume snapshot `backup-with-volume-snapshot.yaml` applied and completed. : `cluster-example-replica-from-volume-snapshot.yaml` : A replica cluster following a cluster with volume snapshot configured.

Replica cluster by way of streaming (pg_basebackup) : *Prerequisites*: `cluster-example.yaml` applied and healthy. : `cluster-example-replica-streaming.yaml` : A replica cluster following `cluster-example` with streaming replication.

PostGIS

PostGIS example : `postgis-example.yaml` : An example of a PostGIS cluster. See PostGIS for details.

Managed roles

Cluster with declarative role management : `cluster-example-with-roles.yaml` : Declares a role with the `managed` stanza. Includes password management with Kubernetes secrets.

Managed services

Cluster with managed services : `cluster-example-managed-services.yaml` : Declares a service with the `managed` stanza. Includes default service disabled and new `rw` service template of `LoadBalancer` type defined.

Declarative tablespaces

Cluster with declarative tablespaces : `cluster-example-with-tablespaces.yaml`

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. : `cluster-example-with-tablespaces-backup.yaml`

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. : `cluster-restore-with-tablespaces.yaml`

For a list of available options, see API reference.

Pooler configuration

Pooler with custom service config : `pooler-external.yaml`

Logical replication via declarative Publication and Subscription objects

Two test manifests contain everything needed to set up logical replication:

Source cluster with a publication : `cluster-example-logical-source.yaml`

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. : `cluster-example-logical-destination.yaml`

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` . : `publication-example.yaml`

A Publication with a constrained publication target : *Prerequisites*: an existing cluster `cluster-example` . : `publication-example-objects.yaml`

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. : `subscription-example.yaml`

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.

Declarative management of Postgres databases

A plain Database : *Prerequisites*: an existing cluster `cluster-example` . : `database-example.yaml`

A Database with ICU local specifications : *Prerequisites*: an existing cluster `cluster-example` running Postgres 16 or more advanced. : `database-example-icu.yaml`

13 Bootstrap

> **Note**
>
> When referring to "PostgreSQL cluster" in this section, the same concepts apply to both PostgreSQL and EDB Postgres Advanced Server, unless differently stated.

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**
>
> EDB Postgres for Kubernetes 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!!!

> **Info**
>
> EDB Postgres for Kubernetes is gradually introducing support for Kubernetes' native `VolumeSnapshot` API for both incremental and differential copy in backup and recovery operations - if supported by the underlying storage classes. Please see "Recovery from Volume Snapshot objects" for details.

The `bootstrap` section

The *bootstrap* method can be defined in the `bootstrap` section of the cluster specification. EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes, although meeting all requirements can be challenging. Be sure to review the warnings in the `pg_basebackup` subsection carefully.

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 EDB Postgres for Kubernetes operator provides for `recovery` , please refer to the dedicated "Recovery" section for guidance on each method.

> **API reference**
>
> Please refer to the "API reference for the `bootstrap` section for more information.

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.

> **Info**
>
> 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.

> **API reference**
>
> Please refer to the "API reference for the `externalClusters` section for more information.

Password files

Whenever a password is supplied within an `externalClusters` entry, EDB Postgres for Kubernetes autonomously manages a PostgreSQL password file for it, residing at `/controller/external/NAME/pgpass` in each instance.

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

Bootstrap an empty cluster (`initdb`)

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

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

```yaml
apiVersion: postgresql.k8s.enterprisedb.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` type. As a result, the `username` in the secret must match the one of the `owner` (for the application secret) and `postgres` for the superuser one.

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

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

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

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

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

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

Passing Options to `initdb`

The PostgreSQL data directory is initialized using the `initdb` PostgreSQL command.

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

> **Warning**
>
> EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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`, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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.k8s.enterprisedb.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**
>
> EDB Postgres for Kubernetes 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.

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, EDB Postgres for Kubernetes 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.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-example-
initdb
spec:
  instances: 3

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

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:


```yaml
apiVersion: postgresql.k8s.enterprisedb.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.

Compatibility Features

EDB Postgres Advanced Server adds many compatibility features to the plain community PostgreSQL. You can find more information about that in theEDB Postgres Advanced Server.

Those features are already enabled during cluster creation on EPAS and are not supported on the community PostgreSQL image. To disable them you can use the `redwood` flag in the `initdb` section like in the following example:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-example-
initdb
spec:
  instances: 3
  imageName: <EPAS-based
image>

  bootstrap:
    initdb:
      database:
app
      owner:
app
      redwood: false
  storage:
    size:
1Gi
```


Bootstrap from another cluster

EDB Postgres for Kubernetes 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` parameter (by default assigned to the value of `name` in the external cluster definition).

Bootstrap from a backup (`recovery`)

Given the variety of backup methods and combinations of backup storage options provided by the EDB Postgres for Kubernetes operator for `recovery`, please refer to the dedicated "Recovery" section for detailed guidance on each method.

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 EDB Postgres for Kubernetes, 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 EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes, unless you are completely certain that all requirements are met and the operation has been thoroughly tested. The EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes.

> **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 EDB Postgres for Kubernetes, 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 EDB Postgres for Kubernetes 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
2. TLS client certificate

Both authentication methods are detailed below.

Requirements

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

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

> **Seealso**
>
> For further information, please refer to the "Planning" section for Warm Standby, the `pg_basebackup` page and the "High Availability, Load Balancing, and Replication" chapter in the PostgreSQL documentation.

About the replication user

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

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

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

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

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

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

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

Username/Password authentication

The first authentication method supported by EDB Postgres for Kubernetes 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.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: target-db
spec:
  instances: 3
  imageName: quay.io/enterprisedb/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).

TLS certificate authentication

The second authentication method supported by EDB Postgres for Kubernetes 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.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-clone-
tls
spec:
  instances: 3
  imageName: quay.io/enterprisedb/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
```

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.


```yaml
apiVersion: postgresql.k8s.enterprisedb.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.

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.

14 Importing Postgres databases

This section describes how to import one or more existing PostgreSQL databases inside a brand new EDB Postgres for Kubernetes 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".

How it works

Conceptually, the import requires you to create a new cluster from scratch (*destination cluster*), using the `initdb` bootstrap method, 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*.

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes project

- **monolith approach**: the destination cluster is designed to host multiple databases and different users, imported from the source cluster

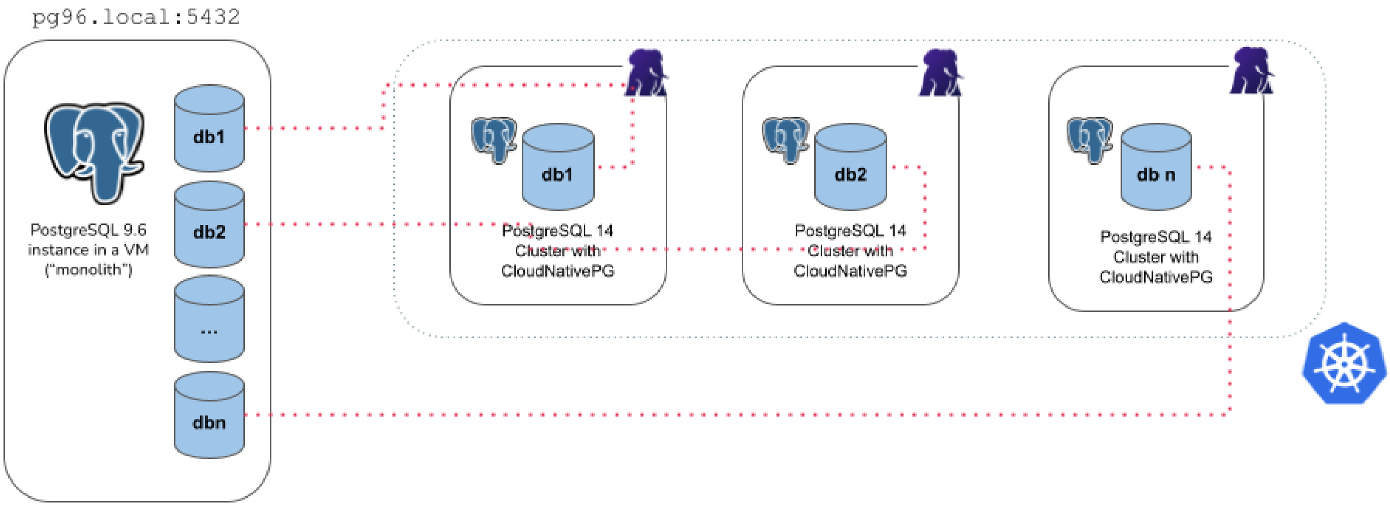
The first import method is available via the `microservice` type, while the latter by the `monolith` type.

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

The `microservice` type

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

- `initdb` bootstrap of the new cluster
- export of the selected database (in `initdb.import.databases`) using `pg_dump -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



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.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
microservice
spec:
  instances: 3

  bootstrap:
    initdb:
      import:
        type: microservice
        databases:
          - angus
        source:
          externalCluster: cluster-
pg96

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

> **Warning**
>
> The example above deliberately uses a source database running a version of PostgreSQL that is not supported anymore by the Community, and consequently by EDB Postgres for Kubernetes. 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.

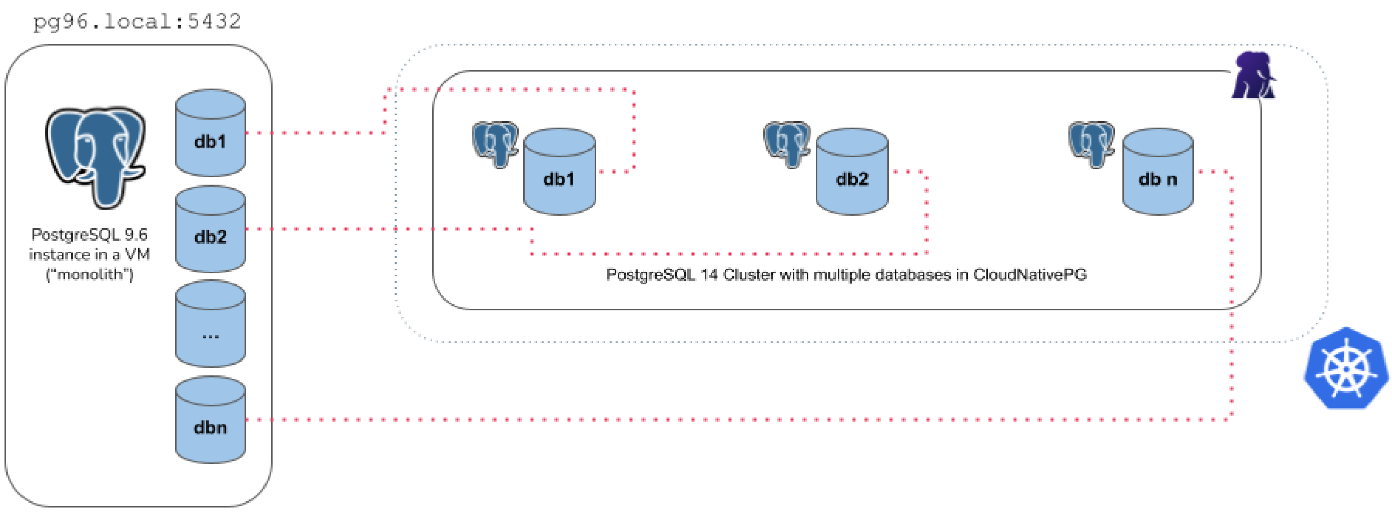
There are a few things you need to be aware of when using the `microservice` type:

- 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")
- 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`

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



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.k8s.enterprisedb.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")
- 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, so there should be enough available space to temporarily contain the dump result on the assigned node, as well as the restored data and indexes. Once the import operation is completed, this folder is automatically deleted by the operator.
- At least one database to be specified in the `initdb.import.databases` array
- Any role that is required by the imported databases must be specified inside `initdb.import.roles`, with the limitations below:
  - The following roles, if present, are not imported: `postgres`, `streaming_replica`, `cnp_pooler_pgbouncer`
  - The `SUPERUSER` option is removed from any imported role

- Wildcard `"*"` can be used as the only element in the `databases` and/or `roles` arrays to import every object of the kind; When matching databases the wildcard will ignore the `postgres` database, template databases, and those databases not allowing connections
- After the clone procedure is done, `ANALYZE VERBOSE` is executed for every database.
- `postImportApplicationSQL` field is not supported

Import optimizations

During the logical import of a database, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes, 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.

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.

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 theLogical Replication section in the documentation.

15 Security

This section contains information about security for EDB Postgres for Kubernetes, 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.

> **About the 4C's Security Model**
>
> 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 EDB Postgres for Kubernetes.

Code

EDB Postgres for Kubernetes' 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 EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes. Every commit must pass all the linters defined by GolangCI-Lint.

Container

Every container image in EDB Postgres for Kubernetes is automatically built via CI/CD pipelines following every commit. These images include not only the operator's image but also the operands' images, specifically for every supported PostgreSQL version (including EDB Postgres Extended and EDB Postgres Advanced). During the CI/CD process, images undergo scanning with 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.

> **Important**
>
> All operand images are automatically rebuilt daily by our pipelines to incorporate security updates at the base image and package level, providing **patch-level updates** for the container images distributed to EDB download sites.

> **Warning**
>
> Running outdated images can expose your environment to security risks and performance issues. We highly recommend that you update to the latest image version and keep your images up to date. This will ensure that you take advantage of the latest updates and patches available.

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).

> **About Container-Level Security**
>
> For more information on the approach that EDB has taken regarding security at the container level in EDB Postgres for Kubernetes, please refer to the blog article "Security and Containers in EDB Postgres for Kubernetes".

Cluster

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

Role Based Access Control (RBAC)

The operator interacts with the Kubernetes API server using a dedicated service account named `postgresql-operator-manager`. This service account is typically installed in the operator namespace, commonly `postgresql-operator-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 `postgresql-operator-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. For OpenShift specificities on this matter, please consult the "Red Hat OpenShift" section, in particular "Pre-defined RBAC objects" section.

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

Below we provide some examples and, most importantly, the reasons why EDB Postgres for Kubernetes 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.

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 EDB Postgres for Kubernetes 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 postgresql-operator-
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.

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

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.

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 therecommended way to access the API server from within a Pod.

For transparency, the permissions associated with the service account are defined in theroles.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 EDB Postgres for Kubernetes.

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

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

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

Pod Security Policies

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

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

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

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

The operator explicitly sets the required security contexts.

On Red Hat OpenShift, Cloud Native PostgreSQL runs in `restricted` security context constraint, the most restrictive one. The goal is to limit the execution of a pod to a namespace allocated UID and SELinux context.

> **Security Context Constraints in OpenShift**
>
> For further information on Security Context Constraints (SCC) in OpenShift, please refer to the"Managing SCC in OpenShift" article.

> **Security Context Constraints and namespaces**
>
> As stated by Openshift documentation SCCs are not applied in the default namespaces (`default` , `kube-system` , `kube-public` , `openshift-node` , `openshift-infra` , `openshift`) and those should not be used to run pods. CNP clusters deployed in those namespaces will be unable to start due to missing SCCs.

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.

> **Example of cluster annotations**
>
> ```
> kind: Cluster
> metadata:
>     name: cluster-apparmor
>     annotations:
>         container.apparmor.security.beta.kubernetes.io/postgres: runtime/default
>         container.apparmor.security.beta.kubernetes.io/initdb: runtime/default
>         container.apparmor.security.beta.kubernetes.io/join: runtime/default
> ```

> **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.

> **AppArmor and OpenShift**
>
> AppArmor is currently available only on Debian distributions like Ubuntu, hence this is not (and will not be) available in OpenShift

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 EDB Postgres for Kubernetes for finer control.

Network policies are beyond the scope of this document. Please refer to the "Network policies" section of the Kubernetes documentation for further information.

Exposed Ports

EDB Postgres for Kubernetes 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

PostgreSQL

The current implementation of EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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" section in the "Connecting from an application" page 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"](ssl_connections.md#"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" page 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" page of the documentation.

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

Storage

EDB Postgres for Kubernetes 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.

16 Postgres instance manager

EDB Postgres for Kubernetes 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.

Startup Probe

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

> **Info**
>
> 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.

EDB Postgres for Kubernetes 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.

> **Info**
>
> 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
```

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, EDB Postgres for Kubernetes 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
```


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.

EDB Postgres for Kubernetes 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.

> **Info**
>
> 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
```

Primary Isolation (alpha)

EDB Postgres for Kubernetes 1.26 introduces an opt-in experimental 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 **disabled by default** and can be enabled by adding the following annotation to the `Cluster` resource:

```
metadata:
  annotations:
    alpha.k8s.enterprisedb.io/livenessPinger: '{"enabled":
true}'
```

> **Warning**
>
> This feature is experimental and will be introduced in a future EDB Postgres for Kubernetes release with a new API. If you decide to use it now, note that the API **will change**.

> **Important**
>
> If you plan to enable this experimental feature, 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 annotation also accepts two optional network settings: `requestTimeout` and `connectionTimeout` , both defaulting to `500` (in milliseconds). In cloud environments, you may need to increase these values. For example:

```
metadata:
  annotations:
    alpha.k8s.enterprisedb.io/livenessPinger: '{"enabled":
true,"requestTimeout":1000,"connectionTimeout":1000}'
```

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.

> **Info**
>
> 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.

EDB Postgres for Kubernetes 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.

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

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, EDB Postgres for Kubernetes 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.
> ```

```
Use the `streaming` and `maximumLag` options with extreme caution. If
you're unfamiliar with PostgreSQL replication, rely on the default
strategy. Seek professional advice if unsure.
```

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:

```yaml
#
<snip>
probes:
  readiness:
    type: streaming
    maximumLag: 64Mi
    failureThreshold: 30
    periodSeconds: 10
```


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 databaseRPO, don't delete the Pod where the primary instance is running. In this case, perform a switchover to another instance first.

Shutdown of the primary during a switchover

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

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

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

Failover

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

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 EDB Postgres for Kubernetes and Kubernetes as well: the "Monitoring" section 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. EDB Postgres for Kubernetes 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" section of the documentation.

17 Scheduling

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

> **Kubernetes documentation**
>
> 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 EDB Postgres for Kubernetes cluster's instances should be scheduled through the `affinity` section in the definition of the cluster, which supports:

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

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, EDB Postgres for Kubernetes configures cluster instances to preferably be scheduled on different nodes, while `pgBouncer` instances might still run on the same nodes.

For example, given the following `Cluster` specification:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  instances: 3
  imageName: quay.io/enterprisedb/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:


```
affinity:
  podAntiAffinity:
    preferredDuringSchedulingIgnoredDuringExecution:
      - podAffinityTerm:
          labelSelector:
            matchExpressions:
              - key: k8s.enterprisedb.io/cluster
                operator: In
                values:
                  - cluster-
example
              - key: k8s.enterprisedb.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.

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.

> **Inter-pod Affinity and Anti-Affinity**
>
> For more details, refer to the Kubernetes documentation.

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.

Disabling Anti-Affinity Policies

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

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:

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

Node selection through `nodeSelector`

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

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

Tolerations

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

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

> **Taints and Tolerations**
>
> More information on taints and tolerations can be found in theKubernetes documentation.

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, EDB Postgres for Kubernetes 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.

18 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.

EDB Postgres for Kubernetes 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:

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

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

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

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

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

For more details, please refer to the"Configure Quality of Service for Pods" section in the Kubernetes documentation.

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

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

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

You can refer to the following example manifest:


```yaml
apiVersion: postgresql.k8s.enterprisedb.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.

> **Managing Compute Resources for Containers**
>
> For more details on resource management, please refer to the"Managing Compute Resources for Containers" page from the Kubernetes documentation.

19 Failure Modes

> **Note**
>
> In previous versions of EDB Postgres for Kubernetes, 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 EDB Postgres for Kubernetes.

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes-specific details, refer to the "Postgres Instance Manager" section, which covers startup, liveness, and readiness probes, as well as the self-healing section below.

> **Important**
>
> If you are running EDB Postgres for Kubernetes in production, we strongly recommend seeking professional support.

Self-Healing

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.

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.

Manual Intervention

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

> **Important**
>
> Do not perform manual operations without professional support.

Disabling Reconciliation

To temporarily disable the reconciliation loop for a PostgreSQL cluster, use the `k8s.enterprisedb.io/reconciliationLoop` annotation:

```yaml
metadata:
  name: cluster-example-no-
reconcile
  annotations:
    k8s.enterprisedb.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.

20 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 (unlessin-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 EDB Postgres for Kubernetes identity, without re-cloning the data. Pods will be deleted and created again with the same PVCs and a new image, if required.

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

Automated updates (`unsupervised`)

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

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

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

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

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

Manual updates (`supervised`)

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

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

You can trigger a switchover with:

```
kubectl cnp promote [cluster]
[new_primary]
```

You can trigger a restart with:

```
kubectl cnp restart [cluster]
[current_primary]
```

You can find more information in the `cnp` plugin page.

21 Replication

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

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

Application-level replication

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

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

A very mature technology

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

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

PostgreSQL 9.0 (2010) introduced WAL streaming and read-only replicas through *hot standby*. In 2011, PostgreSQL 9.1 brought synchronous replication at the transaction level, supporting RPO=0 clusters. Cascading replication was added in PostgreSQL 9.2 (2012). The foundations for logical replication 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.

Streaming replication support

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

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


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

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

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

```
# Require client certificate authentication for the streaming_replica user
hostssl postgres streaming_replica all cert
hostssl replication streaming_replica all cert
```

> **Certificates**
>
> For details on how EDB Postgres for Kubernetes manages certificates, please refer to the "Certificates" section 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.

> **Replication slots for High Availability**
>
> For details on how EDB Postgres for Kubernetes automatically manages replication slots for the High Availability replicas, please refer to the "Replication slots for High Availability" section below.

Continuous backup integration

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

Synchronous Replication

EDB Postgres for Kubernetes 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.

Direct configuration of the `synchronous_standby_names` option is not permitted. However, EDB Postgres for Kubernetes 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` stanza.

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

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 EDB Postgres for Kubernetes cluster.

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, EDB Postgres for Kubernetes automatically sets the content of `synchronous_standby_names` as follows:

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

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 EDB Postgres for Kubernetes.

Given the following manifest:

```
apiVersion: postgresql.k8s.enterprisedb.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.k8s.enterprisedb.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.

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.

Controlling `synchronous_standby_names` Content

By default, EDB Postgres for Kubernetes 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`. EDB Postgres for Kubernetes expects that you manage any standby with an `application_name` listed here, ensuring their high availability. Incorrect entries can jeopardize your PostgreSQL database uptime.

Examples

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

If you set:

```
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:

```
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)
```


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.

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.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name:
foo
spec:
  instances: 3
  postgresql:
    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.

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, EDB Postgres for Kubernetes allows you to configure startup and readiness probes for replicas based on maximum lag. For more details, please refer to the "Postgres instance manager" section.

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:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  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.

Synchronous Replication (Deprecated)

> **Warning**
>
> Prior to EDB Postgres for Kubernetes 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.

EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes automatically updates the `synchronous_standby_names` option in PostgreSQL to the following value:

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

Where:

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

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

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

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

Select nodes for synchronous replication

EDB Postgres for Kubernetes 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.

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

> **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 (RPO).

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

> **Important**
>
> The self-healing enforcement still applies while defining additional constraints for synchronous replica election (see "Synchronous replication").

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

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

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

Replication slots

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

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

EDB Postgres for Kubernetes 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" below for details)
- user-defined replication slots created on the primary

Replication slots for High Availability

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes, 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"replicationSlots" section in the API reference. 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: `_cnp_`)

`.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.


```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  instances: 3
  # Disable replication slots for HA in the
cluster
  replicationSlots:
    highAvailability:
      enabled: false

  storage:
    size:
1Gi
```

User-Defined Replication slots

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

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

EDB Postgres for Kubernetes 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.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  instances: 3
  replicationSlots:
    synchronizeReplicas:
      enabled: true
      excludePatterns:
      - "^foo"
```

For details, please refer to the "replicationSlots" section in the API reference. Here follows a brief description of the main options:

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

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

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

Synchronization frequency

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

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  instances: 3
  # Reduce the frequency of standby HA slots updates to once every 5
minutes
  replicationSlots:
    updateInterval: 300

  storage:
    size:
1Gi
```

Capping the WAL size retained for replication slots

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

Starting with PostgreSQL 13, you can take advantage of the `max_slot_wal_keep_size` configuration option controlling the maximum size of WAL files that replication slots are allowed to retain in the `pg_wal` directory at checkpoint time. By default, in PostgreSQL `max_slot_wal_keep_size` is set to `-1`, meaning that replication slots may retain an unlimited amount of WAL files. As a result, our recommendation is to explicitly set `max_slot_wal_keep_size` when replication slots support is enabled. For example:

```
  #
...

postgresql:

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

Monitoring replication slots

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

> **Monitoring**
>
> Please refer to the "Monitoring" section for details on how to monitor a EDB Postgres for Kubernetes deployment.

22 Backup

> **Info**
>
> 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 EDB Postgres for Kubernetes. 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 CNP-I plugins. This transition aligns with EDB Postgres for Kubernetes' shift towards a **backup-agnostic architecture**, enabled by its extensible interface—**CNP-I**—which standardizes the management of **WAL archiving**, **physical base backups**, and corresponding **recovery processes**.

EDB Postgres for Kubernetes currently supports **physical backups of PostgreSQL clusters** in two main ways:

- **Via CNPG-I plugins**: the EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes, 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.

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 EDB Postgres for Kubernetes, 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)

CNP-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.

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, EDB Postgres for Kubernetes provides out-of-the-box an RPO <= 5 minutes for disaster recovery, even across regions.

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

Cold and Hot backups

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

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

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

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

Comparing Available Backup Options: Object Stores vs Volume Snapshots

EDB Postgres for Kubernetes 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**
>
> CNP-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.

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

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

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.

Comparison Summary

Feature	Object Store	Volume Snapshots
WAL archiving	Required	Recommended^1^
Cold backup		
Hot backup		
Incremental copy		^2^
Differential copy		^2^
Backup from a standby		
Snapshot recovery	^3^	
Retention policies		
Point-in-Time Recovery (PITR)		Requires WAL archive
Underlying technology	Barman Cloud	Kubernetes API

> **Notes:**
>
> 1. WAL archiving must currently use an object store through a plugin (or the deprecated native one).
> 2. Availability of incremental and differential copies depends on the capabilities of the storage class used for PostgreSQL volumes.
> 3. Snapshot recovery can be emulated by using the `bootstrap.recovery.recoveryTarget.targetImmediate` option.

Scheduled Backups

Scheduled backups are the recommended way to implement a reliable backup strategy in EDB Postgres for Kubernetes. They are defined using the `ScheduledBackup` custom resource.

> **Info**
>
> For a complete list of configuration options, refer to the `ScheduledBackupSpec` in the API reference.

Cron Schedule

The `schedule` field defines **when** the backup should occur, using a *six-field cron expression* that includes seconds. This format follows the Go `cron` package specification.

> **Warning**
>
> This format differs from the traditional Unix/Linux `crontab` —it includes a **seconds** field as the first entry.

Example of a daily scheduled backup:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: ScheduledBackup
metadata:
  name: backup-example
spec:
  schedule: "0 0 0 * * *"  # At midnight every
day
  backupOwnerReference: self
  cluster:
    name: pg-
backup
  # method: plugin, volumeSnapshot, or barmanObjectStore
(default)
```

The schedule `"0 0 0 * * *"` triggers a backup every day at midnight (00:00:00). In Kubernetes CronJobs, the equivalent expression would be `0 0 * * *`, since seconds are not supported.

Backup Frequency and RTO

> **Hint**
>
> The frequency of your backups directly impacts your **Recovery Time Objective** (RTO).

To optimize your disaster recovery strategy based on continuous backup:

- Regularly test restoring from your backups.
- Measure the time required for a full recovery.
- Account for the size of base backups and the number of WAL files that must be retrieved and replayed.

In most cases, a **weekly base backup** is sufficient. It is rare to schedule full backups more frequently than once per day.

Immediate Backup

To trigger a backup immediately when the `ScheduledBackup` is created:

```
spec:
  immediate: true
```

Pause Scheduled Backups

To temporarily stop scheduled backups from running:

```
spec:
  suspend: true
```

Backup Owner Reference (`.spec.backupOwnerReference`)

Controls which Kubernetes object is set as the owner of the backup resource:

- `none` : No owner reference (legacy behavior)
- `self` : The `ScheduledBackup` object becomes the owner
- `cluster` : The PostgreSQL cluster becomes the owner

On-Demand Backups

On-demand backups allow you to manually trigger a backup operation at any time by creating a `Backup` resource.

> **Info**
>
> For a full list of available options, see the `BackupSpec` in the API reference.

Example: Requesting an On-Demand Backup

To start an on-demand backup, apply a `Backup` request custom resource like the following:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind:
Backup
metadata:
  name: backup-example
spec:
  method: barmanObjectStore
  cluster:
    name: pg-
backup
```

In this example, the operator will orchestrate the backup process using the `barman-cloud-backup` tool and store the backup in the configured object store.

Monitoring Backup Progress

You can check the status of the backup using:

```
kubectl describe backup backup-
example
```

While the backup is in progress, you'll see output similar to:

```
Name:          backup-example
Namespace:     default
...
Spec:
  Cluster:
    Name:  pg-backup
Status:
  Phase:       running
  Started At:  2020-10-26T13:57:40Z
Events:        <none>
```

Once the backup has successfully completed, the `phase` will be set to `completed` , and the output will include additional metadata:

```
Name:         backup-example
Namespace:    default
...
Status:
  Backup Id:         20201026T135740
  Destination Path:  s3://backups/
  Endpoint URL:      http://minio:9000
  Phase:             completed
  S3 Credentials:
    Access Key Id:
      Name:  minio
      Key:   ACCESS_KEY_ID
    Secret Access Key:
      Name:  minio
      Key:   ACCESS_SECRET_KEY
  Server Name:       pg-backup
  Started At:        2020-10-26T13:57:40Z
  Stopped At:        2020-10-26T13:57:44Z
```

> **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.

Backup Methods

EDB Postgres for Kubernetes currently supports the following backup methods for scheduled and on-demand backups:

- `plugin` – Uses a CNP-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, 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" section of the plugin documentation

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, **EDB Postgres for Kubernetes 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.

How It Works

When `prefer-standby` is the target (the default behavior), EDB Postgres for Kubernetes 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.

Forcing Backup on the Primary

To always run backups on the primary instance, explicitly set the backup target to `primary` in the cluster configuration:

```yaml
apiVersion: postgresql.k8s.enterprisedb.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.

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.k8s.enterprisedb.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.

Retention Policies

EDB Postgres for Kubernetes is evolving toward a **backup-agnostic architecture**, where backup responsibilities are delegated to external **CNP-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 EDB Postgres for Kubernetes.

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" with Barman Cloud Plugin.

> **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.

23 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 EDB Postgres for Kubernetes, 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.

EDB Postgres for Kubernetes supports recovery via:

- A **pluggable backup and recovery interface (CNP-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.

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 EDB Postgres for Kubernetes `Cluster` —either using the **deprecated native Barman Cloud integration** or the **Barman Cloud Plugin**.

> **Info**
>
> 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:


```yaml
apiVersion: barmancloud.k8s.enterprisedb.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:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-restore
spec:
  [...]

  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
```

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.

EDB Postgres for Kubernetes 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 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:

```yaml
apiVersion: postgresql.k8s.enterprisedb.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

  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:

```yaml
apiVersion: postgresql.k8s.enterprisedb.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.

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:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-example-initdb
spec:
  instances: 3

  bootstrap:
    recovery:
      backup:
        name: backup-example

  storage:
    size: 1Gi
```

This bootstrap method allows you to specify just a reference to the backup that needs to be restored.

The previous example 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".

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.

Point in time recovery (PITR)

Instead of replaying all the WALs up to the latest one, after extracting a base backup, you can ask PostgreSQL to stop replaying WALs at any given point in time. PostgreSQL uses this technique to achieve PITR. The presence of a WAL archive is mandatory.

> **Important**
>
> PITR requires you to specify a recovery target by using the options described in Recovery targets.

The operator generates the configuration parameters required for this feature to work if you specify a recovery target.

PITR from an object store

This example uses 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.k8s.enterprisedb.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.

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.

```
apiVersion: postgresql.k8s.enterprisedb.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 EDB Postgres for Kubernetes 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.

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.k8s.enterprisedb.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.k8s.enterprisedb.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
```

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 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.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  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.

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.

> **Important**
>
> If you're not familiar with how PostgreSQL PITR works, we suggest that you configure the recovery cluster as the original one when it comes to `.spec.postgresql.parameters`. Once the new cluster is restored, you can then change the settings as desired.

The way it works is that the operator injects an init container in the first instance of the new cluster, and the init container starts recovering the backup from the object storage.

> **Important**
>
> The duration of the base backup copy in the new PVC depends on the size of the backup, as well as the speed of both the network and the storage.

When the base backup recovery process is complete, the operator starts the Postgres instance in recovery mode. In this phase, PostgreSQL is up, though not able to accept connections, and the pod is healthy according to the liveness probe. By way of the `restore_command`, PostgreSQL starts fetching WAL files from the archive. You can speed up this phase by setting the `maxParallel` option and enabling the parallel WAL restore capability.

This phase terminates when PostgreSQL reaches the target, either the end of the WAL or the required target in case of PITR. You can optionally specify a `recoveryTarget` to perform a PITR. If left unspecified, the recovery continues up to the latest available WAL on the default target timeline (`latest`).

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

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

Restoring into a Cluster with a Backup Section

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**
>
> EDB Postgres for Kubernetes 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 `k8s.enterprisedb.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.

24 Appendix A - Backup on volume snapshots

> **Important**
>
> Please refer to the official Kubernetes documentation for a list of all the supported Container Storage Interface (CSI) drivers that provide snapshotting capabilities.

EDB Postgres for Kubernetes is one of the first known cases of database operators that directly leverages the Kubernetes native Volume Snapshot API for both backup and recovery operations, in an entirely declarative way.

About standard Volume Snapshots

Volume snapshotting was first introduced in Kubernetes 1.12 (2018) as alpha, promoted to beta in 1.17 (2019), and moved to GA in 1.20 (2020). It's now stable, widely available, and standard, providing 3 custom resource definitions: `VolumeSnapshot`, `VolumeSnapshotContent` and `VolumeSnapshotClass`.

This Kubernetes feature defines a generic interface for:

- the creation of a new volume snapshot, starting from a PVC
- the deletion of an existing snapshot
- the creation of a new volume from a snapshot

Kubernetes delegates the actual implementation to the underlying CSI drivers (not all of them support volume snapshots). Normally, storage classes that provide volume snapshotting support **incremental and differential block level backup in a transparent way for the application**, which can delegate the complexity and the independent management down the stack, including cross-cluster availability of the snapshots.

Requirements

For Volume Snapshots to work with a EDB Postgres for Kubernetes cluster, you need to ensure that each storage class used to dynamically provision the PostgreSQL volumes (namely, `storage` and `walStorage` sections) support volume snapshots.

Given that instructions vary from storage class to storage class, please refer to the documentation of the specific storage class and related CSI drivers you have deployed in your Kubernetes system.

Normally, it is the `VolumeSnapshotClass` that is responsible to ensure that snapshots can be taken from persistent volumes of a given storage class, and managed as `VolumeSnapshot` and `VolumeSnapshotContent` resources.

> **Important**
>
> It is your responsibility to verify with the third party vendor that volume snapshots are supported. EDB Postgres for Kubernetes only interacts with the Kubernetes API on this matter, and we cannot support issues at the storage level for each specific CSI driver.

How to configure Volume Snapshot backups

EDB Postgres for Kubernetes allows you to configure a given Postgres cluster for Volume Snapshot backups through the `backup.volumeSnapshot` stanza.

> **Info**
>
> Please refer to `VolumeSnapshotConfiguration` in the API reference for a full list of options.

A generic example with volume snapshots (assuming that PGDATA and WALs share the same storage class) is the following:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: snapshot-cluster
spec:
  instances: 3

  storage:
    storageClass: @STORAGE_CLASS@
    size: 10Gi
  walStorage:
    storageClass: @STORAGE_CLASS@
    size: 10Gi

  backup:
    # Volume snapshot
backups
    volumeSnapshot:
       className: @VOLUME_SNAPSHOT_CLASS_NAME@

  plugins:
  - name: barman-cloud.cloudnative-pg.io
    isWALArchiver: true
    parameters:
      barmanObjectName: @OBJECTSTORE_NAME@
```

As you can see, the `backup` section contains both the `volumeSnapshot` stanza (controlling physical base backups on volume snapshots) and the `plugins` one (controlling the WAL archive).

> **Info**
>
> Once you have defined the `plugin`, you can decide to use both volume snapshot and plugin backup strategies simultaneously to take physical backups.

The `volumeSnapshot.className` option allows you to reference the default `VolumeSnapshotClass` object used for all the storage volumes you have defined in your PostgreSQL cluster.

> **Info**
>
> In case you are using a different storage class for `PGDATA` and WAL files, you can specify a separate `VolumeSnapshotClass` for that volume through the `walClassName` option (which defaults to the same value as `className`).

Once a cluster is defined for volume snapshot backups, you need to define a `ScheduledBackup` resource that requests such backups on a periodic basis.

Hot and cold backups

> **Warning**
>
> As noted in the backup document, a cold snapshot explicitly set to target the primary will result in the primary being fenced for the duration of the backup, making the cluster read-only during this period. For safety, in a cluster already containing fenced instances, a cold snapshot is rejected.

By default, EDB Postgres for Kubernetes requests an online/hot backup on volume snapshots, using the PostgreSQL defaults of the low-level API for base backups:

- it doesn't request an immediate checkpoint when starting the backup procedure
- it waits for the WAL archiver to archive the last segment of the backup when terminating the backup procedure

> **Important**
>
> The default values are suitable for most production environments. Hot backups are consistent and can be used to perform snapshot recovery, as we ensure WAL retention from the start of the backup through a temporary replication slot. However, our recommendation is to rely on cold backups for that purpose.

You can explicitly change the default behavior through the following options in the `.spec.backup.volumeSnapshot` stanza of the `Cluster` resource:

- `online` : accepting `true` (default) or `false` as a value
- `onlineConfiguration.immediateCheckpoint` : whether you want to request an immediate checkpoint before you start the backup procedure or not; technically, it corresponds to the `fast` argument you pass to the `pg_backup_start` / `pg_start_backup()` function in PostgreSQL, accepting `true` (default) or `false`
- `onlineConfiguration.waitForArchive` : whether you want to wait for the archiver to process the last segment of the backup or not; technically, it corresponds to the `wait_for_archive` argument you pass to the `pg_backup_stop` / `pg_stop_backup()` function in PostgreSQL, accepting `true` (default) or `false`

If you want to change the default behavior of your Postgres cluster to take cold backups by default, all you need to do is add the `online: false` option to your manifest, as follows:

```
  #
...
  backup:
    volumeSnapshot:
       online: false
       #
...
```

If you are instead requesting an immediate checkpoint as the default behavior, you can add this section:

```
  #
...
  backup:
    volumeSnapshot:
       online: true
       onlineConfiguration:
         immediateCheckpoint: true
       #
...
```

Overriding the default behavior

You can change the default behavior defined in the cluster resource by setting different values for `online` and, if needed, `onlineConfiguration` in the `Backup` or `ScheduledBackup` objects.

For example, in case you want to issue an on-demand cold backup, you can create a `Backup` object with `.spec.online: false` :


```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind:
Backup
metadata:
  name: snapshot-cluster-cold-backup-example
spec:
  cluster:
    name: snapshot-cluster
  method:
volumeSnapshot
  online: false
```

Similarly, for the ScheduledBackup:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: ScheduledBackup
metadata:
  name: snapshot-cluster-cold-backup-example
spec:
  schedule: "0 0 0 * *
*"
  backupOwnerReference: self
  cluster:
    name: snapshot-cluster
  method:
volumeSnapshot
  online: false
```

Persistence of volume snapshot objects

By default, `VolumeSnapshot` objects created by EDB Postgres for Kubernetes are retained after deleting the `Backup` object that originated them, or the `Cluster` they refer to. Such behavior is controlled by the `.spec.backup.volumeSnapshot.snapshotOwnerReference` option which accepts the following values:

- `none` : no ownership is set, meaning that `VolumeSnapshot` objects persist after the `Backup` and/or the `Cluster` resources are removed
- `backup` : the `VolumeSnapshot` object is owned by the `Backup` resource that originated it, and when the backup object is removed, the volume snapshot is also removed
- `cluster` : the `VolumeSnapshot` object is owned by the `Cluster` resource that is backed up, and when the Postgres cluster is removed, the volume snapshot is also removed

In case a `VolumeSnapshot` is deleted, the `deletionPolicy` specified in the `VolumeSnapshotContent` is evaluated:

- if set to `Retain` , the `VolumeSnapshotContent` object is kept
- if set to `Delete` , the `VolumeSnapshotContent` object is removed as well

> **Warning**
>
> `VolumeSnapshotContent` objects do not keep all the information regarding the backup and the cluster they refer to (like the annotations and labels that are contained in the `VolumeSnapshot` object). Although possible, restoring from just this kind of object might not be straightforward. For this reason, our recommendation is to always backup the `VolumeSnapshot` definitions, even using a Kubernetes level data protection solution.

The value in `VolumeSnapshotContent` is determined by the `deletionPolicy` set in the corresponding `VolumeSnapshotClass` definition, which is referenced in the `.spec.backup.volumeSnapshot.className` option.

Please refer to the Kubernetes documentation on Volume Snapshot Classes for details on this standard behavior.

Backup Volume Snapshot Deadlines

EDB Postgres for Kubernetes supports backups using the volume snapshot method. In some environments, volume snapshots may encounter temporary issues that can be retried.

The `backup.k8s.enterprisedb.io/volumeSnapshotDeadline` annotation defines how long EDB Postgres for Kubernetes should continue retrying recoverable errors before marking the backup as failed.

You can add the `backup.k8s.enterprisedb.io/volumeSnapshotDeadline` annotation to both `Backup` and `ScheduledBackup` resources. For `ScheduledBackup` resources, this annotation is automatically inherited by any `Backup` resources created from the schedule.

If not specified, the default retry deadline is **10 minutes**.

Error Handling

When a retryable error occurs during a volume snapshot operation:

1. EDB Postgres for Kubernetes records the time of the first error.
2. The system retries the operation every **10 seconds**.
3. If the error persists beyond the specified deadline (or the default 10 minutes), the backup is marked as **failed**.

Retryable Errors

EDB Postgres for Kubernetes treats the following types of errors as retryable:

- **Server timeout errors** (HTTP 408, 429, 500, 502, 503, 504)
- **Conflicts** (optimistic locking errors)
- **Internal errors**
- **Context deadline exceeded errors**
- **Timeout errors from the CSI snapshot controller**

Examples

You can add the annotation to a `ScheduledBackup` resource as follows:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: ScheduledBackup
metadata:
  name: daily-backup-schedule
  annotations:
    backup.k8s.enterprisedb.io/volumeSnapshotDeadline: "20"
spec:
  schedule: "0 0 * * *"
  backupOwnerReference: self
  method: volumeSnapshot
  # other configuration...
```

When you define a `ScheduledBackup` with the annotation, any `Backup` resources created from this schedule automatically inherit the specified timeout value.

In the following example, all backups created from the schedule will have a 30-minute timeout for retrying recoverable snapshot errors.

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: ScheduledBackup
metadata:
  name: weekly-backup
  annotations:
    backup.k8s.enterprisedb.io/volumeSnapshotDeadline: "30"
spec:
  schedule: "0 0 * * 0"  # Weekly backup on
Sunday
  method:
volumeSnapshot
  cluster:
    name: my-postgresql-
cluster
```

Alternatively, you can add the annotation directly to a `Backup` Resource:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind:
Backup
metadata:
  name: my-
backup
  annotations:
    backup.k8s.enterprisedb.io/volumeSnapshotDeadline: "15"
spec:
  method:
volumeSnapshot
  # other backup
configuration...
```

Example of Volume Snapshot Backup

The following example shows how to configure volume snapshot base backups on an EKS cluster on AWS using the `ebs-sc` storage class and the `csi-aws-vsc` volume snapshot class.

> **Important**
>
> If you are interested in testing the example, please read"Volume Snapshots" for the Amazon Elastic Block Store (EBS) CSI driver for detailed instructions on the installation process for the storage class and the snapshot class.

The following manifest creates a `Cluster` that is ready to be used for volume snapshots and that stores the WAL archive in a S3 bucket via IAM role for the Service Account (IRSA, see AWS S3):

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: hendrix
spec:
  instances: 3

  storage:
    storageClass: ebs-sc
    size: 10Gi
  walStorage:
    storageClass: ebs-sc
    size: 10Gi

  backup:
    volumeSnapshot:
       className: csi-aws-vsc

  plugins:
  - name: barman-cloud.cloudnative-pg.io
    isWALArchiver: true
    parameters:
      barmanObjectName: @OBJECTSTORE_NAME@

  serviceAccountTemplate:
    metadata:
      annotations:
        eks.amazonaws.com/role-arn: "@ARN@"
---
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: ScheduledBackup
metadata:
  name: hendrix-vs-
backup
spec:
  cluster:
    name: hendrix
  method:
volumeSnapshot
  schedule: '0 0 0 * *
*'
  backupOwnerReference: cluster
  immediate: true
```

The last resource defines daily volume snapshot backups at midnight, requesting one immediately after the cluster is created.

25 Appendix B - Backup on object stores

> **Warning**
>
> As of EDB Postgres for Kubernetes 1.26, **native Barman Cloud support is deprecated** in favor of the **Barman Cloud Plugin**. This page has been moved to the appendix for reference purposes. While the native integration remains functional for now, we strongly recommend beginning a gradual migration to the plugin-based interface after appropriate testing. For guidance, see Migrating from Built-in EDB Postgres for Kubernetes Backup.

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

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

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

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

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

Common object stores

If you are looking for a specific object store such as AWS S3, Microsoft Azure Blob Storage, Google Cloud Storage, or a compatible provider, please refer to Appendix C - Common object stores for backups.

WAL archiving

WAL archiving is the process that feeds a WAL archive in EDB Postgres for Kubernetes.

The WAL archive is defined in the `.spec.backup.barmanObjectStore` stanza of a `Cluster` resource.

> **Info**
>
> Please refer to `BarmanObjectStoreConfiguration` in the barman-cloud API for a full list of options.

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


```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      wal:
        compression: gzip
        encryption:
AES256
```

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

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

> **Important**
>
> By default, EDB Postgres for Kubernetes sets `archive_timeout` to `5min`, ensuring that WAL files, even in case of low workloads, are closed and archived at least every 5 minutes, providing a deterministic time-based value for your Recovery Point Objective (RPO). Even though you change the value of the `archive_timeout` setting in the PostgreSQL configuration, our experience suggests that the default value set by the operator is suitable for most use cases.

When the bandwidth between the PostgreSQL instance and the object store allows archiving more than one WAL file in parallel, you can use the parallel WAL archiving feature of the instance manager like in the following example:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      wal:
        compression: gzip
        maxParallel: 8
        encryption:
AES256
```

In the previous example, the instance manager optimizes the WAL archiving process by archiving in parallel at most eight ready WALs, including the one requested by PostgreSQL.

When PostgreSQL will request the archiving of a WAL that has already been archived by the instance manager as an optimization, that archival request will be just dismissed with a positive status.

Retention policies

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

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

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

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path
here>"
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
    retentionPolicy: "30d"
```

> **There's more ...**
>
> The **recovery window retention policy** is focused on the concept of *Point of Recoverability* (`PoR`), a moving point in time determined by `current time - recovery window` . The *first valid backup* is the first available backup before `PoR` (in reverse chronological order). EDB Postgres for Kubernetes must ensure that we can recover the cluster at any point in time between `PoR` and the latest successfully archived WAL file, starting from the first valid backup. Base backups that are older than the first valid backup will be marked as *obsolete* and permanently removed after the next backup is completed.

Compression algorithms

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

- bzip2
- gzip
- lz4
- snappy
- xz
- zstd

The compression settings for backups and WALs are independent. See the DataBackupConfiguration and WALBackupConfiguration sections in the barman-cloud API reference.

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

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

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

Tagging of backup objects

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

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

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

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

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      tags:
        backupRetentionPolicy: "expire"
      historyTags:
        backupRetentionPolicy: "keep"
```

Extra options for the backup and WAL commands

You can append additional options to the `barman-cloud-backup` and `barman-cloud-wal-archive` commands by using the `additionalCommandArgs` property in the `.spec.backup.barmanObjectStore.data` and `.spec.backup.barmanObjectStore.wal` sections respectively. These properties are lists of strings that will be appended to the `barman-cloud-backup` and `barman-cloud-wal-archive` commands.

For example, you can use the `--read-timeout=60` to customize the connection reading timeout.

For additional options supported by `barman-cloud-backup` and `barman-cloud-wal-archive` commands you can refer to the official barman documentation here.

If an option provided in `additionalCommandArgs` is already present in the declared options in its section (`.spec.backup.barmanObjectStore.data` or `.spec.backup.barmanObjectStore.wal`), the extra option will be ignored.

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

For backups:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      data:
        additionalCommandArgs:
        - "--min-chunk-size=5MB"
        - "--read-timeout=60"
```

For WAL files:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      [...]
      wal:
        additionalCommandArgs:
        - "--max-concurrency=1"
        - "--read-timeout=60"
```

Recovery from an object store

You can recover from a backup created by Barman Cloud and stored on a supported object store. After you define the external cluster, including all the required configuration in the `barmanObjectStore` section, you need to reference it in the `.spec.recovery.source` option.

This example defines a recovery object store in a blob container in Azure:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
restore
spec:
  [...]

  superuserSecret:
    name: superuser-secret

  bootstrap:
    recovery:
      source: clusterBackup

  externalClusters:
    - name: clusterBackup
      barmanObjectStore:
        destinationPath: https://STORAGEACCOUNTNAME.blob.core.windows.net/CONTAINERNAME/
        azureCredentials:
          storageAccount:
            name: recovery-object-store-secret
            key: storage_account_name
          storageKey:
            name: recovery-object-store-secret
            key:
storage_account_key
        wal:
          maxParallel: 8
```

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".

> **Important**
>
> By default, the `recovery` method strictly uses the `name` of the cluster in the `externalClusters` section as the name of the main folder of the backup data within the object store. This name is normally reserved for the name of the server. You can specify a different folder name using the `barmanObjectStore.serverName` property.

> **Note**
>
> This example takes advantage of the parallel WAL restore feature, dedicating up to 8 jobs to concurrently fetch the required WAL files from the archive. This feature can appreciably reduce the recovery time. Make sure that you plan ahead for this scenario and correctly tune the value of this parameter for your environment. It will make a difference when you need it, and you will.

26 WAL archiving

Write-Ahead Log (WAL) archiving in EDB Postgres for Kubernetes 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.

Plugin-Based Architecture

EDB Postgres for Kubernetes 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.

Supported Plugins

Currently, the **Barman Cloud Plugin** is the only officially supported WAL archiving plugin maintained by the EDB Postgres for Kubernetes Community. For full documentation, configuration options, and best practices, see the Barman Cloud Plugin documentation.

Deprecation Notice: Native Barman Cloud

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes Backup.

About the archive timeout

By default, EDB Postgres for Kubernetes sets `archive_timeout` to `5min`, ensuring that WAL files, even in case of low workloads, are closed and archived at least every 5 minutes, providing a deterministic time-based value for your Recovery Point Objective (RPO).

Even though you change the value of the `archive_timeout` setting in the PostgreSQL configuration, our experience suggests that the default value set by the operator is suitable for most use cases.

27 PostgreSQL Role Management

From its inception, EDB Postgres for Kubernetes has managed the creation of specific roles required in PostgreSQL instances:

- some reserved users, such as the `postgres` superuser, `streaming_replica` and `cnp_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, EDB Postgres for Kubernetes 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 `cluster-example-with-roles.yaml`.

Here is an excerpt from that file:

```yaml
apiVersion: postgresql.k8s.enterprisedb.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 EDB Postgres for Kubernetes operator will revert those changes during the next reconciliation cycle.

Password management

The declarative role management feature includes reconciling of role passwords. Passwords are managed in fundamentally different ways in the Kubernetes world and in PostgreSQL, and as a result there are a few things to note.

Managed role configurations may optionally specify the name of a **Secret** where the username and password are stored (encoded in Base64 as is usual in Kubernetes). For example:


```yaml
  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, :

```yaml
apiVersion: v1
data:
  username:
ZGFudGU=
  password:
ZGFudGU=
kind:
Secret
metadata:
  name: cluster-example-
dante
  labels:
    k8s.enterprisedb.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 EDB Postgres for Kubernetes. 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:

```yaml
  managed:
    roles:
    - name: dante
      ensure: present
      [… snipped
…]
      disablePassword: true
```

NOTE: it is considered an error to set both `passwordSecret` and `disablePassword` on a given role. This configuration will be rejected by the validation webhook.

Password expiry, `VALID UNTIL`

The `VALID UNTIL` role attribute in PostgreSQL controls password expiry. Roles created without `VALID UNTIL` specified get NULL by default in PostgreSQL, meaning that their password will never expire.

PostgreSQL uses a timestamp type for `VALID UNTIL`, which includes support for the value `'infinity'` indicating that the password never expires. Please see the PostgreSQL documentation for reference.

With declarative role management, the `validUntil` attribute for managed roles controls password expiry. `validUntil` can only take:

- a Kubernetes timestamp, or
- be omitted (defaulting to `null`)

In the first case, the given `validUntil` timestamp will be set in the database as the `VALID UNTIL` attribute of the role.

In the second case (omitted `validUntil`) the operator will ensure password never expires, mirroring the behavior of PostgreSQL. Specifically:

- in case of new role, it will omit the `VALID UNTIL` clause in the role creation statement
- in case of existing role, it will set `VALID UNTIL` to `infinity` if `VALID UNTIL` was not set to `NULL` in the database (this is due to PostgreSQL not allowing `VALID UNTIL NULL` in the `ALTER ROLE` SQL statement)

> **Warning**
>
> New roles created without `passwordSecret` will have a `NULL` password inside PostgreSQL.

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:
    k8s.enterprisedb.io/reload: "true"
apiVersion: v1
stringData:
  username: cavalcanti
  password: SCRAM-SHA-256$<iteration count>:<salt>$<StoredKey>:
<ServerKey>
```

Unrealizable role configurations

In PostgreSQL, in some cases, commands cannot be honored by the database and will be rejected. Please refer to thePostgreSQL 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 EDB Postgres for Kubernetes 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.

EDB Postgres for Kubernetes will record when such fundamental errors occur, and will display them in the cluster Status. Which segues into...

Status of managed roles

The Cluster status includes a section for the managed roles' status, as shown below:

```
status:
  […snipped…]
  managedRolesStatus:
    byStatus:
      not-managed:
      -
app
      pending-reconciliation:
      - dante
      -
petrarca
      reconciled:
      - ariosto
      reserved:
      -
postgres
      - streaming_replica
    cannotReconcile:
      dante:
      - 'could not perform DELETE on role dante: owner of database
inferno'
      petrarca:
      - 'could not perform UPDATE_MEMBERSHIPS on role petrarca: role "poets" does not
exist'
```

Note the special sub-section `cannotReconcile` for operations the database (and EDB Postgres for Kubernetes) 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 EDB Postgres for Kubernetes users to adopt this feature at their convenience.

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 EDB Postgres for Kubernetes, 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.

> **Info**
>
> EDB Postgres for Kubernetes doesn't use `StatefulSet` for managing data persistence. Rather, it manages PVCs directly. If you want to know more, see Custom pod controller.

Backup and recovery

Since EDB Postgres for Kubernetes supports volume snapshots for both backup and recovery, we recommend that you also consider this aspect when you choose your storage solution, especially if you manage very large databases.

> **Important**
>
> See the Kubernetes documentation for a list of all the supported container storage interface (CSI) drivers that provide snapshot capabilities.

Benchmarking EDB Postgres for Kubernetes

Before deploying the database in production, we recommend that you benchmark EDB Postgres for Kubernetes in a controlled Kubernetes environment. Follow the guidelines in Benchmarking.

Briefly, we recommend operating at two levels:

- Measuring the performance of the underlying storage using fio, with relevant metrics for database workloads such as throughput for sequential reads, sequential writes, random reads, and random writes
- Measuring the performance of the database using pgbench, the default benchmarking tool distributed with PostgreSQL

> **Important**
>
> You must measure both the storage and database performance before putting the database into production. These results are extremely valuable not just in the planning phase (for example, capacity planning). They are also valuable in the production lifecycle, particularly in emergency situations when you don't have time to run this kind of test. Databases change and evolve over time, and so does the distribution of data, potentially affecting performance. Knowing the theoretical maximum throughput of sequential reads or writes is extremely useful in those situations. This is true especially in shared-nothing contexts, where results don't vary due to the influence of external workloads.
>
> Know your system: benchmark it.

Encryption at rest

Encryption at rest is possible with EDB Postgres for Kubernetes. The operator delegates that to the underlying storage class. See the storage class for information about this important security feature.

Persistent Volume Claim (PVC)

The operator creates a PVC for each PostgreSQL instance, with the goal of storing the `PGDATA`. It then mounts it into each pod.

Additionally, it supports creating clusters with:

- A separate PVC on which to store PostgreSQL WAL, as explained in Volume for WAL
- Additional separate volumes reserved for PostgreSQL tablespaces, as explained in Tablespaces

In EDB Postgres for Kubernetes, the volumes attached to a single PostgreSQL instance are defined as a *PVC group*.

Configuration via a storage class

> **Important**
>
> EDB Postgres for Kubernetes 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.k8s.enterprisedb.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:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: postgresql-storage-
class
spec:
  instances: 3
  storage:
    storageClass:
standard
    size:
1Gi
```

Configuration via a PVC template

To further customize the generated PVCs, you can provide a PVC template inside the custom resource, like in the following example:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: postgresql-pvc-
template
spec:
  instances: 3

  storage:
    pvcTemplate:
      accessModes:
        - ReadWriteOnce
      resources:
        requests:
          storage:
1Gi
      storageClassName:
standard
      volumeMode: Filesystem
```

Volume for WAL

By default, PostgreSQL stores all its data in the so-called `PGDATA` (a directory). One of the core directories inside `PGDATA` is `pg_wal`, which contains the log of transactional changes that occurred in the database, in the form of segment files. (`pg_wal` is historically known as `pg_xlog` in PostgreSQL.)

> **Info**
>
> 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.

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.

> **Write-Ahead Log (WAL)**
>
> 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.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: separate-pgwal-volume
spec:
  instances: 3
  storage:
    size:
1Gi
  walStorage:
    size:
1Gi
```

> **Important**
>
> Removing `walStorage` isn't supported. Once added, a separate volume for WALs can't be removed from an existing Postgres cluster.

Volumes for tablespaces

EDB Postgres for Kubernetes supports declarative tablespaces. You can add one or more volumes, each dedicated to a single PostgreSQL tablespace. See Tablespaces for details.

Volume expansion

Kubernetes exposes an API allowing expanding PVCs that's enabled by default. However, it needs to be supported by the underlying `StorageClass`.

To check if a certain `StorageClass` supports volume expansion, you can read the `allowVolumeExpansion` field for your storage class:

```
$ kubectl get storageclass -o jsonpath='{$.allowVolumeExpansion}' premium-storage
true
```

Using the volume expansion Kubernetes feature

Given the storage class supports volume expansion, you can change the size requirement of the `Cluster`, and the operator applies the change to every PVC.

If the `StorageClass` supports online volume resizing, the change is immediately applied to the pods. If the underlying storage class doesn't support that, you must delete the pod to trigger the resize.

The best way to proceed is to delete one pod at a time, starting from replicas and waiting for each pod to be back up.

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:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: postgresql-pvc-
template
spec:
  instances: 3

  storage:
    storageClass:
standard
    size:
1Gi
    resizeInUseVolumes: False
```

To move the entire cluster to a different storage area, you need to re-create all the PVCs and all the pods. Suppose you have a cluster with three replicas, like in the following example:

```
$ kubectl get pods
NAME                READY   STATUS    RESTARTS   AGE
cluster-example-1   1/1     Running   0          2m37s
cluster-example-2   1/1     Running   0          2m22s
cluster-example-3   1/1     Running   0          2m10s
```

To re-create the cluster using different PVCs, you can edit the cluster definition to disable `resizeInUseVolumes`. Then re-create every instance in a different PVC.

For example, re-create the storage for `cluster-example-3`:

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

> **Important**
>
> If you created a dedicated WAL volume, both PVCs must be deleted during this process. The same procedure applies if you want to regenerate the WAL volume PVC. You can do this by also disabling `resizeInUseVolumes` for the `.spec.walStorage` section.

For example, if a PVC dedicated to WAL storage is present:

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

Having done that, the operator orchestrates creating another replica with a resized PVC:


```
$ kubectl get pods
NAME                        READY   STATUS      RESTARTS   AGE
cluster-example-1           1/1     Running     0          5m58s
cluster-example-2           1/1     Running     0          5m43s
cluster-example-4-join-v2   0/1     Completed   0          17s
cluster-example-4           1/1     Running     0          10s
```

Static provisioning of persistent volumes

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes was built.

To use a pre-provisioned volume in EDB Postgres for Kubernetes:

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 EDB Postgres for Kubernetes to create the needed `PersistentVolumeClaim`.

> **Warning**
>
> With static provisioning, it's your responsibility to ensure that Postgres pods can be correctly scheduled by Kubernetes where a pre-provisioned volume exists. (The scheduling configuration is based on the affinity rules of your cluster.) Make sure you check for any pods stuck in `Pending` after you deploy the cluster. If the condition persists, investigate why it's happening.

Block storage considerations (Ceph/Longhorn)

Most block storage solutions in Kubernetes, 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, EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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 `cnp` plugin to destroy the affected instance. EDB Postgres for Kubernetes 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.

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*.

> **Info**
>
> 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 EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes instead of alternative approaches such as pod templates.

Predefined labels

EDB Postgres for Kubernetes manages the following predefined labels:

`k8s.enterprisedb.io/backupDate` : The date of the backup in ISO 8601 format (`YYYYMMDD`). This label is available only on `VolumeSnapshot` resources.

`k8s.enterprisedb.io/backupName` : Backup identifier. This label is available only on `VolumeSnapshot` resources.

`k8s.enterprisedb.io/backupMonth` : The year/month when a backup was taken. This label is available only on `VolumeSnapshot` resources.

`k8s.enterprisedb.io/backupTimeline` : The timeline of the instance when a backup was taken. This label is available only on `VolumeSnapshot` resources.

`k8s.enterprisedb.io/backupYear` : The year a backup was taken. This label is available only on `VolumeSnapshot` resources.

`k8s.enterprisedb.io/cluster` : Name of the cluster.

`k8s.enterprisedb.io/immediateBackup` : Applied to a `Backup` resource if the backup is the first one created from a `ScheduledBackup` object having `immediate` set to `true` .

`k8s.enterprisedb.io/instanceName` : Name of the PostgreSQL instance (replaces the old and deprecated `postgresql` label).

`k8s.enterprisedb.io/jobRole` : Role of the job (that is, `import` , `initdb` , `join` , ...)

`k8s.enterprisedb.io/onlineBackup` : Whether the backup is online (hot) or taken when Postgres is down (cold). This label is available only on `VolumeSnapshot` resources.

`postgresql` : deprecated, Name of the PostgreSQL instance. Use `k8s.enterprisedb.io/instanceName` instead

`k8s.enterprisedb.io/podRole` : Distinguishes pods dedicated to pooler deployment from those used for database instances.

`k8s.enterprisedb.io/poolerName` : Name of the PgBouncer pooler.

`k8s.enterprisedb.io/pvcRole` : Purpose of the PVC, such as `PG_DATA` or `PG_WAL` .

`k8s.enterprisedb.io/reload` : Available on `ConfigMap` and `Secret` resources. When set to `true` , a change in the resource is automatically reloaded by the operator.

`k8s.enterprisedb.io/userType` : Specifies the type of PostgreSQL user associated with the `Secret` , either `superuser` (Postgres superuser access) or `app` (application-level user in EDB Postgres for Kubernetes terminology), and is limited to the default users created by EDB Postgres for Kubernetes (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 `k8s.enterprisedb.io/instanceRole` instead.

`k8s.enterprisedb.io/scheduled-backup` : When available, name of the `ScheduledBackup` resource that created a given `Backup` object.

`k8s.enterprisedb.io/instanceRole` : Whether the instance running in a pod is a `primary` or a `replica` .

Predefined annotations

EDB Postgres for Kubernetes manages the following predefined annotations:

`container.apparmor.security.beta.kubernetes.io/*` : Name of the AppArmor profile to apply to the named container. See AppArmor for details.

`k8s.enterprisedb.io/backupEndTime` : The time a backup ended. This annotation is available only on `VolumeSnapshot` resources.

`k8s.enterprisedb.io/backupEndWAL` : The WAL at the conclusion of a backup. This annotation is available only on `VolumeSnapshot` resources.

`k8s.enterprisedb.io/backupStartTime` : The time a backup started.

`k8s.enterprisedb.io/backupStartWAL` : The WAL at the start of a backup. This annotation is available only on `VolumeSnapshot` resources.

`k8s.enterprisedb.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.

`k8s.enterprisedb.io/clusterManifest` : Manifest of the `Cluster` owning this resource (such as a PVC). This label replaces the old, deprecated `k8s.enterprisedb.io/hibernateClusterManifest` label.

`k8s.enterprisedb.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.

`k8s.enterprisedb.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.

`k8s.enterprisedb.io/hash` : The hash value of the resource.

`k8s.enterprisedb.io/hibernation` : Applied to a `Cluster` resource to control the declarative hibernation feature. Allowed values are `on` and `off` .

`k8s.enterprisedb.io/managedSecrets` : Pull secrets managed by the operator and automatically set in the `ServiceAccount` resources for each Postgres cluster.

`k8s.enterprisedb.io/nodeSerial` : On a pod resource, identifies the serial number of the instance within the Postgres cluster.

`k8s.enterprisedb.io/operatorVersion` : Version of the operator.

`k8s.enterprisedb.io/pgControldata` : Output of the `pg_controldata` command. This annotation replaces the old, deprecated `k8s.enterprisedb.io/hibernatePgControlData` annotation.

`k8s.enterprisedb.io/podEnvHash` : Deprecated, as the `k8s.enterprisedb.io/podSpec` annotation now also contains the pod environment.

`k8s.enterprisedb.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 cnp restart`.
```

`k8s.enterprisedb.io/podSpec` : Snapshot of the `spec` of the pod generated by the operator. This annotation replaces the old, deprecated `k8s.enterprisedb.io/podEnvHash` annotation.

`k8s.enterprisedb.io/poolerSpecHash` : Hash of the pooler resource.

`k8s.enterprisedb.io/pvcStatus` : Current status of the PVC: `initializing`, `ready`, or `detached`.

`k8s.enterprisedb.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`.
```

`k8s.enterprisedb.io/reconciliationLoop` : When set to `disabled` on a `Cluster`, the operator prevents the reconciliation loop from running.

`k8s.enterprisedb.io/reloadedAt` : Contains the latest cluster `reload` time. `reload` is triggered by the user through a plugin.

`k8s.enterprisedb.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.

`k8s.enterprisedb.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.

`k8s.enterprisedb.io/snapshotStartTime` : The time a snapshot started.

`k8s.enterprisedb.io/snapshotEndTime` : The time a snapshot was marked as ready to use.

`k8s.enterprisedb.io/validation` : When set to `disabled` on a EDB Postgres for Kubernetes-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.
```

`k8s.enterprisedb.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.

Prerequisites

By default, no label or annotation defined in the cluster's metadata is inherited by the associated resources. To enable label/annotation inheritance, follow the instructions provided in Operator configuration.

The following continues from that example and limits it to the following:

- Annotations: `categories`
- Labels: `app` , `environment` , and `workload`

> **Note**
>
> Feel free to select the names that most suit your context for both annotations and labels. You can also use wildcards in naming and adopt strategies like using `mycompany/*` for all labels or setting annotations starting with `mycompany/` to be inherited.

Defining cluster's metadata

When defining the cluster, before any resource is deployed, you can set the metadata as follows:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
  annotations:
    categories:
database
  labels:
    environment: production
    workload:
database
    app:
sso
spec:
     # ...
<snip>
```

Once the cluster is deployed, you can verify, for example, that the labels were correctly set in the pods:

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


Current limitations

Currently, EDB Postgres for Kubernetes 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.

30 Monitoring

> **Important**
>
> Installing Prometheus and Grafana is beyond the scope of this project. We assume they are correctly installed in your system. However, for experimentation we provide instructions in Part 4 of the Quickstart.

Monitoring Instances

For each PostgreSQL instance, the operator provides an exporter of metrics for Prometheus via HTTP 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" section below for details).

> **Important**
>
> EDB Postgres for Kubernetes, by default, installs a set of predefined metrics in a `ConfigMap` named `default-monitoring`.

> **Info**
>
> 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 `cnp_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.

> **Prometheus/Grafana**
>
> If you are interested in evaluating the integration of EDB Postgres for Kubernetes with Prometheus and Grafana, you can find a quick setup guide in Part 4 of the quickstart

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:

```yaml
apiVersion:
monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cluster-
example
spec:
  selector:
    matchLabels:
      "k8s.enterprisedb.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 `k8s.enterprisedb.io/cluster` label instead to select the instances.

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:
      "k8s.enterprisedb.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`.

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 `cnp_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 cnp_collector_collection_duration_seconds Collection time duration in seconds
# TYPE cnp_collector_collection_duration_seconds gauge
cnp_collector_collection_duration_seconds{collector="Collect.up"} 0.0031393

# HELP cnp_collector_collections_total Total number of times PostgreSQL was accessed for metrics.
# TYPE cnp_collector_collections_total counter
```



```
cnp_collector_collections_total 2

# HELP cnp_collector_fencing_on 1 if the instance is fenced, 0 otherwise
# TYPE cnp_collector_fencing_on gauge
cnp_collector_fencing_on 0

# HELP cnp_collector_nodes_used NodesUsed represents the count of distinct nodes accommodating the
instances. A value of '-1' suggests that the metric is not available. A value of '1' suggests that all
instances are hosted on a single node, implying the absence of High Availability (HA). Ideally this value
should match the number of instances in the cluster.
# TYPE cnp_collector_nodes_used gauge
cnp_collector_nodes_used 3

# HELP cnp_collector_last_collection_error 1 if the last collection ended with error, 0 otherwise.
# TYPE cnp_collector_last_collection_error gauge
cnp_collector_last_collection_error 0

# HELP cnp_collector_manual_switchover_required 1 if a manual switchover is required, 0 otherwise
# TYPE cnp_collector_manual_switchover_required gauge
cnp_collector_manual_switchover_required 0

# HELP cnp_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 cnp_collector_pg_wal gauge
cnp_collector_pg_wal{value="count"} 9
cnp_collector_pg_wal{value="slots_max"} NaN
cnp_collector_pg_wal{value="keep"} 32
cnp_collector_pg_wal{value="max"} 64
cnp_collector_pg_wal{value="min"} 5
cnp_collector_pg_wal{value="size"} 1.50994944e+08
cnp_collector_pg_wal{value="volume_max"} 128
cnp_collector_pg_wal{value="volume_size"} 2.147483648e+09

# HELP cnp_collector_pg_wal_archive_status Number of WAL segments in the
'/var/lib/postgresql/data/pgdata/pg_wal/archive_status' directory (ready, done)
# TYPE cnp_collector_pg_wal_archive_status gauge
cnp_collector_pg_wal_archive_status{value="done"} 6
cnp_collector_pg_wal_archive_status{value="ready"} 0

# HELP cnp_collector_replica_mode 1 if the cluster is in replica mode, 0 otherwise
# TYPE cnp_collector_replica_mode gauge
cnp_collector_replica_mode 0

# HELP cnp_collector_sync_replicas Number of requested synchronous replicas (synchronous_standby_names)
# TYPE cnp_collector_sync_replicas gauge
cnp_collector_sync_replicas{value="expected"} 0
cnp_collector_sync_replicas{value="max"} 0
cnp_collector_sync_replicas{value="min"} 0
cnp_collector_sync_replicas{value="observed"} 0

# HELP cnp_collector_up 1 if PostgreSQL is up, 0 otherwise.
# TYPE cnp_collector_up gauge
cnp_collector_up{cluster="cluster-example"} 1

# HELP cnp_collector_postgres_version Postgres version
# TYPE cnp_collector_postgres_version gauge
cnp_collector_postgres_version{cluster="cluster-example",full="17.5"} 17.5

# HELP cnp_collector_last_failed_backup_timestamp The last failed backup as a unix timestamp
# TYPE cnp_collector_last_failed_backup_timestamp gauge
cnp_collector_last_failed_backup_timestamp 0
```

```
# HELP cnp_collector_last_available_backup_timestamp The last available backup as a unix timestamp
# TYPE cnp_collector_last_available_backup_timestamp gauge
cnp_collector_last_available_backup_timestamp 1.63238406e+09

# HELP cnp_collector_first_recoverability_point The first point of recoverability for the cluster as a unix
timestamp
# TYPE cnp_collector_first_recoverability_point gauge
cnp_collector_first_recoverability_point 1.63238406e+09

# HELP cnp_collector_lo_pages Estimated number of pages in the pg_largeobject table
# TYPE cnp_collector_lo_pages gauge
cnp_collector_lo_pages{datname="app"} 0
cnp_collector_lo_pages{datname="postgres"} 78

# HELP cnp_collector_wal_buffers_full Number of times WAL data was written to disk because WAL buffers
became full. Only available on PG 14+
# TYPE cnp_collector_wal_buffers_full gauge
cnp_collector_wal_buffers_full{stats_reset="2023-06-19T10:51:27.473259Z"} 6472

# HELP cnp_collector_wal_bytes Total amount of WAL generated in bytes. Only available on PG 14+
# TYPE cnp_collector_wal_bytes gauge
cnp_collector_wal_bytes{stats_reset="2023-06-19T10:51:27.473259Z"} 1.0035147e+07

# HELP cnp_collector_wal_fpi Total number of WAL full page images generated. Only available on PG 14+
# TYPE cnp_collector_wal_fpi gauge
cnp_collector_wal_fpi{stats_reset="2023-06-19T10:51:27.473259Z"} 1474

# HELP cnp_collector_wal_records Total number of WAL records generated. Only available on PG 14+
# TYPE cnp_collector_wal_records gauge
cnp_collector_wal_records{stats_reset="2023-06-19T10:51:27.473259Z"} 26178

# HELP cnp_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 cnp_collector_wal_sync gauge
cnp_collector_wal_sync{stats_reset="2023-06-19T10:51:27.473259Z"} 37

# HELP cnp_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 cnp_collector_wal_sync_time gauge
cnp_collector_wal_sync_time{stats_reset="2023-06-19T10:51:27.473259Z"} 0

# HELP cnp_collector_wal_write Number of times WAL buffers were written out to disk via XLogWrite request.
Only available on PG 14+
# TYPE cnp_collector_wal_write gauge
cnp_collector_wal_write{stats_reset="2023-06-19T10:51:27.473259Z"} 7243

# HELP cnp_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 cnp_collector_wal_write_time gauge
cnp_collector_wal_write_time{stats_reset="2023-06-19T10:51:27.473259Z"} 0

# HELP cnp_last_error 1 if the last collection ended with error, 0 otherwise.
# TYPE cnp_last_error gauge
cnp_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

# 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 program's 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.
# 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.
# 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**
>
> `cnp_collector_postgres_version` is a GaugeVec metric containing the `Major.Minor` version of Postgres (either PostgreSQL or EPAS). The full semantic version `Major.Minor.Patch` can be found inside one of its label field named `full`.

> **Note**
>
> `cnp_collector_first_recoverability_point` and `cnp_collector_last_available_backup_timestamp` will be zero until your first backup to the object store. This is separate from the WAL archival.

User defined metrics

This feature is currently in *beta* state and the format is inspired by the queries.yaml file (release 0.12) of the PostgreSQL Prometheus Exporter.

Custom metrics can be defined by users by referring to the created `Configmap` / `Secret` in a `Cluster` definition under the `.spec.monitoring.customQueriesConfigMap` or `customQueriesSecret` section as in the following example:

```
apiVersion: postgresql.k8s.enterprisedb.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 `k8s.enterprisedb.io/reload` to it, otherwise you will have to reload the instances using the `kubectl cnp reload` subcommand.

> **Important**
>
> When a user defined metric overwrites an already existing metric the instance manager prints a json warning log, containing the message: `Query with the same name already found. Overwriting the existing one.` and a key `queryName` containing the overwritten query name.

Example of a user defined metric

Here you can see an example of a `ConfigMap` containing a single custom query, referenced by the `Cluster` example above:


```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: example-
monitoring
  namespace: test
  labels:
    k8s.enterprisedb.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` file that is already installed in your EDB Postgres for Kubernetes deployment (see "Default set of metrics").

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"
```

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:

```
cnp_some_query_rows{datname="albert"} 2
cnp_some_query_rows{datname="bb"} 5
cnp_some_query_rows{datname="freddie"} 10
```

Here is an example of a query with auto-discovery enabled which also runs on the `template1` database (otherwise not returned by the aforementioned query):


```
some_query: |
  query:
|

SELECT
     current_database() as
datname,
     count(*) as
rows
    FROM some_table

metrics:
    -
datname:
        usage:
"LABEL"
        description: "Name of current database"
    -
rows:
        usage:
"GAUGE"
        description: "number of rows"
  target_databases:
    -
"*"
    -
"template1"
```

The above example will produce the following metrics (provided the databases exist):

```
cnp_some_query_rows{datname="albert"} 2
cnp_some_query_rows{datname="bb"} 5
cnp_some_query_rows{datname="freddie"} 10
cnp_some_query_rows{datname="template1"} 7
cnp_some_query_rows{datname="postgres"} 42
```

Structure of a user defined metric

Every custom query has the following basic structure:

```
<MetricName>:
      query: "<SQLQuery>"
      metrics:
        - <ColumnName>:
            usage: "<MetricType>"
            description: "<MetricDescription>"
```

Here is a short description of all the available fields:

- `<MetricName>` : the name of the Prometheus metric
  - `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 shell-like pattern 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" page from the Prometheus documentation for more information.

Output of a user defined metric

Custom defined metrics are returned by the Prometheus exporter endpoint (`:9187/metrics`) with the following format:

```
cnp_<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 cnp_pg_replication_in_recovery Whether the instance is in recovery
# TYPE cnp_pg_replication_in_recovery gauge
cnp_pg_replication_in_recovery 0
# HELP cnp_pg_replication_lag Replication lag behind primary in seconds
# TYPE cnp_pg_replication_lag gauge
cnp_pg_replication_lag 0
# HELP cnp_pg_replication_streaming_replicas Number of streaming replicas connected to the instance
# TYPE cnp_pg_replication_streaming_replicas gauge
cnp_pg_replication_streaming_replicas 2
# HELP cnp_pg_replication_is_wal_receiver_up Whether the instance wal_receiver is up
# TYPE cnp_pg_replication_is_wal_receiver_up gauge
cnp_pg_replication_is_wal_receiver_up 0
```

Default set of metrics

The operator can be configured to automatically inject in a Cluster a set of monitoring queries defined in a ConfigMap or a Secret, inside the operator's namespace. You have to set the `MONITORING_QUERIES_CONFIGMAP` or `MONITORING_QUERIES_SECRET` key in the "operator configuration", respectively to the name of the ConfigMap or the Secret; the operator will then use the content of the `queries` key.

Any change to the `queries` content will be immediately reflected on all the deployed Clusters using it.

The operator installation manifests come with a predefined ConfigMap, called `postgresql-operator-default-monitoring`, to be used by all Clusters. `MONITORING_QUERIES_CONFIGMAP` is by default set to `postgresql-operator-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: `postgresql-operator-default-monitoring`. So that, if you intend to have default metrics, you should not create a ConfigMap with this name in the cluster's namespace.

Differences with the Prometheus Postgres exporter

EDB Postgres for Kubernetes is inspired by the PostgreSQL Prometheus Exporter, but presents some differences. In particular, the `cache_seconds` field is not implemented in EDB Postgres for Kubernetes' exporter.

Monitoring the EDB Postgres for Kubernetes operator

The operator internally exposes Prometheus metrics via HTTP on port 8080, named `metrics`.

> **Info**
>
> 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.

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 postgresql-operator-system apply -f -
<<EOF
---
apiVersion:
monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: postgresql-operator-controller-
manager
spec:
  selector:
    matchLabels:
      app.kubernetes.io/name: cloud-native-
postgresql
  podMetricsEndpoints:
    - port: metrics
EOF
```

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 `postgresql-operator-system` namespace. Please adapt to your use case.

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 postgresql-operator-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`.

Using curl

Create the `curl` pod with the following command:

```
kubectl apply -f -
<<EOF
---
apiVersion: v1
kind:
Pod
metadata:
  name: curl
spec:
  containers:
  - 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
```

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 EDB Postgres for Kubernetes. NOTE: this does not include inter-operation with notification services. Please refer to the Prometheus documentation.
- `podmonitor.yaml` : a `PodMonitor` for the EDB Postgres for Kubernetes Operator deployment.

In addition, we provide the "raw" sources for the Prometheus alert rules in the `alerts.yaml` file.

A Grafana dashboard for EDB Postgres for Kubernetes 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.

Monitoring on OpenShift

Starting on Openshift 4.6 there is a complete monitoring stack called "Monitoring for user-defined projects" which can be enabled by cluster administrators. Cloud Native PostgreSQL will automatically create a `PodMonitor` object if the option `spec.monitoring.enablePodMonitor` of the `Cluster` definition is set to `true` .

To enable cluster wide `user-defined` monitoring you must first create a `ConfigMap` with the name `cluster-monitoring-config` in the `openshift-monitoring` namespace/project with the following content:

```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-
config
  namespace: openshift-monitoring
data:
  config.yaml:
|
    enableUserWorkload:
true
```

If the `ConfigMap` already exists, just add the variable `enableUserWorkload: true` .

> **Important**
>
> This will enable the monitoring for the whole cluster, if it is needed only for one namespace/project please refer to the official Red Hat documentation or talk with your cluster administrator.

After that, just create the proper PodMonitor in the namespace/project with something similar to this:

```yaml
apiVersion:
monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: cluster-
sample
spec:
  selector:
    matchLabels:
      postgresql: cluster-
sample
  podMetricsEndpoints:
    - port: metrics
```

> **Note**
>
> We currently don't use `ServiceMonitor` because our service doesn't define a port pointing to the metrics. If we added a metric port this could expose sensitive data.

31 Logging

EDB Postgres for Kubernetes 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.

> **Info**
>
> 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.

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.

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.

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": "",
    "location": "",
    "application_name": "",
    "backend_type": "startup"
  },
  "logging_pod": "cluster-example-1",
}
```

> **Info**
>
> Internally, the operator uses PostgreSQL's CSV log format. For more details, refer to the PostgreSQL documentation on CSV log format.

PGAudit Logs

EDB Postgres for Kubernetes 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`. EDB Postgres for Kubernetes 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**
>
> EDB Postgres for Kubernetes 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.k8s.enterprisedb.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"

  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.

EDB Audit logs

Clusters that are running on EDB Postgres Advanced Server (EPAS) can enable EDB Audit as follows:


```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  instances: 3
  imageName: quay.io/enterprisedb/edb-postgres-advanced:13
  licenseKey: <LICENSE>

  postgresql:
    epas:
      audit: true

  storage:
    size:
1Gi
```

Setting `.spec.postgresql.epas.audit: true` enforces the following parameters:

```
edb_audit = 'csv'
edb_audit_destination = 'file'
edb_audit_directory = '/controller/log'
edb_audit_filename = 'edb_audit'
edb_audit_rotation_day = 'none'
edb_audit_rotation_seconds = '0'
edb_audit_rotation_size = '0'
edb_audit_tag = ''
edb_log_every_bulk_value = 'false'
```

Other parameters can be passed via `.spec.postgresql.parameters` as usual.

The audit CSV logs are parsed and routed to stdout in JSON format, similarly to all the remaining logs:

- `.logger` set to `edb_audit`
- `.msg` set to `record`
- `.record` containing the whole parsed record as a JSON object

See the example below:

```
{
  "level": "info",
  "ts": 1624629110.7641866,
  "logger": "edb_audit",
  "msg": "record",
  "record": {
    "log_time": "2021-06-25 13:51:50.763 UTC",
    "user_name": "postgres",
    "database_name": "postgres",
    "process_id": "68",
    "connection_from": "[local]",
    "session_id": "60d5df76.44",
    "session_line_num": "5",
    "process_status": "idle in transaction",
    "session_start_time": "2021-06-25 13:51:50
UTC",
    "virtual_transaction_id": "3/93",
    "transaction_id": "1183",
    "error_severity": "AUDIT",
    "sql_state_code": "00000",
    "message": "statement: GRANT EXECUTE ON function pg_catalog.pg_read_binary_file(text) TO
\"streaming_replica\"",
    "detail": "",
    "hint": "",
    "internal_query": "",
    "internal_query_pos": "",
    "context": "",
    "query": "",
    "query_pos": "",
    "location": "",
    "application_name": "",
    "backend_type": "client
backend",
    "command_tag": "GRANT",
    "audit_tag": "",
    "type": "grant"
  },
  "logging_pod": "cluster-example-1",
}
```

See EDB Audit file for more details about the records' fields.

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`
- `edb_audit` : from the EDB Audit extension
- `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` and `edb_audit` , which follows a specific structure, all other `logger` values contain the `msg` field with the escaped message that is logged.

32 Certificates

EDB Postgres for Kubernetes 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.

EDB Postgres for Kubernetes is very flexible when it comes to TLS certificates. It primarily operates in two modes:

1. **Operator managed** – Certificates are internally managed by the operator in a fully automated way and signed using a CA created by EDB Postgres for Kubernetes.
2. **User provided** – Certificates are generated outside the operator and imported in the cluster definition as secrets. EDB Postgres for Kubernetes 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 CNP.

> **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.

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).

> **Info**
>
> 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 EDB Postgres for Kubernetes 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.

Server certificates

Server CA secret

The operator generates a self-signed CA and stores it in a generic secret containing the following keys:

- `ca.crt` – CA certificate used to validate the server certificate, used as `sslrootcert` in clients' connection strings.
- `ca.key` – The key used to sign the server SSL certificate automatically.

Server TLS secret

The operator uses the generated self-signed CA to sign a server TLS certificate. It's stored in a secret of type `kubernetes.io/tls` and configured to be used as `ssl_cert_file` and `ssl_key_file` by the instances. This approach enables clients to verify their identity and connect securely.

Server alternative DNS names

In addition to the default ones, you can specify DNS server alternative names as part of the generated server TLS secret.

Client certificates

Client CA secret

By default, the same self-signed CA as the server CA is used. The public part is passed as `ssl_ca_file` to all the instances so it can verify client certificates it signed. The private key is stored in the same secret and used to sign client certificates generated by the `kubectl cnp` plugin.

Client `streaming_replica` certificate

The operator uses the generated self-signed CA to sign a client certificate for the user `streaming_replica`, storing it in a secret of type `kubernetes.io/tls`. To allow secure connection to the primary instance, this certificate is passed as `sslcert` and `sslkey` in the replicas' connection strings.

User-provided certificates mode

Server certificates

If required, you can also provide the two server certificates, generating them using a separate component such ascert-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 `k8s.enterprisedb.io/reload` to it. Otherwise you must reload the instances using the `kubectl cnp reload` subcommand.

Example

Given the following files:

- `server-ca.crt` – The certificate of the CA that signed the server TLS certificate.
- `server.crt` – The certificate of the server TLS certificate.
- `server.key` – The private key of the server TLS certificate.

Create a secret containing the CA certificate:

```
kubectl create secret generic my-postgresql-server-ca
\
  --from-file=ca.crt=./server-ca.crt
```

Create a secret with the TLS certificate:

```
kubectl create secret tls my-postgresql-server
\
  --cert=./server.crt --key=./server.key
```

Create a PostgreSQL cluster referencing those secrets:

```
kubectl apply -f - <<EOF
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  instances:
3
  certificates:
    serverCASecret: my-postgresql-server-
ca
    serverTLSSecret: my-postgresql-
server

storage:
    storageClass:
standard
    size:
1Gi
EOF
```

The new cluster uses the provided server certificates for TLS connections.

Cert-manager example

This simple example shows how to use cert-manager to set up a self-signed CA and generate the needed TLS server certificate:

```
---
apiVersion: cert-manager.io/v1
kind:
Issuer
metadata:
  name: selfsigned-
issuer
spec:
  selfSigned: {}
---
apiVersion: v1
kind:
Secret
metadata:
  name: my-postgres-server-
cert
  labels:
    k8s.enterprisedb.io/reload: ""
---
apiVersion: cert-manager.io/v1
kind:
Certificate
metadata:
  name: my-postgres-server-
cert
spec:
  secretName: my-postgres-server-
cert
  usages:
    - server
auth
  dnsNames:
    - cluster-example-
lb.internal.mydomain.net
    - cluster-example-
rw
    - cluster-example-
rw.default
    - cluster-example-
rw.default.svc
    - cluster-example-
r
    - cluster-example-
r.default
    - cluster-example-
r.default.svc
    - cluster-example-
ro
    - cluster-example-
ro.default
    - cluster-example-
ro.default.svc
  issuerRef:
    name: selfsigned-
issuer
    kind:
Issuer
    group: cert-manager.io
```

Cert-manager creates a secret named `my-postgres-server-cert`. It contains all the needed files and can be referenced from a cluster as follows:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  instances: 3
  certificates:
    serverTLSSecret: my-postgres-server-
cert
    serverCASecret: my-postgres-server-
cert
  storage:
    size:
1Gi
```

You can find a complete example using cert-manager to manage both server and client CA and certificates in the cluster-example-cert-manager.yaml deployment manifest.

Client certificate

If required, you can also provide the two client certificates, generating them using a separate component such as cert-manager 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 cnp certificate`.

> **Note**
>
> If you want ConfigMaps and secrets to be automatically reloaded by instances, you can add a label with the key `k8s.enterprisedb.io/reload` to it. Otherwise, you must reload the instances using the `kubectl cnp reload` subcommand.

Cert-manager example

This simple example shows how to use cert-manager to set up a self-signed CA and generate the needed TLS server certificate:


```yaml
---
apiVersion: cert-manager.io/v1
kind: Issuer
metadata:
  name: selfsigned-issuer
spec:
  selfSigned: {}
---
apiVersion: v1
kind: Secret
metadata:
  name: my-postgres-client-cert
  labels:
    k8s.enterprisedb.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:

```yaml
apiVersion: postgresql.k8s.enterprisedb.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 thecluster-example-cert-manager.yaml deployment manifest.

33 Client TLS/SSL connections

> **Certificates**
>
> See Certificates for more details on how EDB Postgres for Kubernetes supports TLS certificates.

The EDB Postgres for Kubernetes 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 CNP operator come with a certification authority (CA) to create and sign TLS client certificates. Using the cnp 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 cluster-example-pg-hba.yaml 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.)

Issuing a new certificate

> **About CNP plugin for kubectl**
>
> See the Certificates in the EDB Postgres for Kubernetes 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 cnp certificate cluster-app \
  --cnp-cluster cluster-example \
  --cnp-user app
```

You can now verify the certificate:

```
kubectl get secret cluster-app \
  -o jsonpath="{.data['tls\.crt']}" \
  | base64 -d | openssl x509 -text -noout \
  | head -n 11
```

Output:

```

Certificate:
  Data:
    Version: 3 (0x2)
    Serial Number:
      5d:e1:72:8a:39:9f:ce:51:19:9d:21:ff:1e:4b:24:5d
    Signature Algorithm: ecdsa-with-SHA256
    Issuer: OU = default, CN = cluster-example
    Validity
      Not Before: Mar 22 10:22:14 2021 GMT
      Not After : Mar 22 10:22:14 2022 GMT
    Subject: CN = app
```

As you can see, TLS client certificates by default are created with 90 days of validity, and with a simple CN that corresponds to the username in PostgreSQL. You can specify the validity and threshold values using the `EXPIRE_CHECK_THRESHOLD` and `CERTIFICATE_DURATION` parameters. This is necessary to leverage the `cert` authentication method for `hostssl` entries in `pg_hba.conf`.

Testing the connection via a TLS certificate

Next, test this client certificate by configuring a demo client application that connects to your EDB Postgres for Kubernetes cluster.

The following manifest, called `cert-test.yaml`, creates a demo pod with a test application in the same namespace where your database cluster is running:

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: cert-test
spec:
  replicas: 1
  selector:
    matchLabels:
      app: webtest
  template:
    metadata:
      labels:
        app: webtest
    spec:
      containers:
        - image: ghcr.io/cloudnative-pg/webtest:1.7.0
          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 EDB Postgres for Kubernetes 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 cnp 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)
```

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.

34 Connecting from an application

Applications are supposed to work with the services created by EDB Postgres for Kubernetes 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.

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

DNS resolution

You can use the Kubernetes DNS service to point to a given server. The Kubernetes DNS service is required by the operator. You can do that by using the name of the service if the application is deployed in the same namespace as the PostgreSQL cluster. In case the PostgreSQL cluster resides in a different namespace, you can use the full qualifier: `service-name.namespace-name` .

DNS is the preferred and recommended discovery method.

Environment variables

If you deploy your application in the same namespace that contains the PostgreSQL cluster, you can also use environment variables to connect to the database.

For example, suppose that your PostgreSQL cluster is called `pg-database` , you can use the following environment variables in your applications:

- `PG_DATABASE_R_SERVICE_HOST` : the IP address of the service pointing to all the PostgreSQL instances for read-only workloads

- `PG_DATABASE_RO_SERVICE_HOST` : the IP address of the service pointing to all hot-standby replicas of the cluster

- `PG_DATABASE_RW_SERVICE_HOST` : the IP address of the service pointing to the *primary* instance of the cluster

Secrets

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

- `[cluster name]-app` (unless you have provided an existing secret through `.spec.bootstrap.initdb.secret.name`)
- `[cluster name]-superuser` (if `.spec.enableSuperuserAccess` is set to `true` and you have not specified a different secret using `.spec.superuserSecret`)

Each secret contain the following:

- username
- password
- hostname to the RW service
- port number
- database name
- a working `.pgpass file`
- uri
- jdbc-uri

The `-app` credentials are the ones that should be used by applications connecting to the PostgreSQL cluster, and correspond to the user *owning* the database.

The `-superuser` ones are supposed to be used only for administrative purposes, and correspond to the `postgres` user.

> **Important**
>
> Superuser access over the network is disabled by default.

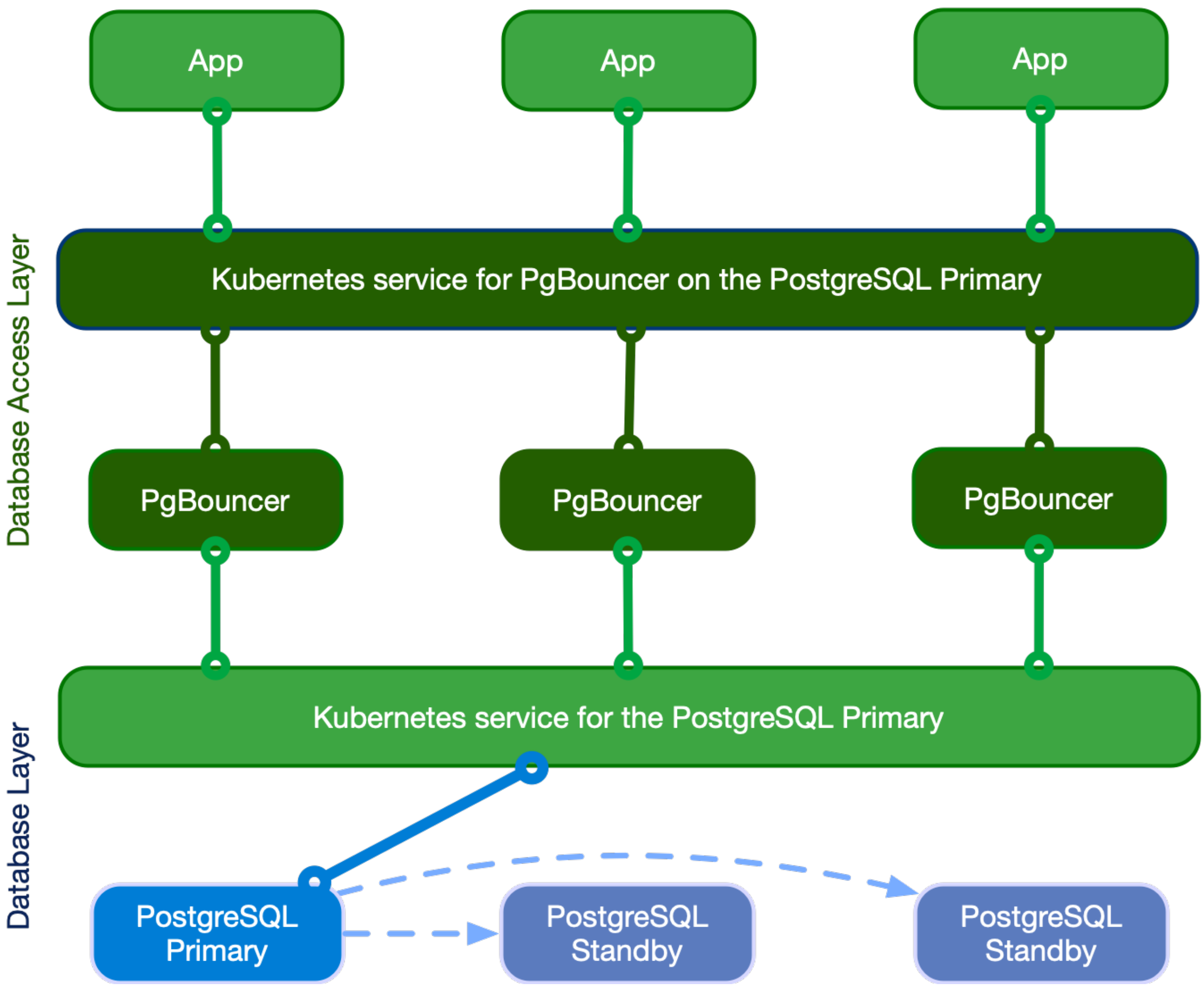
35 Connection pooling

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes is a deployment of PgBouncer pods that sits between your applications and a PostgreSQL service, for example, the `rw` service. It creates a separate, scalable, configurable, and highly available database access layer.

Architecture

The following diagram highlights how introducing a database access layer based on PgBouncer changes the architecture of EDB Postgres for Kubernetes. 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.



Quick start

This example helps to show how EDB Postgres for Kubernetes implements a PgBouncer pooler:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind:
Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-
example

  instances: 3
  type: rw
  pgbouncer:
    poolMode: session
    parameters:
      max_client_conn: "1000"
      default_pool_size: "10"
```

> **Important**
>
> The pooler name can't be the same as any cluster name in the same namespace.

This example creates a `Pooler` resource called `pooler-example-rw` that's strictly associated with the Postgres `Cluster` resource called `cluster-example`. It points to the primary, identified by the read/write service (`rw` , therefore `cluster-example-rw`).

The `Pooler` resource must live in the same namespace as the Postgres cluster. It consists of a Kubernetes deployment of 3 pods running thelatest stable image of PgBouncer, configured with the `session` pooling mode 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.

EDB Postgres for Kubernetes also creates a secret with the same name as the pooler containing the configuration files used with PgBouncer.

> **API reference**
>
> For details, see `PgBouncerSpec` in the API reference.

Pooler resource lifecycle

`Pooler` resources aren't cluster-managed resources. You create poolers manually when they're needed. You can also deploy multiple poolers per PostgreSQL cluster.

What's important is that the life cycles of the `Cluster` and the `Pooler` resources are currently independent. Deleting the cluster doesn't imply the deletion of the pooler, and vice versa.

> **Important**
>
> Once you know how a pooler works, you have full freedom in terms of possible architectures. You can have clusters without poolers, clusters with a single pooler, or clusters with several poolers, that is, one per application.

> **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.

Security

Any PgBouncer pooler is transparently integrated with EDB Postgres for Kubernetes 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 Authentication).

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

Certificates

By default, a PgBouncer pooler uses the same certificates that are used by the cluster. However, if you provide those certificates, the pooler accepts secrets with the following formats:

1. Basic Auth
2. TLS
3. Opaque

In the Opaque case, it looks for the following specific keys that need to be used:

- tls.crt
- tls.key

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

Authentication

Password-based authentication is the only supported method for clients of PgBouncer in EDB Postgres for Kubernetes.

Internally, the implementation relies on PgBouncer's `auth_user` and `auth_query` options. Specifically, the operator:

- Creates a standard user called `cnp_pooler_pgbouncer` in the PostgreSQL server
- Creates the lookup function in the `postgres` database and grants execution privileges to the cnp_pooler_pgbouncer user (PoLA)
- Issues a TLS certificate for this user
- Sets `cnp_pooler_pgbouncer` as the `auth_user`
- Configures PgBouncer to use the TLS certificate to authenticate `cnp_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 cnp_pooler_pgbouncer WITH
LOGIN;
```

Then, for each application database, grant the permission for `cnp_pooler_pgbouncer` to connect to it:

```
GRANT CONNECT ON DATABASE { database name here } TO
cnp_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
cnp_pooler_pgbouncer;
```

> **Important**
>
> Given that `user_search` is a `SECURITY DEFINER` function, you need to create it through a role with `SUPERUSER` privileges, such as the `postgres` user.

Pod templates

You can take advantage of pod templates specification in the `template` section of a `Pooler` resource. For details, see `PoolerSpec` 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 `quay.io/enterprisedb/pgbouncer`.

This example shows `Pooler` specifying `PodAntiAffinity``:

```yaml
apiVersion: postgresql.k8s.enterprisedb.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:


```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind:
Pooler
metadata:
  name: pooler-example-rw
spec:
  cluster:
    name: cluster-
example
  instances: 3
  type: rw

  template:
    metadata:
      labels:
        app:
pooler
    spec:
      containers:
        - name: pgbouncer
          image: my-
pgbouncer:latest
          resources:
            requests:
              cpu: "0.1"
              memory: 100Mi
            limits:
              cpu: "0.5"
              memory: 500Mi
```

Service Template

Sometimes, your pooler will require some different labels, annotations, or even change the type of the service, you can achieve that by using the `serviceTemplate` field:

```yaml
apiVersion: postgresql.k8s.enterprisedb.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
  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.

High availability (HA)

Because of Kubernetes' deployments, you can configure your pooler to run on a single instance or over multiple pods. The exposed service makes sure that your clients are randomly distributed over the available pods running PgBouncer, which then manages and reuses connections toward the underlying server (if using the `rw` service) or servers (if using the `ro` service with multiple replicas).

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

PgBouncer configuration options

The operator manages most of the configuration options for PgBouncer, allowing you to modify only a subset of them.

> **Warning**
>
> You are responsible for correctly setting the value of each option, as the operator doesn't validate them.

These are the PgBouncer options you can customize, with links to the PgBouncer documentation for each parameter. Unless stated otherwise, the default values are the ones directly set by PgBouncer.

- `application_name_add_host`
- `autodb_idle_timeout`
- `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 EDB Postgres for Kubernetes
- `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 "Monitoring" 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.

Monitoring

The PgBouncer implementation of the `Pooler` comes with a default Prometheus exporter. It makes available several metrics having the `cnp_pgbouncer_` prefix by running:

- `SHOW LISTS` (prefix: `cnp_pgbouncer_lists`)
- `SHOW POOLS` (prefix: `cnp_pgbouncer_pools`)
- `SHOW STATS` (prefix: `cnp_pgbouncer_stats`)

Like the EDB Postgres for Kubernetes 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_*`).

> **Info**
>
> 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 `cnp_pgbouncer` metrics:

```
# HELP cnp_pgbouncer_collection_duration_seconds Collection time duration in seconds
# TYPE cnp_pgbouncer_collection_duration_seconds gauge
cnp_pgbouncer_collection_duration_seconds{collector="Collect.up"} 0.002338805
# HELP cnp_pgbouncer_collection_errors_total Total errors occurred accessing PostgreSQL for metrics.
# TYPE cnp_pgbouncer_collection_errors_total counter
cnp_pgbouncer_collection_errors_total{collector="sql: Scan error on column index 16, name
\"load_balance_hosts\": converting NULL to int is unsupported"} 5
# HELP cnp_pgbouncer_collections_total Total number of times PostgreSQL was accessed for metrics.
# TYPE cnp_pgbouncer_collections_total counter
cnp_pgbouncer_collections_total 5
# HELP cnp_pgbouncer_last_collection_error 1 if the last collection ended with error, 0 otherwise.
# TYPE cnp_pgbouncer_last_collection_error gauge
cnp_pgbouncer_last_collection_error 0
# HELP cnp_pgbouncer_lists_databases Count of databases.
# TYPE cnp_pgbouncer_lists_databases gauge
cnp_pgbouncer_lists_databases 1
# HELP cnp_pgbouncer_lists_dns_names Count of DNS names in the cache.
# TYPE cnp_pgbouncer_lists_dns_names gauge
cnp_pgbouncer_lists_dns_names 0
# HELP cnp_pgbouncer_lists_dns_pending Not used.
# TYPE cnp_pgbouncer_lists_dns_pending gauge
cnp_pgbouncer_lists_dns_pending 0
# HELP cnp_pgbouncer_lists_dns_queries Count of in-flight DNS queries.
# TYPE cnp_pgbouncer_lists_dns_queries gauge
cnp_pgbouncer_lists_dns_queries 0
# HELP cnp_pgbouncer_lists_dns_zones Count of DNS zones in the cache.
# TYPE cnp_pgbouncer_lists_dns_zones gauge
cnp_pgbouncer_lists_dns_zones 0
# HELP cnp_pgbouncer_lists_free_clients Count of free clients.
# TYPE cnp_pgbouncer_lists_free_clients gauge
cnp_pgbouncer_lists_free_clients 49
# HELP cnp_pgbouncer_lists_free_servers Count of free servers.
# TYPE cnp_pgbouncer_lists_free_servers gauge
cnp_pgbouncer_lists_free_servers 0
# HELP cnp_pgbouncer_lists_login_clients Count of clients in login state.
# TYPE cnp_pgbouncer_lists_login_clients gauge
cnp_pgbouncer_lists_login_clients 0
# HELP cnp_pgbouncer_lists_pools Count of pools.
# TYPE cnp_pgbouncer_lists_pools gauge
cnp_pgbouncer_lists_pools 1
# HELP cnp_pgbouncer_lists_used_clients Count of used clients.
# TYPE cnp_pgbouncer_lists_used_clients gauge
cnp_pgbouncer_lists_used_clients 1
# HELP cnp_pgbouncer_lists_used_servers Count of used servers.
# TYPE cnp_pgbouncer_lists_used_servers gauge
cnp_pgbouncer_lists_used_servers 0
# HELP cnp_pgbouncer_lists_users Count of users.
# TYPE cnp_pgbouncer_lists_users gauge
cnp_pgbouncer_lists_users 2
# HELP cnp_pgbouncer_pools_cl_active Client connections that are linked to server connection and can
process queries.
# TYPE cnp_pgbouncer_pools_cl_active gauge
cnp_pgbouncer_pools_cl_active{database="pgbouncer",user="pgbouncer"} 1
# HELP cnp_pgbouncer_pools_cl_active_cancel_req Client connections that have forwarded query cancellations
```

```
to the server and are waiting for the server response.
# TYPE cnp_pgbouncer_pools_cl_active_cancel_req gauge
cnp_pgbouncer_pools_cl_active_cancel_req{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_pools_cl_cancel_req Client connections that have not forwarded query cancellations to
the server yet.
# TYPE cnp_pgbouncer_pools_cl_cancel_req gauge
cnp_pgbouncer_pools_cl_cancel_req{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_pools_cl_waiting Client connections that have sent queries but have not yet got a
server connection.
# TYPE cnp_pgbouncer_pools_cl_waiting gauge
cnp_pgbouncer_pools_cl_waiting{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_pools_cl_waiting_cancel_req Client connections that have not forwarded query
cancellations to the server yet.
# TYPE cnp_pgbouncer_pools_cl_waiting_cancel_req gauge
cnp_pgbouncer_pools_cl_waiting_cancel_req{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_pools_load_balance_hosts Number of hosts not load balancing between hosts
# TYPE cnp_pgbouncer_pools_load_balance_hosts gauge
cnp_pgbouncer_pools_load_balance_hosts{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_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 cnp_pgbouncer_pools_maxwait gauge
cnp_pgbouncer_pools_maxwait{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_pools_maxwait_us Microsecond part of the maximum waiting time.
# TYPE cnp_pgbouncer_pools_maxwait_us gauge
cnp_pgbouncer_pools_maxwait_us{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_pools_pool_mode The pooling mode in use. 1 for session, 2 for transaction, 3 for
statement, -1 if unknown
# TYPE cnp_pgbouncer_pools_pool_mode gauge
cnp_pgbouncer_pools_pool_mode{database="pgbouncer",user="pgbouncer"} 3
# HELP cnp_pgbouncer_pools_sv_active Server connections that are linked to a client.
# TYPE cnp_pgbouncer_pools_sv_active gauge
cnp_pgbouncer_pools_sv_active{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_pools_sv_active_cancel Server connections that are currently forwarding a cancel
request
# TYPE cnp_pgbouncer_pools_sv_active_cancel gauge
cnp_pgbouncer_pools_sv_active_cancel{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_pools_sv_idle Server connections that are unused and immediately usable for client
queries.
# TYPE cnp_pgbouncer_pools_sv_idle gauge
cnp_pgbouncer_pools_sv_idle{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_pools_sv_login Server connections currently in the process of logging in.
# TYPE cnp_pgbouncer_pools_sv_login gauge
cnp_pgbouncer_pools_sv_login{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_pools_sv_tested Server connections that are currently running either
server_reset_query or server_check_query.
# TYPE cnp_pgbouncer_pools_sv_tested gauge
cnp_pgbouncer_pools_sv_tested{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_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 cnp_pgbouncer_pools_sv_used gauge
cnp_pgbouncer_pools_sv_used{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_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 cnp_pgbouncer_pools_sv_wait_cancels gauge
cnp_pgbouncer_pools_sv_wait_cancels{database="pgbouncer",user="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_avg_bind_count Average number of prepared statements readied for execution by
clients and forwarded to PostgreSQL by pgbouncer.
# TYPE cnp_pgbouncer_stats_avg_bind_count gauge
cnp_pgbouncer_stats_avg_bind_count{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_avg_client_parse_count Average number of prepared statements created by clients.
# TYPE cnp_pgbouncer_stats_avg_client_parse_count gauge
```



```
cnp_pgbouncer_stats_avg_client_parse_count{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_avg_query_count Average queries per second in last stat period.
# TYPE cnp_pgbouncer_stats_avg_query_count gauge
cnp_pgbouncer_stats_avg_query_count{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_avg_query_time Average query duration, in microseconds.
# TYPE cnp_pgbouncer_stats_avg_query_time gauge
cnp_pgbouncer_stats_avg_query_time{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_avg_recv Average received (from clients) bytes per second.
# TYPE cnp_pgbouncer_stats_avg_recv gauge
cnp_pgbouncer_stats_avg_recv{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_avg_sent Average sent (to clients) bytes per second.
# TYPE cnp_pgbouncer_stats_avg_sent gauge
cnp_pgbouncer_stats_avg_sent{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_avg_server_parse_count Average number of prepared statements created by
pgbouncer on a server.
# TYPE cnp_pgbouncer_stats_avg_server_parse_count gauge
cnp_pgbouncer_stats_avg_server_parse_count{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_avg_wait_time Time spent by clients waiting for a server, in microseconds
(average per second).
# TYPE cnp_pgbouncer_stats_avg_wait_time gauge
cnp_pgbouncer_stats_avg_wait_time{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_avg_xact_count Average transactions per second in last stat period.
# TYPE cnp_pgbouncer_stats_avg_xact_count gauge
cnp_pgbouncer_stats_avg_xact_count{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_avg_xact_time Average transaction duration, in microseconds.
# TYPE cnp_pgbouncer_stats_avg_xact_time gauge
cnp_pgbouncer_stats_avg_xact_time{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_total_bind_count Total number of prepared statements readied for execution by
clients and forwarded to PostgreSQL by pgbouncer
# TYPE cnp_pgbouncer_stats_total_bind_count gauge
cnp_pgbouncer_stats_total_bind_count{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_total_client_parse_count Total number of prepared statements created by clients.
# TYPE cnp_pgbouncer_stats_total_client_parse_count gauge
cnp_pgbouncer_stats_total_client_parse_count{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_total_query_count Total number of SQL queries pooled by pgbouncer.
# TYPE cnp_pgbouncer_stats_total_query_count gauge
cnp_pgbouncer_stats_total_query_count{database="pgbouncer"} 15
# HELP cnp_pgbouncer_stats_total_query_time Total number of microseconds spent by pgbouncer when actively
connected to PostgreSQL, executing queries.
# TYPE cnp_pgbouncer_stats_total_query_time gauge
cnp_pgbouncer_stats_total_query_time{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_total_received Total volume in bytes of network traffic received by pgbouncer.
# TYPE cnp_pgbouncer_stats_total_received gauge
cnp_pgbouncer_stats_total_received{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_total_sent Total volume in bytes of network traffic sent by pgbouncer.
# TYPE cnp_pgbouncer_stats_total_sent gauge
cnp_pgbouncer_stats_total_sent{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_total_server_parse_count Total number of prepared statements created by
pgbouncer on a server.
# TYPE cnp_pgbouncer_stats_total_server_parse_count gauge
cnp_pgbouncer_stats_total_server_parse_count{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_total_wait_time Time spent by clients waiting for a server, in microseconds.
# TYPE cnp_pgbouncer_stats_total_wait_time gauge
cnp_pgbouncer_stats_total_wait_time{database="pgbouncer"} 0
# HELP cnp_pgbouncer_stats_total_xact_count Total number of SQL transactions pooled by pgbouncer.
# TYPE cnp_pgbouncer_stats_total_xact_count gauge
cnp_pgbouncer_stats_total_xact_count{database="pgbouncer"} 15
# HELP cnp_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 cnp_pgbouncer_stats_total_xact_time gauge
cnp_pgbouncer_stats_total_xact_time{database="pgbouncer"} 0
```

> **Info**
>
> 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:
    matchLabels:
      k8s.enterprisedb.io/poolerName: <POOLER_NAME>
  podMetricsEndpoints:
  - port: metrics
```

Logging

Logs are directly sent to standard output, in JSON format, like in the following example:

```
{
  "level": "info",
  "ts": SECONDS.MICROSECONDS,
  "msg": "record",
  "pipe": "stderr",
  "record": {
    "timestamp": "YYYY-MM-DD HH:MM:SS.MS
UTC",
    "pid": "<PID>",
    "level": "LOG",
    "msg": "kernel file descriptor limit: 1048576 (hard: 1048576); max_client_conn: 100, max expected fd
use: 112"
  }
}
```

Pausing connections

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

1. Closes all active connections toward the PostgreSQL server, after waiting for the queries to complete
2. Pauses any new connection coming from the client

When the `paused` option is reset to `false`, the operator invokes the `RESUME` command in PgBouncer, reopening the taps toward the PostgreSQL service defined in the `Pooler` resource.

> PAUSE
>
> For more information, see `PAUSE` in the PgBouncer documentation.

> 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 `cnp` plugin, and then restoring the `paused` attribute to `false`.

Limitations

Single PostgreSQL cluster

The current implementation of the pooler is designed to work as part of a specific EDB Postgres for Kubernetes cluster (a service). It isn't currently possible to create a pooler that spans multiple clusters.

Controlled configurability

EDB Postgres for Kubernetes 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.

36 Replica clusters

A replica cluster is a EDB Postgres for Kubernetes `Cluster` resource designed to replicate data from another PostgreSQL instance, ideally also managed by EDB Postgres for Kubernetes.

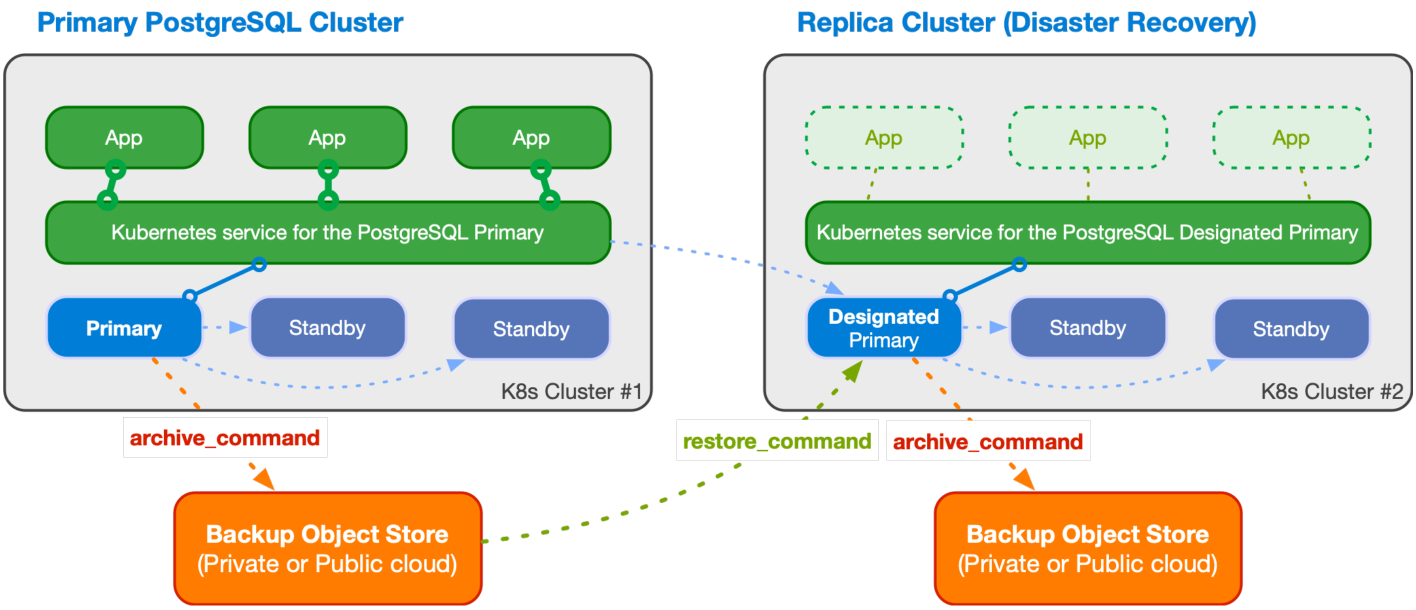
Typically, a replica cluster is deployed in a different Kubernetes cluster in another region. These clusters can be configured to perform cascading replication and can rely on object stores for data replication from the source, as detailed further down.

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 EDB Postgres for Kubernetes cluster across different Kubernetes clusters, typically located in different regions. In EDB Postgres for Kubernetes terms, this is known as a "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 EDB Postgres for Kubernetes terms, this is referred to as a "Standalone Replica Cluster".

For example, the diagram below — taken from the"Architecture" section — illustrates a distributed PostgreSQL topology spanning two Kubernetes clusters, with a symmetric replica cluster primarily serving disaster recovery purposes.



Basic Concepts

EDB Postgres for Kubernetes 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).

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. EDB Postgres for Kubernetes applies any local redefinitions once the cluster is promoted.

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 Cluster".

If you are planning to promote the cluster at some point, EDB Postgres for Kubernetes 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 "Distributed Topology".

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" section.

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 CNP-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.

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.
  - EDB Postgres for Kubernetes 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.
  - EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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.

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.

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.

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 EDB Postgres for Kubernetes 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.

Distributed Topology

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 EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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:
        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" section above and use identical role definitions, including secrets, in all `Cluster` objects participating in the distributed topology.

Demoting a Primary to a Replica Cluster

EDB Postgres for Kubernetes 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. EDB Postgres for Kubernetes 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 `cnp` 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 `cnp` plugin, checking the status of the cluster:

```
kubectl cnp status cluster-eu-south
```


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, EDB Postgres for Kubernetes will initiate the promotion of the replica cluster to a primary cluster. Initially, EDB Postgres for Kubernetes 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 `cnp` plugin to check the status of the cluster:

```
kubectl cnp 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.

Standalone Replica Clusters

> **Important**
>
> Standalone Replica Clusters were previously known as Replica Clusters before the introduction of the Distributed Topology strategy in EDB Postgres for Kubernetes 1.24.

In EDB Postgres for Kubernetes, 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.

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.

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.

```yaml
  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.

```yaml
externalClusters:
- name: <MAIN-CLUSTER>
  connectionParameters:
    host: <MAIN-CLUSTER>-rw.<NAMESPACE>.svc
    user: streaming_replica
    sslmode: verify-full
    dbname: postgres
  sslKey:
    name: <MAIN-CLUSTER>-replication
    key: tls.key
  sslCert:
    name: <MAIN-CLUSTER>-replication
    key: tls.crt
  sslRootCert:
    name: <MAIN-CLUSTER>-ca
    key: ca.crt
```

Example 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.

```yaml
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.

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 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.

Delayed replicas

EDB Postgres for Kubernetes supports the creation of **delayed replicas** through the `.spec.replica.minApplyDelay` option, 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:

```yaml
  # ...
  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.

37 Kubernetes Upgrade and Maintenance

Maintaining an up-to-date Kubernetes cluster is crucial for ensuring optimal performance and security, particularly for self-managed clusters, especially those running on bare metal infrastructure. Regular updates help address technical debt and mitigate business risks, despite the controlled downtimes associated with temporarily removing a node from the cluster for maintenance purposes. For further insights on embracing risk in operations, refer to the "Embracing Risk" chapter from the Site Reliability Engineering book.

Importance of Regular Updates

Updating Kubernetes involves planning and executing maintenance tasks, such as applying security updates to underlying Linux servers, replacing malfunctioning hardware components, or upgrading the cluster to the latest Kubernetes version. These activities are essential for maintaining a robust and secure infrastructure.

Maintenance Operations in a Cluster

Typically, maintenance operations are carried out on one node at a time, following a structured process:

1. eviction of workloads (`drain`): workloads are gracefully moved away from the node to be updated, ensuring a smooth transition.
2. performing the operation: the actual maintenance operation, such as a system update or hardware replacement, is executed.
3. rejoining the node to the cluster (`uncordon`): the updated node is reintegrated into the cluster, ready to resume its responsibilities.

This process requires either stopping workloads for the entire upgrade duration or migrating them to other nodes in the cluster.

Temporary PostgreSQL Cluster Degradation

While the standard approach ensures service reliability and leverages Kubernetes' self-healing capabilities, there are scenarios where operating with a temporarily degraded cluster may be acceptable. This is particularly relevant for PostgreSQL clusters relying on **node-local storage**, where the storage is local to the Kubernetes worker node running the PostgreSQL database. Node-local storage, or simply *local storage*, is employed to enhance performance.

> **Note**
>
> If your database files reside on shared storage accessible over the network, the default self-healing behavior of the operator can efficiently handle scenarios where volumes are reused by pods on different nodes after a drain operation. In such cases, you can skip the remaining sections of this document.

Pod Disruption Budgets

By default, EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes will prevent draining the node where the instance is housed. Additionally, in clusters with 3 or more instances, EDB Postgres for Kubernetes guarantees that only one replica at a time is gracefully shut down during a drain operation.

Each PostgreSQL `Cluster` is equipped with two associated `PodDisruptionBudget` resources - you can easily confirm it with the `kubectl get pdb` command.

Our recommendation is to leave pod disruption budgets enabled for every production Postgres cluster. This can be effortlessly managed by toggling the `.spec.enablePDB` option, as detailed in the API reference.

PostgreSQL Clusters used for Development or Testing

For PostgreSQL clusters used for development purposes, often consisting of a single instance, it is essential to disable pod disruption budgets. Failure to do so will prevent the node hosting that cluster from being drained.

The following example illustrates how to disable pod disruption budgets for a 1-instance development cluster:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name:
dev
spec:
  instances: 1
  enablePDB: false

  storage:
    size:
1Gi
```

This configuration ensures smoother maintenance procedures without restrictions on draining the node during development activities.

Node Maintenance Window

> **Important**
>
> While EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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.

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.

38 EDB Postgres for Kubernetes Plugin

EDB Postgres for Kubernetes provides a plugin for `kubectl` to manage a cluster in Kubernetes. The plugin also works with `oc` in an OpenShift environment.

Install

You can install the `cnp` plugin using a variety of methods.

> **Note**
>
> For air-gapped systems, installation via package managers, using previously downloaded files, may be a good option.

Via the installation script

```
curl -sSfL \
  https://github.com/EnterpriseDB/kubectl-cnp/raw/main/install.sh | \
  sudo sh -s -- -b
/usr/local/bin
```

Using the Debian or RedHat packages

In the releases section of the GitHub repository, you can navigate to any release of interest (pick the same or newer release than your EDB Postgres for Kubernetes operator), and in it you will find an **Assets** section. In that section are pre-built packages for a variety of systems. As a result, you can follow standard practices and instructions to install them in your systems.

Debian packages

For example, let's install the 1.26.0 release of the plugin, for an Intel based 64 bit server. First, we download the right `.deb` file.

```
wget https://github.com/EnterpriseDB/kubectl-cnp/releases/download/v1.26.0/kubectl-
cnp_1.26.0_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 cnp.
(Reading database ... 6688 files and directories currently installed.)
Preparing to unpack kube-plugin.deb ...
Unpacking kubectl-cnp (1.26.0) ...
Setting up kubectl-cnp (1.26.0) ...
```

RPM packages

As in the example for `.rpm` packages, let's install the 1.26.0 release for an Intel 64 bit machine. Note the `--output` flag to provide a file name.

```
curl -L https://github.com/EnterpriseDB/kubectl-cnp/releases/download/v1.26.0/kubectl-
cnp_1.26.0_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:
 cnp                x86_64               1.26.0-1                  @commandline               20 M

Transaction Summary
====================================================================================================
Install  1 Package

Total size: 20 M
Installed size: 78 M
Is this ok [y/N]: y
```

Supported Architectures

EDB Postgres for Kubernetes Plugin is currently built for the following operating system and architectures:

- Linux
  - amd64
  - arm 5/6/7
  - arm64
  - s390x
  - ppc64le
- macOS
  - amd64
  - arm64
- Windows
  - 386
  - amd64
  - arm 5/6/7
  - arm64

Configuring auto-completion

To configure auto-completion for the plugin, a helper shell script needs to be installed into your current PATH. Assuming the latter contains `/usr/local/bin`, this can be done with the following commands:

```
cat > kubectl_complete-cnp <<EOF
#!/usr/bin/env sh

# Call the __complete command passing it all arguments
kubectl cnp __complete "\$@"
EOF

chmod +x kubectl_complete-cnp

# Important: the following command may require superuser permission
sudo mv kubectl_complete-cnp /usr/local/bin
```


> **Important**
>
> The name of the script needs to be exactly the one provided since is used by the kubectl auto-complete process

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-cnp <<EOF
#!/usr/bin/env sh

# Call the __complete command passing it all
arguments
kubectl cnp __complete
"\$@"
EOF

chmod +x kubectl_complete-cnp

# Important: the following command may require superuser
permission
sudo mv kubectl_complete-cnp
/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

Use

Once the plugin is installed and deployed, you can start using it like this:

```
kubectl cnp <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.

Generation of installation manifests

The `cnp` 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 cnp install generate --
help
```

The main options are:

- `-n` : specifies the namespace in which to install the operator (default: `postgresql-operator-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 cnp 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 PG4K 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

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 cnp status
sandbox
```

```
output
Cluster Summary
Name:                default/sandbox
System ID:           7423474350493388827
PostgreSQL Image:    docker.enterprisedb.com/k8s_enterprise/edb-postgres-extended:16.4
Primary instance:    sandbox-1
Primary start time:  2024-10-08 18:31:57 +0000 UTC (uptime 1m14s)
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  1.26.0           k8s-eu-worker
sandbox-2  0/604DE38    Standby (async)   OK      BestEffort  1.26.0           k8s-eu-worker2
sandbox-3  0/604DE38    Standby (async)   OK      BestEffort  1.26.0           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 cnp status sandbox --
verbose
```

```
output
Cluster Summary
Name:                default/sandbox
System ID:           7423474350493388827
PostgreSQL Image:    docker.enterprisedb.com/k8s_enterprise/edb-postgres-extended: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

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  1.26.0           k8s-eu-worker
sandbox-2  0/6053720    Standby (async)   OK      BestEffort  1.26.0           k8s-eu-worker2
sandbox-3  0/6053720    Standby (async)   OK      BestEffort  1.26.0           k8s-eu-worker
```

With an additional `-v` (e.g. `kubectl cnp status sandbox -v -v`), you can also view PostgreSQL configuration, HBA settings, and certificates.

The command also supports output in `yaml` and `json` format.

Promote

The meaning of this command is to `promote` a pod in the cluster to primary, so you can start with maintenance work or test a switch-over situation in your cluster

```
kubectl cnp promote cluster-example cluster-example-
2
```

Or you can use the instance node number to promote

```
kubectl cnp promote cluster-example
2
```

Certificates

Clusters created using the EDB Postgres for Kubernetes 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 cnp certificate cluster-cert --cnp-cluster cluster-example --cnp-user
appuser
```

After the secret it's created, you can get it using `kubectl`

```
kubectl get secret cluster-
cert
```

And the content of the same in plain text using the following commands:

```
kubectl get secret cluster-cert -o json | jq -r '.data | map(@base64d) | .
[]'
```

Restart

The `kubectl cnp 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 cnp restart
[clusterName]

# this command will restart a single instance, according to the policy
above
kubectl cnp restart [clusterName]
[pod]
```

If the in-place restart is requested but the change cannot be applied without a switchover, the switchover will take precedence over the in-place restart. A common case for this will be a minor upgrade of PostgreSQL image.

> **Note**
>
> If you want ConfigMaps and Secrets to be **automatically** reloaded by instances, you can add a label with key `k8s.enterprisedb.io/reload` to it.

Reload

The `kubectl cnp 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 cnp reload
[cluster_name]
```

Maintenance

The `kubectl cnp 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 cnp maintenance set --all-
namespaces
```

And you'll have the list of all the cluster to update

```
output
The following are the new values for the clusters
Namespace  Cluster Name     Maintenance  reusePVC
---------  ------------     -----------  --------
default    cluster-example  true         false
default    pg-backup        true         false
test       cluster-example  true         false
Do you want to proceed? [y/n]: y
```

Report

The `kubectl cnp report` command bundles various pieces of information into a ZIP file. It aims to provide the needed context to debug problems with clusters in production.

It has two sub-commands: `operator` and `cluster` .

report Operator

The `operator` sub-command requests the operator to provide information regarding the operator deployment, configuration and events.

> **Important**
>
> All confidential information in Secrets and ConfigMaps is REDACTED. The Data map will show the**keys** but the values will be empty. The flag `-S` / `--stopRedaction` will defeat the redaction and show the values. Use only at your own risk, this will share private data.

> **Note**
>
> By default, operator logs are not collected, but you can enable operator log collection with the `--logs` flag

- **deployment information**: the operator Deployment and operator Pod
- **configuration**: the Secrets and ConfigMaps in the operator namespace
- **events**: the Events in the operator namespace
- **webhook configuration**: the mutating and validating webhook configurations
- **webhook service**: the webhook service
- **logs**: logs for the operator Pod (optional, off by default) in JSON-lines format

The command will generate a ZIP file containing various manifest in YAML format (by default, but settable to JSON with the `-o` flag). Use the `-f` flag to name a result file explicitly. If the `-f` flag is not used, a default time-stamped filename is created for the zip file.

> **Note**
>
> The report plugin obeys `kubectl` conventions, and will look for objects constrained by namespace. The PG4K Operator will generally not be installed in the same namespace as the clusters. E.g. the default installation namespace is postgresql-operator-system

```
kubectl cnp report operator -n
<namespace>
```

results in

```
output
Successfully written report to "report_operator_<TIMESTAMP>.zip" (format: "yaml")
```

With the `-f` flag set:

```
kubectl cnp report operator -n <namespace> -f
reportRedacted.zip
```

Unzipping the file will produce a time-stamped top-level folder to keep the directory tidy:

```
unzip reportRedacted.zip
```

will result in:


```output
Archive:  reportRedacted.zip
   creating: report_operator_<TIMESTAMP>/
   creating: report_operator_<TIMESTAMP>/manifests/
  inflating: report_operator_<TIMESTAMP>/manifests/deployment.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/operator-pod.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/events.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/validating-webhook-configuration.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/mutating-webhook-configuration.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/webhook-service.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/postgresql-operator-ca-secret.yaml
  inflating: report_operator_<TIMESTAMP>/manifests/postgresql-operator-webhook-cert.yaml
```

If you activated the `--logs` option, you'd see an extra subdirectory:

```output
Archive:  report_operator_<TIMESTAMP>.zip
  <snipped …>
  creating: report_operator_<TIMESTAMP>/operator-logs/
  inflating: report_operator_<TIMESTAMP>/operator-logs/postgresql-operator-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.

```output
====== Begin of Previous Log =====
2023-03-28T12:56:41.251711811Z {"level":"info","ts":"2023-03-28T12:56:41Z","logger":"setup","msg":"Starting EDB Postgres for Kubernetes Operator","version":"1.26.0","build":{"Version":"1.26.0+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 EDB Postgres for Kubernetes Operator","version":"1.26.0","build":{"Version":"1.26.0+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 postgresql-operator-ca-
secret.yaml
```



```yaml
data:
  ca.crt: ""
  ca.key: ""
metadata:
  creationTimestamp: "2022-03-22T10:42:28Z"
  managedFields:
  - apiVersion: v1
    fieldsType:
FieldsV1
    fieldsV1:
```

With the `-S` (`--stopRedaction`) option activated, secrets are shown:

```
kubectl cnp report operator -n <namespace> -f reportNonRedacted.zip -
S
```

You'll get a reminder that you're about to view confidential information:

```
output
WARNING: secret Redaction is OFF. Use it with caution
Successfully written report to "reportNonRedacted.zip" (format: "yaml")
```

```
unzip
reportNonRedacted.zip
head postgresql-operator-ca-
secret.yaml
```

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

report Cluster

The `cluster` sub-command gathers the following:

- **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 cnp report cluster <clusterName>
[flags]
```

Note that, unlike the `operator` sub-command, for the `cluster` sub-command you need to provide the cluster name, and very likely the namespace, unless the cluster is in the default one.

```
kubectl cnp report cluster example -f report.zip -n
example_namespace
```

and then:

```
unzip report.zip
```

```
output
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 cnp report cluster example -n example_namespace --
logs
```

will result in:

```
output
Successfully written report to "report_cluster_example_<TIMESTAMP>.zip" (format: "yaml")
```

```
unzip report_cluster_<TIMESTAMP>.zip
```

```
output
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
```

OpenShift support

The `report operator` directive will detect automatically if the cluster is running on OpenShift, and will get the Cluster Service Version and the Install Plan, and add them automatically to the zip under the `openshift` sub-folder.

> **Note**
>
> the namespace becomes very important on OpenShift. The default namespace for OpenShift in CNP is "openshift-operators". Many (most) clients will use a different namespace for the CNP operator.


```
kubectl cnp report operator -n openshift-
operators
```

results in

```
Successfully written report to "report_operator_<TIMESTAMP>.zip" (format:
"yaml")
```

You can find the OpenShift-related files in the `openshift` sub-folder:

```
unzip
report_operator_<TIMESTAMP>.zip
cd
report_operator_<TIMESTAMP>/
cd openshift
head clusterserviceversions.yaml
```


```
apiVersion:
operators.coreos.com/v1alpha1
items:
- apiVersion:
operators.coreos.com/v1alpha1
  kind:
ClusterServiceVersion
  metadata:
    annotations:
      alm-examples: |-

[

{
            "apiVersion": "postgresql.k8s.enterprisedb.io/v1",
```


Logs

The `kubectl cnp logs` command allows to follow the logs of a collection of pods related to EDB Postgres for Kubernetes in a single go.

It has at the moment one available sub-command: `cluster`.

Cluster logs

The `cluster` sub-command gathers all the pod logs for a cluster in a single stream or file. This means that you can get all the pod logs in a single terminal window, with a single invocation of the command.

As in all the cnp plugin sub-commands, you can get instructions and help with the `-h` flag:

`kubectl cnp 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 `cnp` 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 cnp logs cluster <clusterName>
[flags]
```

Using the `-f` option to follow:

```
kubectl cnp report cluster cluster-example -
f
```

Using `--tail` option to display 3 lines from each pod and the `-f` option to follow:

```
kubectl cnp report cluster cluster-example -f --tail
3
```

```
output
{"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 cnp logs cluster cluster-example --output my-cluster.log

Successfully written logs to "my-cluster.log"
```

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 cnp logs cluster`, as shown in the following example:

```
$ kubectl cnp logs cluster cluster-example | kubectl cnp logs pretty
2024-10-15T17:35:00.336 INFO     cluster-example-1 instance-manager Starting EDB Postgres for Kubernetes
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 PG4K logs in JSON format, such as `stern`, or `kubectl logs`, as in the following example:

```
$ kubectl logs cluster-example-1 | kubectl cnp logs pretty
2024-10-15T17:35:00.336 INFO     cluster-example-1 instance-manager Starting EDB Postgres for Kubernetes
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 cnp logs cluster cluster-example | kubectl cnp 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 cnp logs cluster cluster-example | kubectl cnp logs pretty --sorting-group-size=3
2024-10-15T17:35:20.426 INFO     cluster-example-2 instance-manager Starting EDB Postgres for Kubernetes
Instance Manager
2024-10-15T17:35:20.426 INFO     cluster-example-2 instance-manager Checking for free disk space for WALs
before starting PostgreSQL
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.

> **Info**
>
> You can also increase the verbosity of the log by adding more `-v` options.

Destroy

The `kubectl cnp 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 `k8s.enterprisedb.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 cnp destroy [CLUSTER_NAME]
[INSTANCE_ID]
```

The following example removes the `cluster-example-2` pod and the associated PVCs:

```
kubectl cnp destroy cluster-example
2
```

Cluster hibernation

Sometimes you may want to suspend the execution of a EDB Postgres for Kubernetes `Cluster` while retaining its data, then resume its activity at a later time. We've called this feature **cluster hibernation**.

Hibernation is only available via the `kubectl cnp hibernate [on|off]` commands.

Hibernating a EDB Postgres for Kubernetes cluster means destroying all the resources generated by the cluster, except the PVCs that belong to the PostgreSQL primary instance.

You can hibernate a cluster with:

```
kubectl cnp hibernate on <cluster-
name>
```

This will:

1. shutdown every PostgreSQL instance
2. detach the PVCs containing the data of the primary instance, and annotate them with the latest database status and the latest cluster configuration
3. delete the `Cluster` resource, including every generated resource - except the aforementioned PVCs

When hibernated, a EDB Postgres for Kubernetes cluster is represented by just a group of PVCs, in which the one containing the `PGDATA` is annotated with the latest available status, including content from `pg_controldata`.

> **Warning**
>
> A cluster having fenced instances cannot be hibernated, as fencing is part of the hibernation procedure too.

In case of error the operator will not be able to revert the procedure. You can still force the operation with:

```
kubectl cnp hibernate on cluster-example --
force
```

A hibernated cluster can be resumed with:

```
kubectl cnp hibernate off <cluster-
name>
```

Once the cluster has been hibernated, it's possible to show the last configuration and the status that PostgreSQL had after it was shut down. That can be done with:

```
kubectl cnp hibernate status <cluster-
name>
```

Benchmarking the database with pgbench

Pgbench can be run against an existing PostgreSQL cluster with following command:

```
kubectl cnp pgbench <cluster-name> -- --time 30 --client 1 --jobs
1
```

Refer to the Benchmarking pgbench section for more details.

Benchmarking the storage with fio

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

```
kubectl cnp fio <fio-job-name> -n
<namespace>
```

Refer to the Benchmarking fio section for more details.

Requesting a new physical backup

The `kubectl cnp backup` command requests a new physical backup for an existing Postgres cluster by creating a new `Backup` resource.

> **Info**
>
> From release 1.21, the `backup` command accepts a new flag, `-m` to specify the backup method. To request a backup using volume snapshots, set `-m volumeSnapshot`

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

```
kubectl cnp backup
[cluster_name]
```

or, if using volume snapshots:

```
kubectl cnp backup [cluster_name] -m
volumeSnapshot
```

The created backup will be named after the request time:

```
$ kubectl cnp 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 "Backup" section contains more information about the configuration settings.

Launching psql

The `kubectl cnp psql` command starts a new PostgreSQL interactive front-end process (psql) connected to an existing Postgres cluster, as if you were running it from the actual pod. This means that you will be using the `postgres` user.

> **Important**
>
> As you will be connecting as `postgres` user, in production environments this method should be used with extreme care, by authorized personnel only.

```
$ kubectl cnp psql cluster-example

psql (17.0 (Debian 17.0-1.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 cnp psql --replica cluster-example

psql (17.0 (Debian 17.0-1.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.

> **Note**
>
> When connecting to instances running on OpenShift, you must explicitly pass a username to the `psql` command, because of a security measure built into OpenShift:
>
> ```
> kubectl cnp psql cluster-example -- -U
> postgres
> ```

Snapshotting a Postgres cluster

> **Warning**
>
> The `kubectl cnp snapshot` command has been removed. Please use the `backup` command to request backups using volume snapshots.

Using pgAdmin4 for evaluation/demonstration purposes only

pgAdmin stands as the most popular and feature-rich open-source administration and development platform for PostgreSQL. For more information on the project, please refer to the official documentation.

Given that the pgAdmin Development Team maintains official Docker container images, you can install pgAdmin in your environment as a standard Kubernetes deployment.

> **Important**
>
> Deployment of pgAdmin in Kubernetes production environments is beyond the scope of this document and, more broadly, of the EDB Postgres for Kubernetes project.

However, **for the purposes of demonstration and evaluation**, EDB Postgres for Kubernetes offers a suitable solution. The `cnp` 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`.

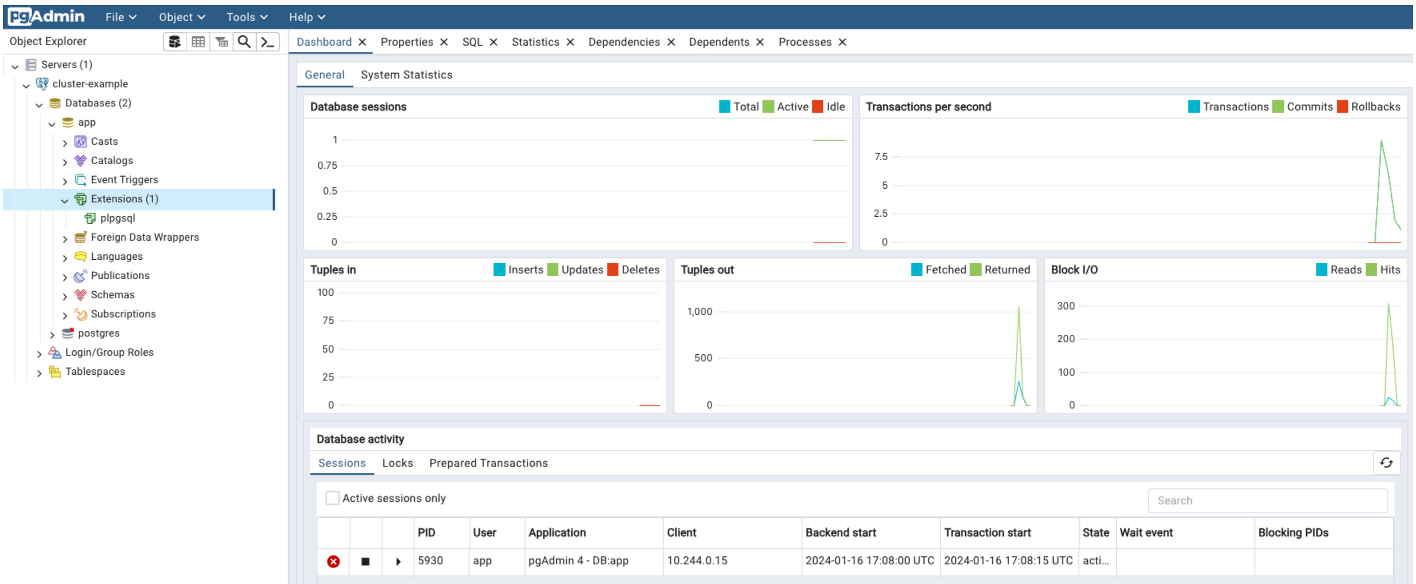
For example, you can install a demo deployment of pgAdmin4 for the `cluster-example` cluster as follows:

```
kubectl cnp pgadmin4 cluster-
example
```

This command will produce:

```output
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.



As usual, you can use the `--dry-run` option to generate the YAML file:

```
kubectl cnp 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 cnp pgadmin4 --mode desktop cluster-
example
```

After concluding your demo, ensure the termination of the pgAdmin deployment by executing:

```
kubectl cnp pgadmin4 --dry-run cluster-example | kubectl delete -f
-
```

> **Warning**
>
> Never deploy pgAdmin in production using the plugin.

Logical Replication Publications

The `cnp publication` command group is designed to streamline the creation and removal of PostgreSQL logical replication publications. Be aware that these commands are primarily intended for assisting in the creation of logical replication publications, particularly on remote PostgreSQL databases.

> **Warning**
>
> It is crucial to have a solid understanding of both the capabilities and limitations of PostgreSQL's native logical replication system before using these commands. In particular, be mindful of the logical replication restrictions.

Creating a new publication

To create a logical replication publication, use the `cnp publication create` command. The basic structure of this command is as follows:

```
kubectl cnp 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 cnp 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 cnp 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 cnp 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.

> **Info**
>
> There are two sample files that have been provided for illustration and inspiration: logical-source and logical-destination.

Dropping a publication

The `cnp 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 cnp publication drop
\
  --publication <PUBLICATION_NAME> \
  [--external-cluster <EXTERNAL_CLUSTER>]
  <LOCAL_CLUSTER> [options]
```

To access further details and precise instructions, use the following command:

```
kubectl cnp publication drop --
help
```

Logical Replication Subscriptions

The `cnp subscription` command group is a dedicated set of commands designed to simplify the creation and removal of PostgreSQL logical replication subscriptions. These commands are specifically crafted to aid in the establishment of logical replication subscriptions, especially when dealing with remote PostgreSQL databases.

> **Warning**
>
> Before using these commands, it is essential to have a comprehensive understanding of both the capabilities and limitations of PostgreSQL's native logical replication system. In particular, be mindful of the logical replication restrictions.

In addition to subscription management, we provide a helpful command for synchronizing all sequences from the source cluster. While its applicability may vary, this command can be particularly useful in scenarios involving major upgrades or data import from remote servers.

Creating a new subscription

To create a logical replication subscription, use the `cnp subscription create` command. The basic structure of this command is as follows:

```
kubectl cnp 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 cnp 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 cnp 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.

> **Info**
>
> There are two sample files that have been provided for illustration and inspiration: logical-source and logical-destination.

Dropping a subscription

The `cnp subscription drop` command seamlessly complements the `create` command. You can drop a `SUBSCRIPTION` with the following command structure:

```
kubectl cnp subcription drop
\
  --subscription <SUBSCRIPTION_NAME> \
  <LOCAL_CLUSTER> [options]
```

To access further details and precise instructions, use the following command:

```
kubectl cnp subscription drop --
help
```

Synchronizing sequences

One notable constraint of PostgreSQL logical replication, implemented through publications and subscriptions, is the lack of sequence synchronization. This becomes particularly relevant when utilizing logical replication for live database migration, especially to a higher version of PostgreSQL. A crucial step in this process involves updating sequences before transitioning applications to the new database (*cutover*).

To address this limitation, the `cnp 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 cnp subscription sync-sequences
\
  --subscription <SUBSCRIPTION_NAME> \
  <LOCAL_CLUSTER>
```

For comprehensive details and specific instructions, utilize the following command:

```
kubectl cnp 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 cnp subscription sync-sequences destination-cluster
\
  --subscription=app
```

> **Warning**
>
> Prioritize testing subscriptions in a non-production environment to guarantee their effectiveness and detect any potential issues before deploying them in a production setting.

Integration with K9s

The `cnp` plugin can be easily integrated in K9s, a popular terminal-based UI to interact with Kubernetes clusters.

See `k9s/plugins.yml` for details.

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

Command	Resource Permissions
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

[^1]: The permissions are cluster scope ClusterRole resources.

///Footnotes Go Here///

Additionally, assigning the `list` permission on the `clusters` will enable autocompletion for multiple commands.

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:

```yaml
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: cnp-log
rules:
  - verbs:
      -
get
    apiGroups:
      - postgresql.k8s.enterprisedb.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: cnp-status
rules:
  - verbs:
      -
get
    apiGroups:
      - postgresql.k8s.enterprisedb.io
    resources:
      -
clusters
  - verbs:
      - list
    apiGroups:
      - ''
    resources:
      - pods
  - verbs:
      -
create
    apiGroups:
      - ''
    resources:
      - pods/exec
  - verbs:
      -
create
    apiGroups:
      - ''
    resources:
      - 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.

39 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.

> **Info**
>
> "Fast" mode does not wait for PostgreSQL clients to disconnect and will terminate an online backup in progress. All active transactions are rolled back and clients are forcibly disconnected, then the server is shut down. "Immediate" mode will abort all PostgreSQL server processes immediately, without a clean shutdown.

RTO and RPO impact

Failover may result in the service being impacted (RTO) and/or data being lost (RPO):

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 "Instance Manager" section when explaining the switchover process, the `.spec.switchoverDelay` option affects the RPO and RTO of your PostgreSQL database. Setting it to a low value, might favor RTO over RPO but lead to data loss at cluster level and/or backup level. On the contrary, setting it to a high value, might remove the risk of data loss while leaving the cluster without an active primary for a longer time during the switchover.

Delayed failover

As anticipated above, the `.spec.failoverDelay` option allows you to delay the start of the failover procedure by a number of seconds after the primary has been detected to be unhealthy. By default, this setting is set to `0`, triggering the failover procedure immediately.

Sometimes failing over to a new primary can be more disruptive than waiting for the primary to come back online. This is especially true of network disruptions where multiple tiers are affected (i.e., downstream logical subscribers) or when the time to perform the failover is longer than the expected outage.

Enabling a new configuration option to delay failover provides a mechanism to prevent premature failover for short-lived network or node instability.

40 Fencing

Fencing in EDB Postgres for Kubernetes is the ultimate process of protecting the data in one, more, or even all instances of a PostgreSQL cluster when they appear to be malfunctioning. When an instance is fenced, the PostgreSQL server process (`postmaster`) is guaranteed to be shut down, while the pod is kept running. This makes sure that, until the fence is lifted, data on the pod is not modified by PostgreSQL and that the file system can be investigated for debugging and troubleshooting purposes.

How to fence instances

In EDB Postgres for Kubernetes you can fence:

- a specific instance
- a list of instances
- an entire Postgres `Cluster`

Fencing is controlled through the content of the `k8s.enterprisedb.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:

- `k8s.enterprisedb.io/fencedInstances: '["cluster-example-1"]'` will fence just the `cluster-example-1` instance

- `k8s.enterprisedb.io/fencedInstances: '["cluster-example-1","cluster-example-2"]'` will fence the `cluster-example-1` and `cluster-example-2` instances

- `k8s.enterprisedb.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 cnp fencing on` subcommand:

```
# to fence only one instance
kubectl cnp fencing on cluster-example 1

# to fence all the instances in a Cluster
kubectl cnp fencing on cluster-example "*"
```

Here is an example of a `Cluster` with an instance that was previously fenced:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
    annotations:
      k8s.enterprisedb.io/fencedInstances: '["cluster-example-1"]'
[...]
```

How to lift fencing

Fencing can be lifted by clearing the annotation, or set it to a different value.

As for fencing, this can be done either manually with `kubectl annotate`, or using the `kubectl cnp fencing` subcommand as follows:

```
# to lift the fencing only for one instance
# N.B.: at the moment this won't work if the whole cluster was fenced previously,
#       in that case you will have to manually set the annotation as explained above
kubectl cnp fencing off cluster-example 1

# to lift the fencing for all the instances in a Cluster
kubectl cnp fencing off cluster-example "*"
```

How fencing works

Once an instance is set for fencing, the procedure to shut down the `postmaster` process is initiated, identical to the one of the switchover. This consists of an initial fast shutdown with a timeout set to `.spec.switchoverDelay`, followed by an immediate shutdown. Then:

- the Pod will be kept alive
- the Pod won't be marked as *Ready*
- all the changes that don't require the Postgres instance to be up will be reconciled, including:
  - configuration files
  - certificates and all the cryptographic material
- metrics will not be collected, except `cnp_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`.

41 Declarative hibernation

EDB Postgres for Kubernetes 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.

Hibernation

To hibernate a cluster, set the `k8s.enterprisedb.io/hibernation=on` annotation:

```
$ kubectl annotate cluster <cluster-name> --overwrite
k8s.enterprisedb.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 `k8s.enterprisedb.io/hibernation` condition:

```
$ kubectl get cluster <cluster-name> -o "jsonpath={.status.conditions[?
(.type==\"k8s.enterprisedb.io/hibernation\")]}"

{
        "lastTransitionTime":"2023-03-05T16:43:35Z",
        "message":"Cluster has been
hibernated",
        "reason":"Hibernated",
        "status":"True",
        "type":"k8s.enterprisedb.io/hibernation"
}
```

The hibernation status can also be read with the `status` sub-command of the `cnp` plugin for `kubectl`:

```
$ kubectl cnp status <cluster-
name>
Cluster Summary
Name:               cluster-
example
Namespace:          default
PostgreSQL Image:
quay.io/enterprisedb/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
[..]
```

Rehydration

To rehydrate a cluster, either set the `k8s.enterprisedb.io/hibernation` annotation to `off`:

```
$ kubectl annotate cluster <cluster-name> --overwrite k8s.enterprisedb.io/hibernation=off
```

Or, just unset it altogether:

```
$ kubectl annotate cluster <cluster-name> k8s.enterprisedb.io/hibernation-
```

The Pods will be recreated and the cluster will resume operation.

42 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 EDB Postgres for Kubernetes.

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` project in GitHub
- The `postgis-containers` Container Registry in GitHub

Additionally, EDB provides container images for EDB Postgres Advanced Server that include PostGIS and makes them available in the official registry on Quay.io with the `-postgis` suffix.

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 EDB Postgres for Kubernetes 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" section 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

> **Info**
>
> 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?" FAQ or the "Database import" section.

Create a new PostgreSQL cluster with PostGIS

Let's suppose you want to create a new PostgreSQL 14 cluster with PostGIS 3.2.

The first step is to ensure you use the right PostGIS container image for the operand, and properly set the `.spec.imageName` option in the `Cluster` resource.

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

> **Warning**
>
> Please consider that, although convention over configuration applies in EDB Postgres for Kubernetes, you should spend time configuring and tuning your system for production. Also the `imageName` in the example below deliberately points to the latest available image for PostgreSQL 14 - you should use a specific image name or, preferably, the SHA256 digest for true immutability.

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: postgis-
example
spec:
  instances: 3
  imageName: ghcr.io/enterprisedb/postgresql:16-postgis
  bootstrap:
    initdb:
      postInitTemplateSQL:
        - CREATE EXTENSION
postgis;
        - CREATE EXTENSION
postgis_topology;
        - CREATE EXTENSION
fuzzystrmatch;
        - CREATE EXTENSION
postgis_tiger_geocoder;

  storage:
    size:
1Gi
```

The example relies on the `postInitTemplateSQL` option which executes a list of queries against the `template1` database, before the actual creation of the application database (called `app`). This means that, once you have applied the manifest and the cluster is up, you will have the above extensions installed in both the template database and the application database, ready for use.

> **Info**
>
> Take some time and look at the available options in `.spec.bootstrap.initdb` from the API reference, such as `postInitApplicationSQL` .

You can easily verify the available version of PostGIS that is in the container, by connecting to the `app` database (you might obtain different values from the ones in this document):


```
$ kubectl exec -ti postgis-example-1 -- psql app
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
psql (16.0 (Debian 16.0-1.pgdg110+1))
Type "help" for help.

app=# SELECT * FROM pg_available_extensions WHERE name ~ '^postgis' ORDER BY 1;
           name           | default_version | installed_version |                          comment
--------------------------+-----------------+-------------------+------------------------------------------
-------------
 postgis                  | 3.2.2           | 3.2.2             | PostGIS geometry and geography spatial
types and functions
 postgis-3                | 3.2.2           |                   | PostGIS geometry and geography spatial
types and functions
 postgis_raster           | 3.2.2           |                   | PostGIS raster types and functions
 postgis_raster-3         | 3.2.2           |                   | PostGIS raster types and functions
 postgis_sfcgal           | 3.2.2           |                   | PostGIS SFCGAL functions
 postgis_sfcgal-3         | 3.2.2           |                   | PostGIS SFCGAL functions
 postgis_tiger_geocoder   | 3.2.2           | 3.2.2             | PostGIS tiger geocoder and reverse
geocoder
 postgis_tiger_geocoder-3 | 3.2.2           |                   | PostGIS tiger geocoder and reverse
geocoder
 postgis_topology         | 3.2.2           | 3.2.2             | PostGIS topology spatial types and
functions
 postgis_topology-3       | 3.2.2           |                   | PostGIS topology spatial types and
functions
(10 rows)
```

The next step is to verify that the extensions listed in the `postInitTemplateSQL` section have been correctly installed in the `app` database.

```
app=# \dx
                                        List of installed extensions
          Name          | Version |   Schema   |                        Description
------------------------+---------+------------+------------------------------------------------------------
 fuzzystrmatch          | 1.1     | public     | determine similarities and distance between strings
 plpgsql                | 1.0     | pg_catalog | PL/pgSQL procedural language
 postgis                | 3.2.2   | public     | PostGIS geometry and geography spatial types and functions
 postgis_tiger_geocoder | 3.2.2   | tiger      | PostGIS tiger geocoder and reverse geocoder
 postgis_topology       | 3.2.2   | topology   | PostGIS topology spatial types and functions
(5 rows)
```

Finally:

```
app=# SELECT postgis_full_version();
                                                                            postgis_full_version
-------------------------------------------------------------------------------------------------------------
------------------------------------------------------------
 POSTGIS="3.2.2 628da50" [EXTENSION] PGSQL="140" GEOS="3.9.0-CAPI-1.16.2" PROJ="7.2.1" LIBXML="2.9.10"
LIBJSON="0.15" LIBPROTOBUF="1.3.3" WAGYU="0.5.0 (Internal)" TOPOLOGY
(1 row)
```


43 Container Image Requirements

The EDB Postgres for Kubernetes 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 cnp status`)

Important

Only PostgreSQL versions officially supported by PGDG are allowed.

Info

Barman Cloud executables are no longer required in EDB Postgres for Kubernetes. The recommended approach is to use the dedicated Barman Cloud Plugin.

No entry point or command is required in the image definition. EDB Postgres for Kubernetes automatically overrides it with its instance manager.

Warning

EDB Postgres for Kubernetes only supports **Primary with multiple/optional Hot Standby Servers architecture** for PostgreSQL application container images.

EDB provides and maintains public PostgreSQL container images that are fully compatible with EDB Postgres for Kubernetes. These images are published on quay.io.

Image Tag Requirements

To ensure the operator makes informed decisions, it must accurately detect the PostgreSQL major version. This detection can occur in two ways:

1. Utilizing the `major` field of the `imageCatalogRef`, if defined.
2. Auto-detecting the major version from the image tag of the `imageName` if not explicitly specified.

For auto-detection to work, the image tag must adhere to a specific format. It should commence with a valid PostgreSQL major version number (e.g., 15.6 or 16), optionally followed by a dot and the patch level.

Following this, the tag can include any character combination valid and accepted in a Docker tag, preceded by a dot, an underscore, or a minus sign.

Examples of accepted image tags:

- `12.1`
- `13.3.2.1-1`
- `13.4`

- `14`
- `15.5-10`
- `16.0`

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

> **Note**
>
> Image tag requirements do no apply for images defined in a catalog.

44 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.

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes more reliable, and easier to understand.

PVC resizing

This is a well known limitation of `StatefulSet`: it does not support resizing PVCs. This is inconvenient for a database. Resizing volumes requires convoluted workarounds.

In contrast, EDB Postgres for Kubernetes leverages the configured storage class to manage the underlying PVCs directly, and can handle PVC resizing if the storage class supports it.

Primary Instances versus Replicas

The `StatefulSet` controller is designed to create a set of Pods from just one template. Given that we use one `Pod` per PostgreSQL instance, we have two kinds of Pods:

1. primary instance (only one)
2. replicas (multiple, optional)

This difference is relevant when deciding the correct deployment strategy to execute for a given operation.

Some operations should be performed on the replicas first, and then on the primary, but only after an updated replica is promoted as the new primary. For example, when you want to apply a different PostgreSQL image version, or when you increase configuration parameters like `max_connections` (which are treated specially by PostgreSQL because EDB Postgres for Kubernetes uses hot standby replicas).

While doing that, EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes will:

- apply the new setting to the primary instance
- restart it
- apply the new setting on the replicas

The `StatefulSet` controller, being application-independent, can't incorporate this behavior, which is specific to PostgreSQL's native replication technology.

Coherence of PVCs

PostgreSQL instances can be configured to work with multiple PVCs: this is how WAL storage can be separated from `PGDATA`.

The two data stores need to be coherent from the PostgreSQL point of view, as they're used simultaneously. If you delete the PVC corresponding to the WAL storage of an instance, the PVC where `PGDATA` is stored will not be usable anymore.

This behavior is specific to PostgreSQL and is not implemented in the `StatefulSet` controller - the latter not being application specific.

After the user dropped a PVC, a `StatefulSet` would just recreate it, leading to a corrupted PostgreSQL instance.

EDB Postgres for Kubernetes would instead classify the remaining PVC as unusable, and start creating a new pair of PVCs for another instance to join the cluster correctly.

Local storage, remote storage, and database size

Sometimes you need to take down a Kubernetes node to do an upgrade. After the upgrade, depending on your upgrade strategy, the updated node could go up again, or a new node could replace it.

Supposing the unavailable node was hosting a PostgreSQL instance, depending on your database size and your cloud infrastructure, you may prefer to choose one of the following actions:

1. drop the PVC and the Pod residing on the downed node; create a new PVC cloning the data from another PVC; after that, schedule a Pod for it
2. drop the Pod, schedule the Pod in a different node, and mount the PVC from there
3. leave the Pod and the PVC as they are, and wait for the node to be back up.

The first solution is practical when your database size permits, allowing you to immediately bring back the desired number of replicas.

The second solution is only feasible when you're not using the storage of the local node, and re-mounting the PVC in another host is possible in a reasonable amount of time (which only you and your organization know).

The third solution is appropriate when the database is big and uses local node storage for maximum performance and data durability.

The EDB Postgres for Kubernetes controller implements all these strategies so that the user can select the preferred behavior at the cluster level (read the "Kubernetes upgrade" section for details).

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

45 Networking

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes on existing infrastructure, networking restrictions might be in place that could impair the communication of the operator with PostgreSQL clusters. Specifically, existing Network Policies might restrict certain types of traffic.

Or, you might be interested in adding network policies in your environment for increased security. As mentioned in the security document, please ensure the operator can reach every cluster pod on ports 8000 and 5432, and that pods can connect to each other.

Cross-namespace network policy for the operator

Following the quickstart guide or using helm chart for deployment will install the operator in a dedicated namespace (`postgresql-operator-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 `networkpolicy-example.yaml` 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.

Cross-cluster networking

While bootstrapping 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.

46 Benchmarking

The CNP kubectl plugin provides an easy way for benchmarking a PostgreSQL deployment in Kubernetes using EDB Postgres for Kubernetes.

Benchmarking is focused on two aspects:

- the **database**, by relying on pgbench
- the **storage**, by relying on fio

> **IMPORTANT**
>
> `pgbench` and `fio` must be run in a staging or pre-production environment. Do not use these plugins in a production environment, as it might have catastrophic consequences on your databases and the other workloads/applications that run in the same shared environment.

pgbench

The `kubectl` CNP 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 cnp pgbench \
  -n <namespace> <cluster-name> \
  --job-name <pgbench-job> \
  --db-name <db-name> \
  -- <pgbench options>
```

> **IMPORTANT**
>
> Please refer to the `pgbench` documentation 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 cnp 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 cnp 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 cnp 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 cnp 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 cnp pgbench \
  --db-name pgbench \
  --job-name pgbench-init \
  --node-selector workload=pgbench \
  cluster-example \
  -- --initialize --scale 1000
```

The job status can be fetched by running:

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

NAME       COMPLETIONS   DURATION   AGE
job-name   1/1           15s        41s
```

Once the job is completed the results can be gathered by executing:

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

fio

The kubectl CNP 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 cnp fio <fio-name>
```

Example with custom values:

```
kubectl cnp 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 cnp fio fio-job \
  -n fio  \
  --storageClass standard \
  --pvcSize 2Gi
```

The deployment status can be fetched by running:

```
kubectl get deployment/fio-job -n fio

NAME          READY   UP-TO-DATE   AVAILABLE   AGE
fio-job        1/1     1            1           14s
```

After running kubectl plugin command `fio`.

It will:

1. Create a PVC
2. Create a ConfigMap representing the configuration of a fio job
3. Create a fio deployment composed by a single Pod, which will run fio on the PVC, create graphs after completing the benchmark and start serving the generated files with a webserver. We use the `fio-tools` image for that.

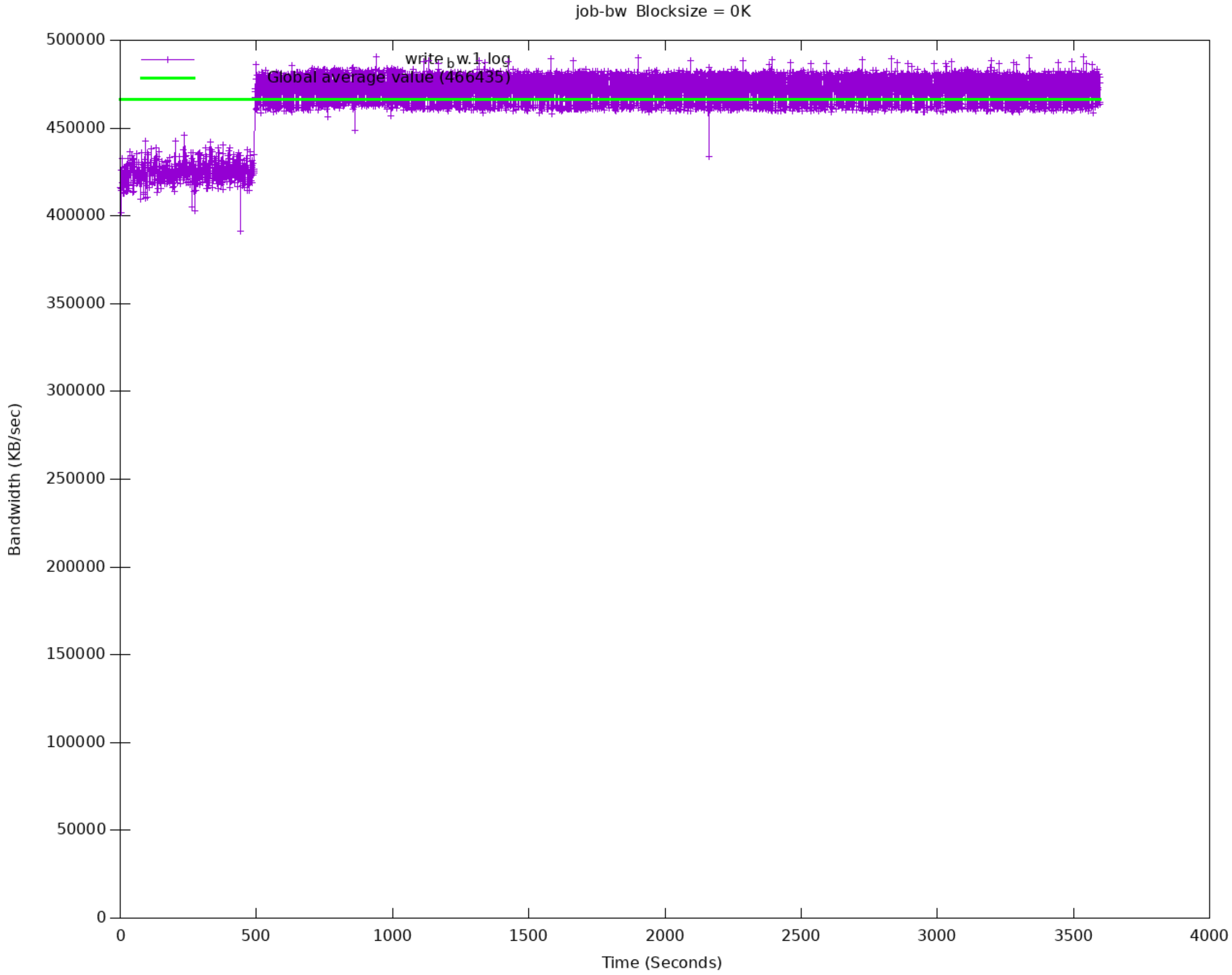
The Pod created by the deployment will be ready when it starts serving the results. You can forward the port of the pod created by the deployment

```
kubectl port-forward -n <namespace> deployment/<fio-name> 8000
```

and then use a browser and connect to `http://localhost:8000/` to get the data.

The default 8k block size has been chosen to emulate a PostgreSQL workload. Disks that cap the amount of available IOPS can show very different throughput values when changing this parameter.

Below is an example diagram of sequential writes on a local disk mounted on a dedicated Kubernetes node (1 hour benchmark):



After all testing is done, fio deployment and resources can be deleted by:

```
kubectl cnp 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.

47 Free evaluation

EDB Postgres for Kubernetes is available for a free evaluation.

Use your EDB account to evaluate Postgres for Kubernetes. If you don't have an account, register for one. Then follow the installation guide to install the operator, using the access token you obtained from your EDB account.

Evaluating using PostgreSQL

By default, EDB Postgres for Kubernetes installs the latest available version of Community Postgresql.

PostgreSQL container images are available at quay.io/enterprisedb/postgresql.

48 License and License keys

License keys are a legacy management mechanism for EDB Postgres for Kubernetes. You do not need a license key if you have installed using an EDB subscription token, and in this case, the licensing commands in this section can be ignored.

If you are not using an EDB subscription token and installing from public repositories, then you will need a license key. The only exception is when you run the operator with Community PostgreSQL: in this case, if the license key is unset, a cluster will be started with the default trial license - which automatically expires after 30 days. This is not the recommended way of trialing EDB Postgres for Kubernetes - see the installation guide for the recommended options.

The following documentation is only for users who have installed the operator using a license key.

Company level license keys

A license key allows you to create an unlimited number of PostgreSQL clusters in your installation.

The license key needs to be available in a `Secret` in the same namespace where the operator is deployed (`ConfigMap` is also available, but not recommended for a license key).

> **Operator configuration**
>
> For more information, refer to Operator configuration.

Once the company level license is installed, the validity of the license key can be checked inside the cluster status.

```
kubectl get cluster cluster_example -o
yaml
[...]
status:
  [...]
  licenseStatus:
    licenseExpiration: "2021-11-06T09:36:02Z"
    licenseStatus: Trial
    valid: true
    isImplicit: false
    isTrial: true
[...]
```

Kubernetes installations via YAML manifest

When the operator is installed in Kubernetes using the YAML manifest, it is deployed by default in the `postgresql-operator-system` namespace.

Given the namespace name, and the license key, you can create the config map with the following command:

```
kubectl create configmap -n [NAMESPACE_NAME_HERE] \
    postgresql-operator-controller-manager-config \
    --from-literal=EDB_LICENSE_KEY=[LICENSE_KEY_HERE]
```

Operator pods will need to be recreated to apply the new configuration. You can use the following command:

```
kubectl rollout restart deployment -n [NAMESPACE_NAME_HERE]
\
    postgresql-operator-controller-
manager
```

Cluster level license keys

Each `Cluster` resource has a `licenseKey` parameter in its definition. You can find the expiration date, as well as more information about the license, in the cluster status:

```
kubectl get cluster cluster_example -o
yaml
[...]
status:
  [...]
  licenseStatus:
    licenseExpiration: "2021-11-06T09:36:02Z"
    licenseStatus: Trial
    valid: true
    isImplicit: false
    isTrial: true
[...]
```

A cluster license key can be updated with a new one at any moment, to extend the expiration date or move the cluster to a production license.

License key secret at cluster level

Each `Cluster` resource can also have a `licenseKeySecret` parameter, which contains the name and key of a secret. That secret contains the license key provided by EDB.

This field will take precedence over `licenseKey` : it will be refreshed when you change the secret, in order to extend the expiration date, or switching from a trial license to a production license.

EDB Postgres for Kubernetes is distributed under the EDB Limited Usage License Agreement, available at enterprisedb.com/limited-use-license.

EDB Postgres for Kubernetes: Copyright (C) 2019-2022 EnterpriseDB Corporation.

What happens when a license expires

After the license expiration, the operator will cease any reconciliation attempt on the cluster, effectively stopping to manage its status. This also includes any self-healing and high availability capabilities, such as automated failover and switchovers.

The pods and the data will still be available.

49 Red Hat OpenShift

EDB Postgres for Kubernetes is certified to run on Red Hat OpenShift Container Platform (OCP) version 4.x and is available directly from the Red Hat Catalog.

The goal of this section is to help you decide the best installation method for EDB Postgres for Kubernetes based on your organizations' security and access control policies.

The first and critical step is to design the architecture of your PostgreSQL clusters in your OpenShift environment.

Once the architecture is clear, you can proceed with the installation. EDB Postgres for Kubernetes can be installed and managed via:

- OpenShift web console
- OpenShift command-line interface (CLI) called `oc` , for full control

EDB Postgres for Kubernetes supports all available install modes defined by OpenShift:

- cluster-wide, in all namespaces
- local, in a single namespace
- local, watching multiple namespaces (only available using `oc`)

> **Note**
>
> A project is a Kubernetes namespace with additional annotations, and is the central vehicle by which access to resources for regular users is managed.

In most cases, the default cluster-wide installation of EDB Postgres for Kubernetes is the recommended one, with either central management of PostgreSQL clusters or delegated management (limited to specific users/projects according to RBAC definitions - see "Important OpenShift concepts" and "Users and Permissions" below).

> **Important**
>
> Both the installation and upgrade processes require access to an OpenShift Container Platform cluster using an account with `cluster-admin` permissions. From "Default cluster roles", a `cluster-admin` is *"a super-user that can perform any action in any project. When bound to a user with a local binding, they have full control over quota and every action on every resource in the project"*.

Architecture

The same concepts that have been included in the generic Kubernetes/PostgreSQL architecture page apply for OpenShift as well.

Here as well, the critical factor is the number of availability zones or data centers for your OpenShift environment.

As outlined in the "Disaster Recovery Strategies for Applications Running on OpenShift" blog article written by Raffaele Spazzoli back in 2020 about stateful applications, in order to fully exploit EDB Postgres for Kubernetes, you need to plan, design and implement an OpenShift cluster spanning 3 or more availability zones. While this doesn't pose an issue in most of the public cloud provider deployments, it is definitely a challenge in on-premise scenarios.

If your OpenShift cluster has only **one availability zone**, the zone is your Single Point of Failure (SPoF) from a High Availability standpoint - provided that you have wisely adopted a share-nothing architecture, making sure that your PostgreSQL clusters have at least one standby (two if using synchronous replication), and that each PostgreSQL instance runs on a different Kubernetes worker node using different storage. Make sure that continuous backup data is stored additionally in a storage service outside the OpenShift cluster, allowing you to perform Disaster Recovery operations beyond your data center.

Most likely you will have another OpenShift cluster in another data center, either in the same metropolitan area or in another region, in an active/passive strategy. You can set up an independent "Replica cluster", with the understanding that this is primarily a Disaster Recovery solution - very effective but with some limitations that require manual intervention, as explained in the feature page. The same solution can be applied to additional OpenShift clusters, even in a cascading manner.

On the other hand, if your OpenShift cluster spans **multiple availability zones** in a region, you can fully leverage the capabilities of the operator for resilience and self-healing, and the region can become your SPoF, i.e. it would take a full region outage to bring down your cluster. Moreover, you can take advantage of multiple OpenShift clusters in different regions by setting up replica clusters, as previously mentioned.

Reserving Nodes for PostgreSQL Workloads

For optimal performance and resource allocation in your PostgreSQL database operations, it is highly recommended to isolate PostgreSQL workloads by dedicating specific worker nodes solely to `postgres` in production. This is particularly crucial whether you're operating in a single availability zone or a multi-availability zone environment.

A worker node in OpenShift that is dedicated to running PostgreSQL workloads is commonly referred to as a **Postgres node** or `postgres` node.

This dedicated approach ensures that your PostgreSQL workloads are not competing for resources with other applications, leading to enhanced stability and performance.

For further details, please refer to the "Reserving Nodes for PostgreSQL Workloads" section within the broader "Architecture" documentation. The primary difference when working in OpenShift involves how labels and taints are applied to the nodes, as described below.

To label a node as a `postgres` node, execute the following command:

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

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

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

By correctly labeling and tainting your nodes, you ensure that only PostgreSQL workloads are scheduled on these dedicated nodes via affinity and tolerations, reinforcing the stability and performance of your database environment.

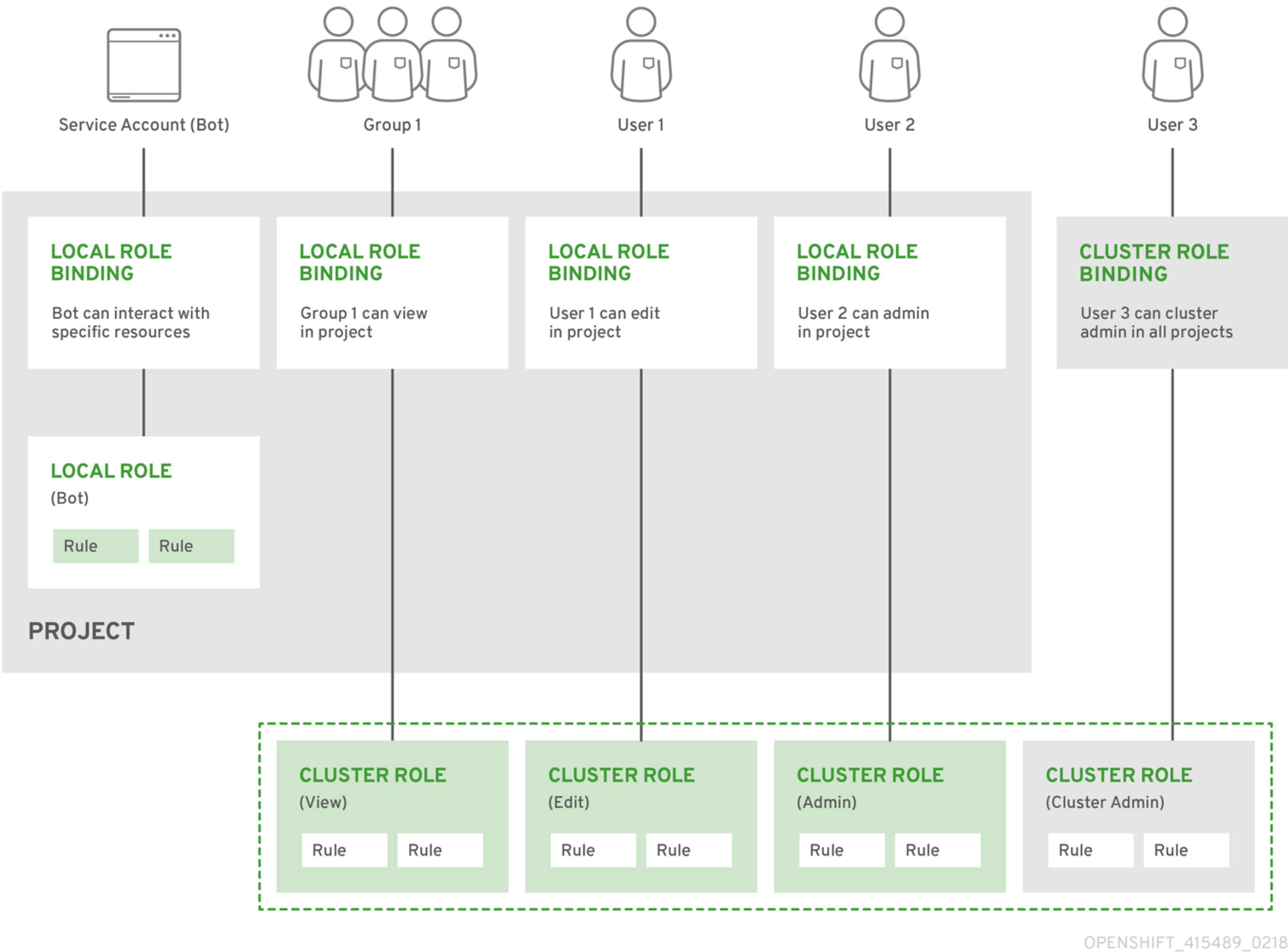
Important OpenShift concepts

To understand how the EDB Postgres for Kubernetes operator fits in an OpenShift environment, you must familiarize yourself with the following Kubernetes-related topics:

- Operators
- Authentication
- Authorization via Role-based Access Control (RBAC)
- Service Accounts and Users
- Rules, Roles and Bindings
- Cluster RBAC vs local RBAC through projects

This is especially true in case you are not comfortable with the elevated permissions required by the default cluster-wide installation of the operator.

We have also selected the diagram below from the OpenShift documentation, as it clearly illustrates the relationships between cluster roles, local roles, cluster role bindings, local role bindings, users, groups and service accounts.



The "Predefined RBAC objects" section below contains important information about how EDB Postgres for Kubernetes adheres to Kubernetes and OpenShift RBAC implementation, covering default installed cluster roles, roles, service accounts.

If you are familiar with the above concepts, you can proceed directly to the selected installation method. Otherwise, we recommend that you read the following resources taken from the OpenShift documentation and the Red Hat blog:

- "Operator Lifecycle Manager (OLM) concepts and resources"
- "Understanding authentication"
- "Role-based access control (RBAC)", covering rules, roles and bindings for authorization, as well as cluster RBAC vs local RBAC through projects
- "Default project service accounts and roles"
- "With Kubernetes Operators comes great responsibility" blog article

Cluster Service Version (CSV)

Technically, the operator is designed to run in OpenShift via the Operator Lifecycle Manager (OLM), according to the Cluster Service Version (CSV) defined by EDB.

The CSV is a YAML manifest that defines not only the user interfaces (available through the web dashboard), but also the RBAC rules required by the operator and the custom resources defined and owned by the operator (such as the `Cluster` one, for example). The CSV defines also the available `installModes` for the operator, namely: `AllNamespaces` (cluster-wide), `SingleNamespace` (single project), `MultiNamespace` (multi-project), and `OwnNamespace`.

> **There's more ...**
>
> You can find out more about CSVs and install modes by reading "Operator group membership" and "Defining cluster service versions (CSVs)" from the OpenShift documentation.

Limitations for multi-tenant management

Red Hat OpenShift Container Platform provides limited support for simultaneously installing different variations of an operator on a single cluster. Like any other operator, EDB Postgres for Kubernetes becomes an extension of the control plane. As the control plane is shared among all tenants (projects) of an OpenShift cluster, operators too become shared resources in a multi-tenant environment.

Operator Lifecycle Manager (OLM) can install operators multiple times in different namespaces, with one important limitation: they all need to share the same API version of the operator.

For more information, please refer to "Operator groups" in OpenShift documentation.

Channels

EDB Postgres for Kubernetes is distributed through the following OLM channels, each serving a distinct purpose:

- `candidate` : this channel provides early access to the next potential `fast` release. It includes the latest pre-release versions with new features and fixes, but is considered experimental and not supported. Use this channel only for testing and validation purposes—**not in production environments**. Versions in `candidate` may not appear in other channels if no further updates are recommended.

- `fast` : designed for users who want timely access to the latest stable features and patches. The head of the `fast` channel always points to the latest **patch release** of the latest **minor release** of EDB Postgres for Kubernetes.

- `stable` : similar to `fast` , but restricted to the latest **minor release** currently under EDB's Long Term Support (LTS) policy. Designed for users who require predictable updates and official support while benefiting from ongoing stability and maintenance.

- `stable-vX.Y` : tracks the latest patch release within a specific minor version (e.g., `stable-v1.26`). These channels are ideal for environments that require version pinning and predictable updates within a stable minor release.

The `fast` and `stable` channels may span multiple minor versions, whereas each `stable-vX.Y` channel is limited to patch updates within a specific minor release.

EDB Postgres for Kubernetes follow *trunk-based development* and *continuous delivery* principles. As a result, we generally recommend using the `fast` channel to stay current with the latest stable improvements and fixes.

Installation via web console

Ensuring access to EDB private registry

> **Important**
>
> You'll need access to the private EDB repository where both the operator and operand images are stored. Access requires a valid EDB subscription plan. Please refer to "Accessing EDB private image registries" for further details.

The OpenShift install will use pull secrets in order to access the operand and operator images, which are held in a private repository.

Once you have credentials to the private repository, you will need to create a pull secret in the `openshift-operators` namespace, named:

- `postgresql-operator-pull-secret` , for the EDB Postgres for Kubernetes operator images

You can create each secret via the `oc create` command, as follows:

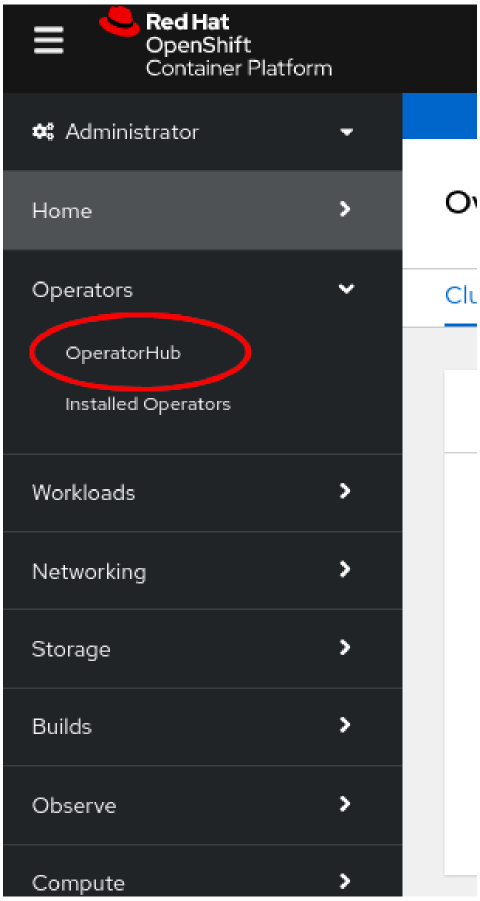

```
oc create secret docker-registry postgresql-operator-pull-secret
\
  -n openshift-operators --docker-server=docker.enterprisedb.com \
  --docker-username="@@REPOSITORY@@" \
  --docker-password="@@TOKEN@@"
```

where:

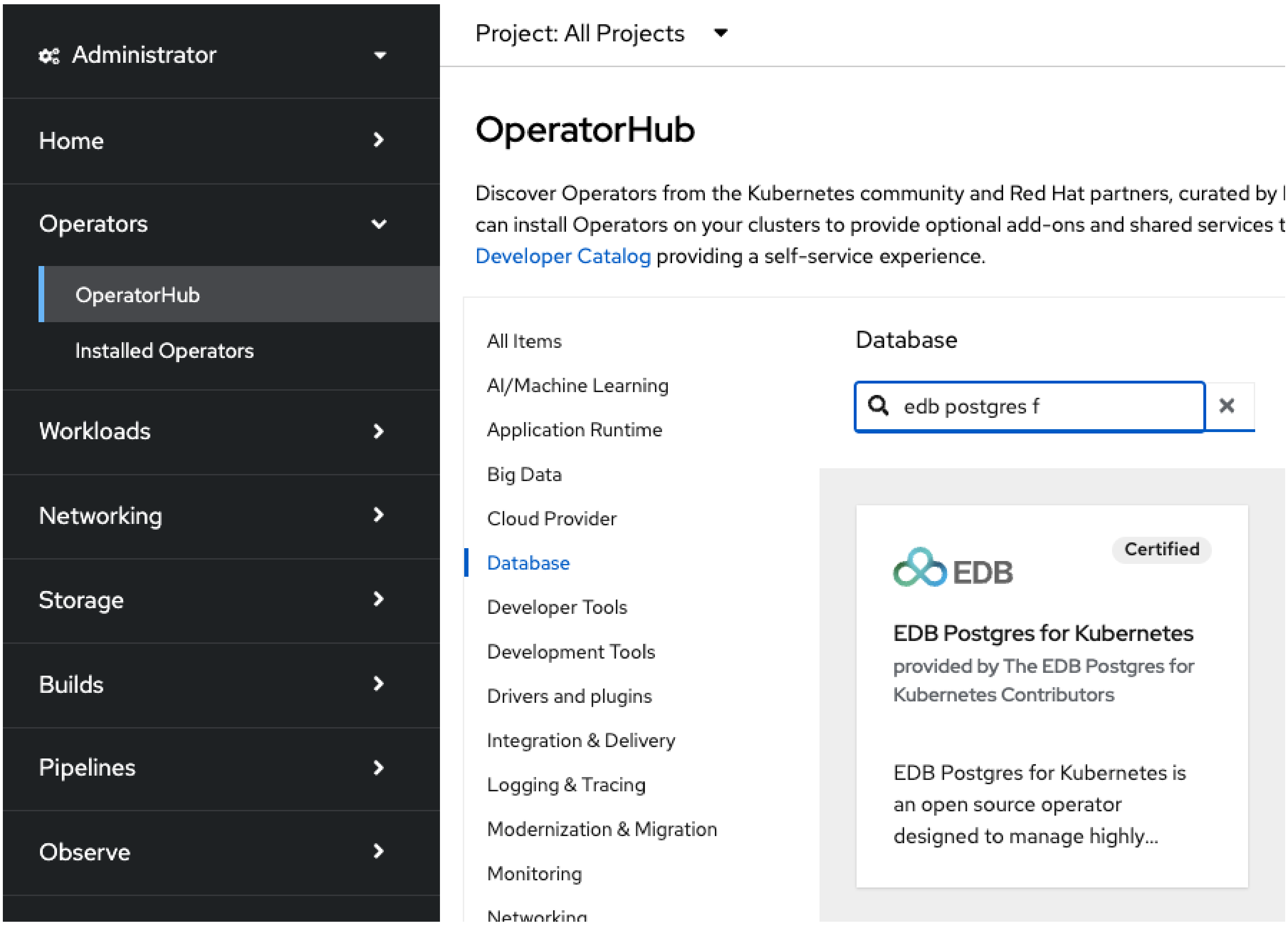
- `@@REPOSITORY@@` is the name of the repository, as explained in "Which repository to choose?"
- `@@TOKEN@@` is the repository token for your EDB account, as explained in "How to retrieve the token"

The EDB Postgres for Kubernetes operator can be found in the Red Hat OperatorHub directly from your OpenShift dashboard.

1. Navigate in the web console to the `Operators -> OperatorHub` page:



2. Scroll in the `Database` section or type a keyword into the `Filter by keyword` box (in this case, "PostgreSQL") to find the EDB Postgres for Kubernetes Operator, then select it:



3. Read the information about the Operator and select `Install`.

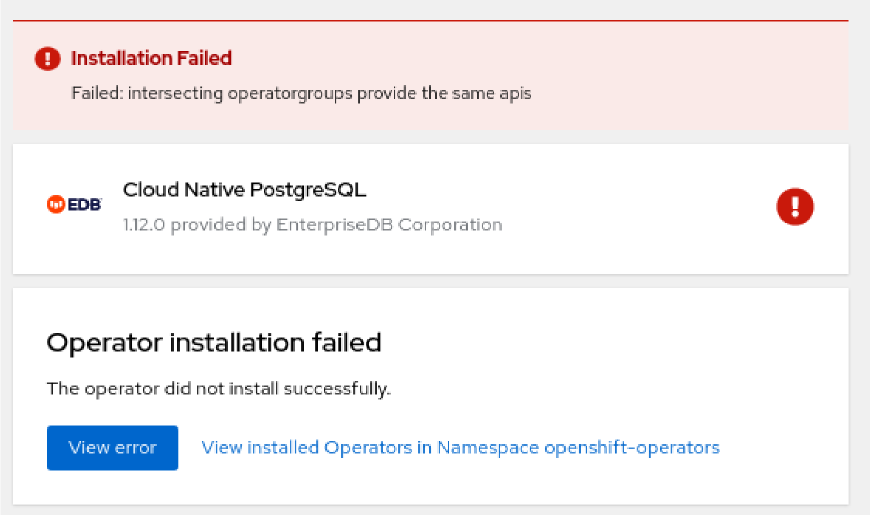
4. The following `Operator installation` page expects you to choose:

   - the installation mode: cluster-wide or single namespace installation
   - the update channel (see the "Channels" section for more information - if unsure, pick `fast`)
   - the approval strategy, following the availability on the market place of a new release of the operator, certified by Red Hat:
   - `Automatic`: OLM automatically upgrades the running operator with the new version
   - `Manual`: OpenShift waits for human intervention, by requiring an approval in the `Installed Operators` section

   > **Important**
   >
   > The process of the operator upgrade is described in the "Upgrades" section.

> **Important**
>
> It is possible to install the operator in a single project (technically speaking: `OwnNamespace` install mode) multiple times in the same cluster. There will be an operator installation in every namespace, with different upgrade policies as long as the API is the same (see "Limitations for multi-tenant management").

Choosing cluster-wide vs local installation of the operator is a critical turning point. Trying to install the operator globally with an existing local installation is blocked, by throwing the error below. If you want to proceed you need to remove every local installation of the operator first.

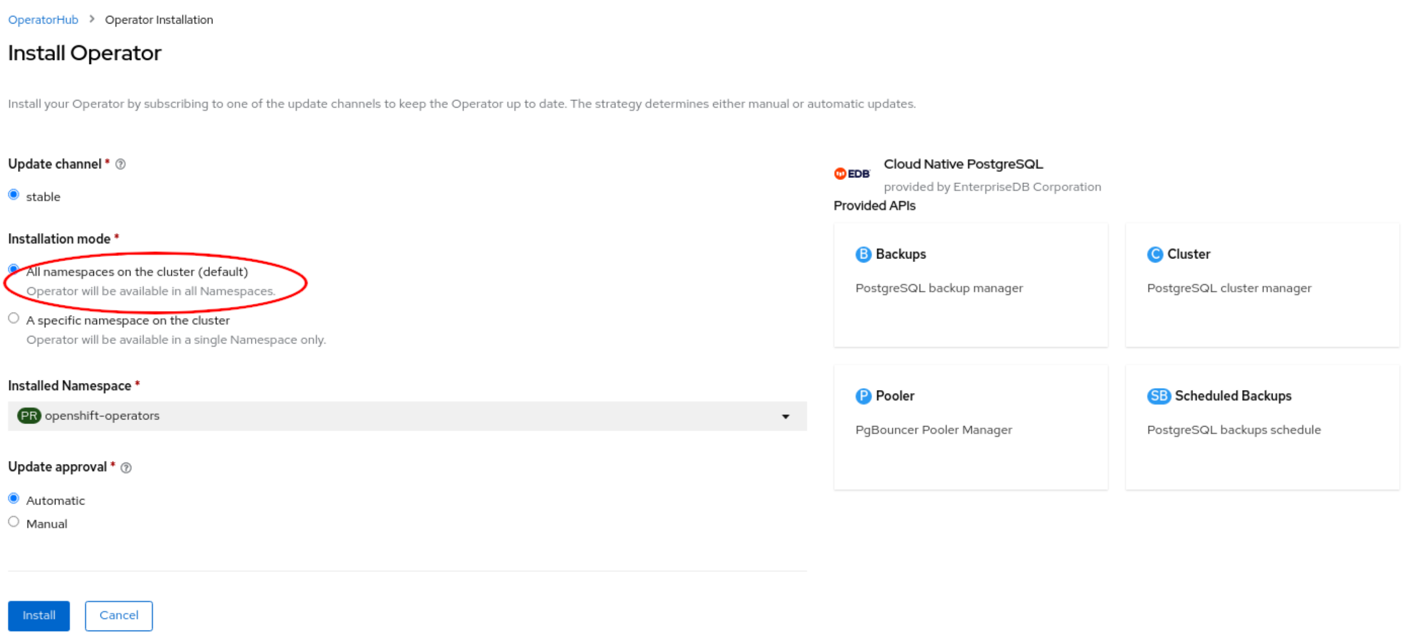


Cluster-wide installation

With cluster-wide installation, you are asking OpenShift to install the Operator in the default `openshift-operators` namespace and to make it available to all the projects in the cluster. This is the default and normally recommended approach to install EDB Postgres for Kubernetes.

> **Warning**
>
> This doesn't mean that every user in the OpenShift cluster can use the EDB Postgres for Kubernetes Operator, deploy a `Cluster` object or even see the `Cluster` objects that are running in their own namespaces. There are some special roles that users must have in the namespace in order to interact with EDB Postgres for Kubernetes' managed custom resources - primarily the `Cluster` one. Please refer to the "Users and Permissions" section below for details.

From the web console, select `All namespaces on the cluster (default)` as `Installation mode`:



As a result, the operator will be visible in every namespaces. Otherwise, as with any other OpenShift operator, check the logs in any pods in the `openshift-operators` project on the `Workloads → Pods` page that are reporting issues to troubleshoot further.

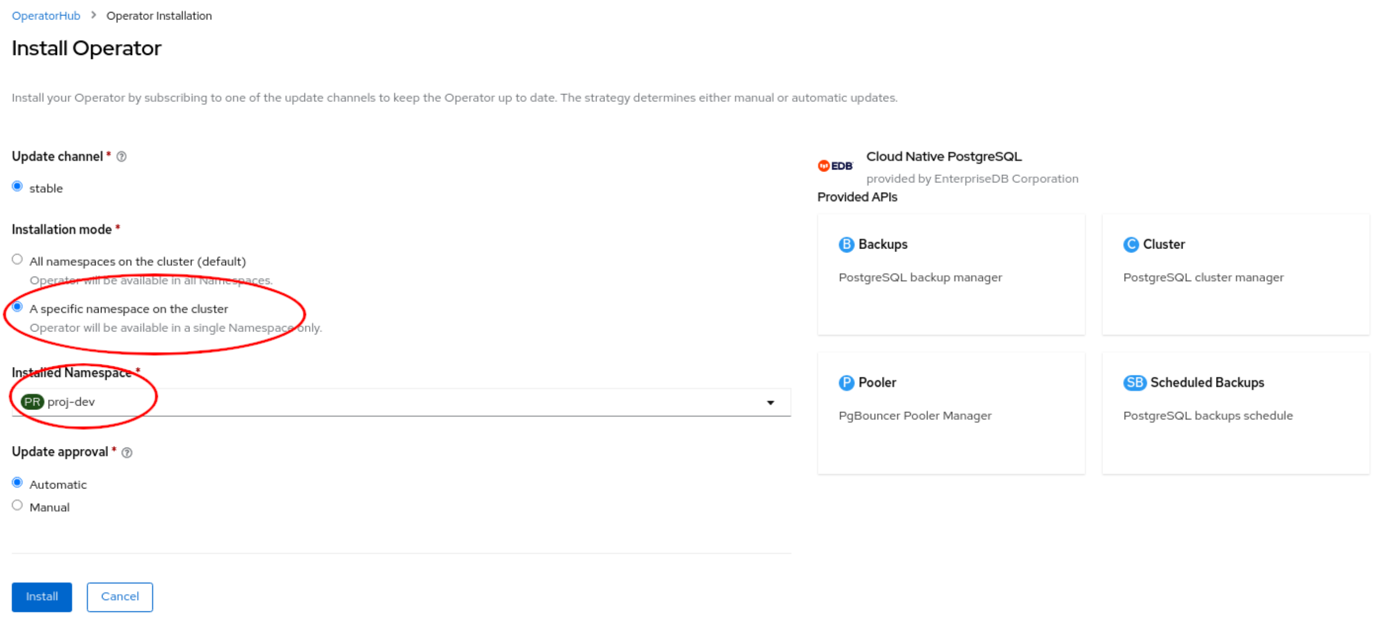
> **Beware**
>
> By choosing the cluster-wide installation you cannot easily move to a single project installation at a later time.

Single project installation

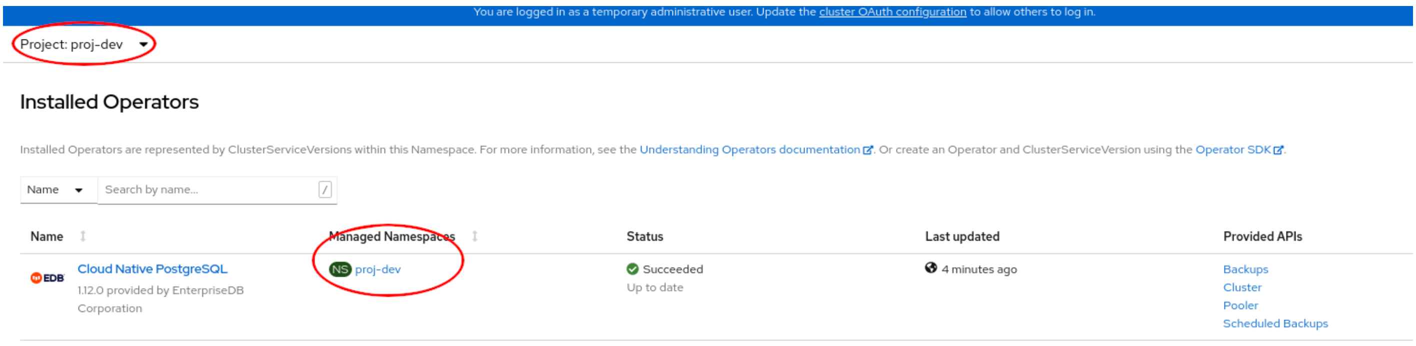
With single project installation, you are asking OpenShift to install the Operator in a given namespace, and to make it available to that project only.

> **Warning**
>
> This doesn't mean that every user in the namespace can use the EDB Postgres for Kubernetes Operator, deploy a `Cluster` object or even see the `Cluster` objects that are running in the namespace. Similarly to the cluster-wide installation mode, there are some special roles that users must have in the namespace in order to interact with EDB Postgres for Kubernetes' managed custom resources - primarily the `Cluster` one. Please refer to the "Users and Permissions" section below for details.

From the web console, select `A specific namespace on the cluster` as `Installation mode`, then pick the target namespace (in our example `proj-dev`):



As a result, the operator will be visible in the selected namespace only. You can verify this from the `Installed operators` page:



In case of a problem, from the `Workloads → Pods` page check the logs in any pods in the selected installation namespace that are reporting issues to troubleshoot further.

> **Beware**
>
> By choosing the single project installation you cannot easily move to a cluster-wide installation at a later time.

This installation process can be repeated in multiple namespaces in the same OpenShift cluster, enabling independent installations of the operator in different projects. In this case, make sure you read "Limitations for multi-tenant management".

Installation via the `oc` CLI

> **Important**
>
> Please refer to the "Installing the OpenShift CLI" section below for information on how to install the `oc` command-line interface.

Instead of using the OpenShift Container Platform web console, you can install the EDB Postgres for Kubernetes Operator from the OperatorHub and create a subscription using the `oc` command-line interface. Through the `oc` CLI you can install the operator in all namespaces, a single namespace or multiple namespaces.

> **Warning**
>
> Multiple namespace installation is currently supported by OpenShift. However, definition of multiple target namespaces for an operator may be removed in future versions of OpenShift.

This section primarily covers the installation of the operator in multiple projects with a simple example, by creating an `OperatorGroup` and a `Subscription` objects.

> **Info**
>
> In our example, we will install the operator in the `my-operators` namespace and make it only available in the `web-staging`, `web-prod`, `bi-staging`, and `bi-prod` namespaces. Feel free to change the names of the projects as you like or add/remove some namespaces.

1. Check that the `cloud-native-postgresql` operator is available from the OperatorHub:

```
oc get packagemanifests -n openshift-marketplace cloud-native-postgresql
```

2. Inspect the operator to verify the installation modes (`MultiNamespace` in particular) and the available channels:

```
oc describe packagemanifests -n openshift-marketplace cloud-native-postgresql
```

3. Create an `OperatorGroup` object in the `my-operators` namespace so that it targets the `web-staging`, `web-prod`, `bi-staging`, and `bi-prod` namespaces:

```
apiVersion: operators.coreos.com/v1
kind: OperatorGroup
metadata:
  name: cloud-native-postgresql
  namespace: my-operators
spec:
  targetNamespaces:
  - web-staging
  - web-prod
  - bi-staging
  - bi-prod
```

!!! Important Alternatively, you can list namespaces using a label selector, as explained in "Target namespace selection".

4. Create a `Subscription` object in the `my-operators` namespace to subscribe to the `fast` channel of the `cloud-native-postgresql` operator that is available in the `certified-operators` source of the `openshift-marketplace` (as previously located in steps 1 and 2):

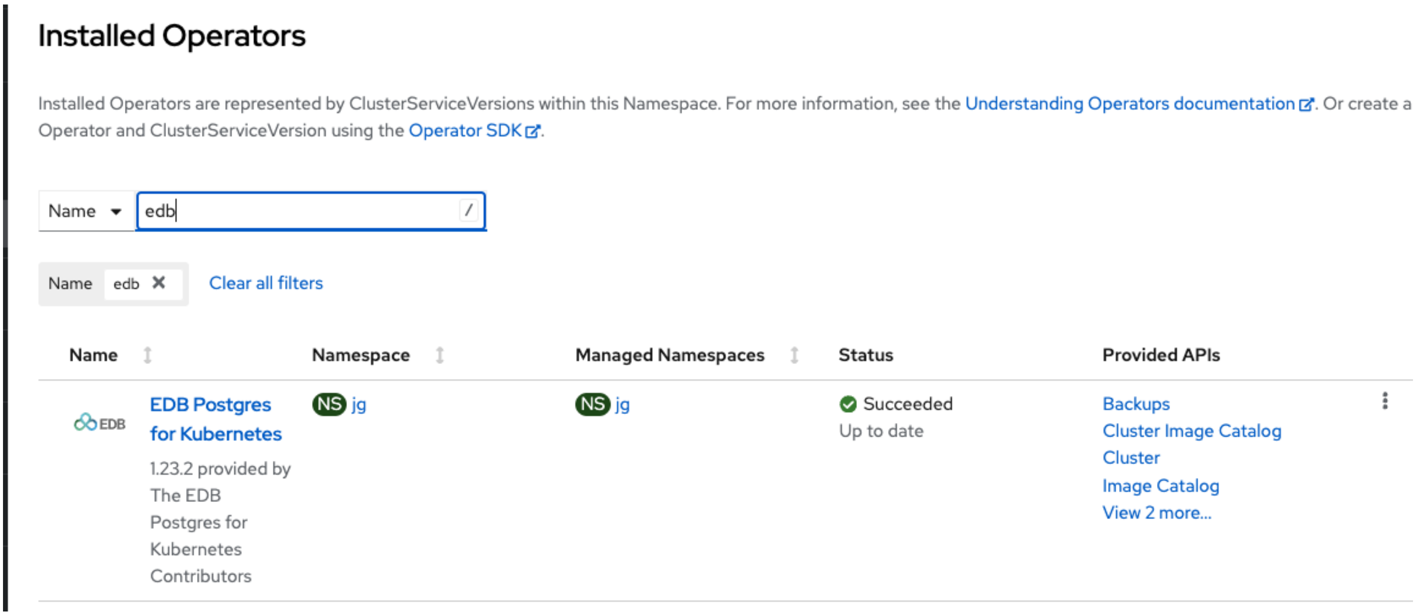
```
apiVersion: operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: cloud-native-postgresql
  namespace: my-operators
spec:
  channel: fast
  name: cloud-native-postgresql
  source: certified-operators
  sourceNamespace: openshift-marketplace
```

5. Use `oc apply -f` with the above YAML file definitions for the `OperatorGroup` and `Subscription` objects.

The method described in this section can be very powerful in conjunction with proper `RoleBinding` objects, as it enables mapping EDB Postgres for Kubernetes' predefined `ClusterRole` s to specific users in selected namespaces.

> **Info**
>
> The above instructions can also be used for single project binding. The only difference is the number of specified target namespaces (one) and, possibly, the namespace of the operator group (ideally, the same as the target namespace).

The result of the above operation can also be verified from the webconsole, as shown in the image below.



Cluster-wide installation with `oc`

If you prefer, you can also use `oc` to install the operator globally, by taking advantage of the default `OperatorGroup` called `global-operators` in the `openshift-operators` namespace, and create a new `Subscription` object for the `cloud-native-postgresql` operator in the same namespace:

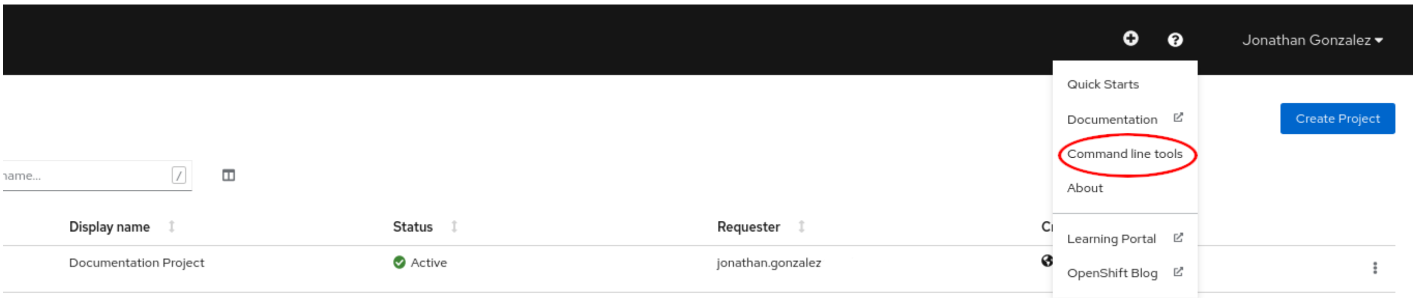

```yaml
apiVersion:
operators.coreos.com/v1alpha1
kind: Subscription
metadata:
  name: cloud-native-
postgresql
  namespace: openshift-operators
spec:
  channel: fast
  name: cloud-native-
postgresql
  source: certified-operators
  sourceNamespace: openshift-marketplace
```

Once you run `oc apply -f` with the above YAML file, the operator will be available in all namespaces.

Installing the OpenShift CLI (`oc`)

The `oc` command represents the OpenShift command-line interface (CLI). It is highly recommended to install it on your system. Below you find a basic set of instructions to install `oc` from your OpenShift dashboard.

First, select the question mark at the top right corner of the dashboard:



Then follow the instructions you are given, by downloading the binary that suits your needs in terms of operating system and architecture:

Command Line Tools

Copy Login Command

oc - OpenShift Command Line Interface (CLI)

With the OpenShift command line interface, you can create applications and manage OpenShift projects from a terminal.

The oc binary offers the same capabilities as the kubectl binary, but it is further extended to natively support OpenShift Container Platform features.

- Download oc for Linux for x86_64
- Download oc for Mac for x86_64
- Download oc for Windows for x86_64
- Download oc for Linux for ARM 64
- Download oc for Linux for IBM Power, little endian
- Download oc for Linux for IBM Z
- LICENSE

helm - Helm 3 CLI

Helm 3 is a package manager for Kubernetes applications which enables defining, installing, and upgrading applications packaged as Helm Charts.

Download Helm

odo - Developer-focused CLI for OpenShift

odo is a fast, iterative, and straightforward CLI tool for developers who write, build, and deploy applications on OpenShift.

odo abstracts away complex Kubernetes and OpenShift concepts, thus allowing developers to focus on what is most important to them: code.

Download odo

> OpenShift CLI
>
> For more detailed and updated information, please refer to the official OpenShift CLI documentation directly maintained by Red Hat.

Predefined RBAC objects

EDB Postgres for Kubernetes comes with a predefined set of resources that play an important role when it comes to RBAC policy configuration.

Custom Resource Definitions (CRD)

The EDB Postgres for Kubernetes operator owns the following custom resource definitions (CRD):

- `Backup`
- `Cluster`
- `Pooler`
- `ScheduledBackup`
- `ImageCatalog`
- `ClusterImageCatalog`

You can verify this by running:

```
oc get customresourcedefinitions.apiextensions.k8s.io | grep
postgresql
```

which returns something similar to:

```
backups.postgresql.k8s.enterprisedb.io                          20YY-MM-DDTHH:MM:SSZ
clusterimagecatalogs.postgresql.k8s.enterprisedb.io             20YY-MM-DDTHH:MM:SSZ
clusters.postgresql.k8s.enterprisedb.io                         20YY-MM-DDTHH:MM:SSZ
imagecatalogs.postgresql.k8s.enterprisedb.io                    20YY-MM-DDTHH:MM:SSZ
poolers.postgresql.k8s.enterprisedb.io                          20YY-MM-DDTHH:MM:SSZ
scheduledbackups.postgresql.k8s.enterprisedb.io                 20YY-MM-DDTHH:MM:SSZ
```

Service accounts

The namespace where the operator has been installed (by default `openshift-operators`) contains the following predefined service accounts: `builder` , `default` , `deployer` , and most importantly `postgresql-operator-manager` (managed by the CSV).

> **Important**
>
> Service accounts in Kubernetes are namespaced resources. Unless explicitly authorized, a service account cannot be accessed outside the defined namespace.

You can verify this by running:

```
oc get serviceaccounts -n openshift-
operators
```

which returns something similar to:

```
NAME                          SECRETS   AGE
builder                       2         ...
default                       2         ...
deployer                      2         ...
postgresql-operator-manager   2         ...
```

The `default` service account is automatically created by Kubernetes and present in every namespace. The `builder` and `deployer` service accounts are automatically created by OpenShift (see "Default project service accounts and roles").

The `postgresql-operator-manager` service account is the one used by the Cloud Native PostgreSQL operator to work as part of the Kubernetes/OpenShift control plane in managing PostgreSQL clusters.

> **Important**
>
> Do not delete the `postgresql-operator-manager` ServiceAccount as it can stop the operator from working.

Cluster roles

The Operator Lifecycle Manager (OLM) automatically creates a set of cluster role objects to facilitate role binding definitions and granular implementation of RBAC policies. Some cluster roles have rules that apply to Custom Resource Definitions that are part of EDB Postgres for Kubernetes, while others that are part of the broader Kubernetes/OpenShift realm.

Cluster roles on EDB Postgres for Kubernetes CRDs

For every CRD owned by EDB Postgres for Kubernetes' CSV, OLM deploys some predefined cluster roles that can be used by customer facing users and service accounts. In particular:

- a role for the full administration of the resource (`admin` suffix)
- a role to edit the resource (`edit` suffix)
- a role to view the resource (`view` suffix)
- a role to view the actual CRD (`crdview` suffix)

> **Important**
>
> Cluster roles per se are no security threat. They are the recommended way in OpenShift to define templates for roles to be later "bound" to actual users in a specific project or globally. Indeed, cluster roles can be used in conjunction with `ClusterRoleBinding` objects for global permissions or with `RoleBinding` objects for local permissions. This makes it possible to reuse cluster roles across multiple projects while enabling customization within individual projects through local roles.

You can verify the list of predefined cluster roles by running:

```
oc get clusterroles | grep
postgresql
```

which returns something similar to:

```
backups.postgresql.k8s.enterprisedb.io-v1-admin                 YYYY-MM-DDTHH:MM:SSZ
backups.postgresql.k8s.enterprisedb.io-v1-crdview               YYYY-MM-DDTHH:MM:SSZ
backups.postgresql.k8s.enterprisedb.io-v1-edit                  YYYY-MM-DDTHH:MM:SSZ
backups.postgresql.k8s.enterprisedb.io-v1-view                  YYYY-MM-DDTHH:MM:SSZ
cloud-native-postgresql.VERSION-HASH                            YYYY-MM-DDTHH:MM:SSZ
clusterimagecatalogs.postgresql.k8s.enterprisedb.io-v1-admin    YYYY-MM-DDTHH:MM:SSZ
clusterimagecatalogs.postgresql.k8s.enterprisedb.io-v1-crdview  YYYY-MM-DDTHH:MM:SSZ
clusterimagecatalogs.postgresql.k8s.enterprisedb.io-v1-edit     YYYY-MM-DDTHH:MM:SSZ
clusterimagecatalogs.postgresql.k8s.enterprisedb.io-v1-view     YYYY-MM-DDTHH:MM:SSZ
clusters.postgresql.k8s.enterprisedb.io-v1-admin                YYYY-MM-DDTHH:MM:SSZ
clusters.postgresql.k8s.enterprisedb.io-v1-crdview              YYYY-MM-DDTHH:MM:SSZ
clusters.postgresql.k8s.enterprisedb.io-v1-edit                 YYYY-MM-DDTHH:MM:SSZ
clusters.postgresql.k8s.enterprisedb.io-v1-view                 YYYY-MM-DDTHH:MM:SSZ
imagecatalogs.postgresql.k8s.enterprisedb.io-v1-admin           YYYY-MM-DDTHH:MM:SSZ
imagecatalogs.postgresql.k8s.enterprisedb.io-v1-crdview         YYYY-MM-DDTHH:MM:SSZ
imagecatalogs.postgresql.k8s.enterprisedb.io-v1-edit            YYYY-MM-DDTHH:MM:SSZ
imagecatalogs.postgresql.k8s.enterprisedb.io-v1-view            YYYY-MM-DDTHH:MM:SSZ
poolers.postgresql.k8s.enterprisedb.io-v1-admin                 YYYY-MM-DDTHH:MM:SSZ
poolers.postgresql.k8s.enterprisedb.io-v1-crdview               YYYY-MM-DDTHH:MM:SSZ
poolers.postgresql.k8s.enterprisedb.io-v1-edit                  YYYY-MM-DDTHH:MM:SSZ
poolers.postgresql.k8s.enterprisedb.io-v1-view                  YYYY-MM-DDTHH:MM:SSZ
scheduledbackups.postgresql.k8s.enterprisedb.io-v1-admin        YYYY-MM-DDTHH:MM:SSZ
scheduledbackups.postgresql.k8s.enterprisedb.io-v1-crdview      YYYY-MM-DDTHH:MM:SSZ
scheduledbackups.postgresql.k8s.enterprisedb.io-v1-edit         YYYY-MM-DDTHH:MM:SSZ
scheduledbackups.postgresql.k8s.enterprisedb.io-v1-view         YYYY-MM-DDTHH:MM:SSZ
```

You can inspect an actual role as any other Kubernetes resource with the `get` command. For example:

```
oc get -o yaml clusterrole clusters.postgresql.k8s.enterprisedb.io-v1-
admin
```

By looking at the relevant skimmed output below, you can notice that the `clusters.postgresql.k8s.enterprisedb.io-v1-admin` cluster role enables everything on the `cluster` resource defined by the `postgresql.k8s.enterprisedb.io` API group:


```
apiVersion: rbac.authorization.k8s.io/v1
kind:
ClusterRole
metadata:
  name: clusters.postgresql.k8s.enterprisedb.io-v1-admin
rules:
- apiGroups:
  - postgresql.k8s.enterprisedb.io
  resources:
  -
clusters
  verbs:
  - '*'
```

> **There's more ...**
>
> If you are interested in the actual implementation of RBAC by an OperatorGroup, please refer to the"OperatorGroup: RBAC" section from the Operator Lifecycle Manager documentation.

Cluster roles on Kubernetes CRDs

When installing a `Subscription` object in a given namespace (e.g. `openshift-operators` for cluster-wide installation of the operator), OLM also creates a cluster role that is used to grant permissions to the `postgresql-operator-manager` service account that the operator uses. The name of this cluster role varies, as it depends on the installed version of the operator and the time of installation.

You can retrieve it by running the following command:

```
oc get clusterrole --
selector=olm.owner.kind=ClusterServiceVersion
```

You can then use the name returned by the above query (which should have the form of `cloud-native-postgresql.VERSION-HASH`) to look at the rules, resources and verbs via the `describe` command:

```
oc describe clusterrole cloud-native-postgresql.VERSION-
HASH
```

```
Name:         cloud-native-postgresql.VERSION.HASH
Labels:       olm.owner=cloud-native-postgresql.VERSION
              olm.owner.kind=ClusterServiceVersion
              olm.owner.namespace=openshift-operators
              operators.coreos.com/cloud-native-postgresql.openshift-operators=
Annotations:  <none>
PolicyRule:
  Resources                                                    Non-Resource URLs  Resource Names  Verbs
  ---------                                                    -----------------  --------------  -----
  configmaps                                                   []                 []              [create
delete get list patch update watch]
  secrets                                                      []                 []              [create
delete get list patch update watch]
  services                                                     []                 []              [create
delete get list patch update watch]
  deployments.apps                                             []                 []              [create
delete get list patch update watch]
  poddisruptionbudgets.policy                                  []                 []              [create
delete get list patch update watch]
  backups.postgresql.k8s.enterprisedb.io                       []                 []              [create
delete get list patch update watch]
  clusters.postgresql.k8s.enterprisedb.io                      []                 []              [create
```



```
delete get list patch update watch]
  poolers.postgresql.k8s.enterprisedb.io                          []              []              [create
delete get list patch update watch]
  scheduledbackups.postgresql.k8s.enterprisedb.io                 []              []              [create
delete get list patch update watch]
  persistentvolumeclaims                                          []              []              [create
delete get list patch watch]
  pods/exec                                                       []              []              [create
delete get list patch watch]
  pods                                                            []              []              [create
delete get list patch watch]
  jobs.batch                                                      []              []              [create
delete get list patch watch]
  podmonitors.monitoring.coreos.com                               []              []              [create
delete get list patch watch]
  serviceaccounts                                                 []              []              [create ge
list patch update watch]
  rolebindings.rbac.authorization.k8s.io                          []              []              [create ge
list patch update watch]
  roles.rbac.authorization.k8s.io                                 []              []              [create ge
list patch update watch]
  leases.coordination.k8s.io                                      []              []              [create ge
update]
  events                                                          []              []              [create
patch]
  mutatingwebhookconfigurations.admissionregistration.k8s.io      []              []              [get list
update]
  validatingwebhookconfigurations.admissionregistration.k8s.io    []              []              [get list
update]
  customresourcedefinitions.apiextensions.k8s.io                  []              []              [get list
update]
  namespaces                                                      []              []              [get list
watch]
  nodes                                                           []              []              [get list
watch]
  clusters.postgresql.k8s.enterprisedb.io/status                  []              []              [get patch
update watch]
  poolers.postgresql.k8s.enterprisedb.io/status                   []              []              [get patch
update watch]
  configmaps/status                                               []              []              [get patch
update]
  secrets/status                                                  []              []              [get patch
update]
  backups.postgresql.k8s.enterprisedb.io/status                   []              []              [get patch
update]
  scheduledbackups.postgresql.k8s.enterprisedb.io/status          []              []              [get patch
update]
  pods/status                                                     []              []              [get]
  clusters.postgresql.k8s.enterprisedb.io/finalizers              []              []              [update]
  poolers.postgresql.k8s.enterprisedb.io/finalizers               []              []              [update]
```

> **Important**
>
> The above permissions are exclusively reserved for the operator's service account to interact with the Kubernetes API server. They are not directly accessible by the users of the operator that interact only with `Cluster`, `Pooler`, `Backup`, and `ScheduledBackup` resources (see "Cluster roles on EDB Postgres for Kubernetes CRDs").

The operator automates in a declarative way a lot of operations related to PostgreSQL management that otherwise would require manual and imperative interventions. Such operations also include security related matters at RBAC (e.g. service accounts), pod (e.g. security context constraints) and Postgres levels (e.g. TLS certificates).

For more information about the reasons why the operator needs these elevated permissions, please refer to the"Security / Cluster / RBAC" section.

Users and Permissions

A very common way to use the EDB Postgres for Kubernetes operator is to rely on the `cluster-admin` role and manage resources centrally.

Alternatively, you can use the RBAC framework made available by Kubernetes/OpenShift, as with any other operator or resources.

For example, you might be interested in binding the `clusters.postgresql.k8s.enterprisedb.io-v1-admin` cluster role to specific groups or users in a specific namespace, as any other cloud native application. The following example binds that cluster role to a specific user in the `web-prod` project:

```
kind:
RoleBinding
apiVersion: rbac.authorization.k8s.io/v1
metadata:
  name: web-prod-admin
  namespace: web-prod
subjects:
  - kind: User
    apiGroup: rbac.authorization.k8s.io
    name: mario@cioni.org
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind:
ClusterRole
  name: clusters.postgresql.k8s.enterprisedb.io-v1-admin
```

The same process can be repeated with any other predefined `ClusterRole`.

If, on the other hand, you prefer not to use cluster roles, you can create specific namespaced roles like in this example:

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: web-prod-view
  namespace: web-prod
rules:
- apiGroups:
  - postgresql.k8s.enterprisedb.io
  resources:
  -
clusters
  verbs:
  -
get
  - list
  - watch
```

Then, assign this role to a given user:

```
apiVersion: rbac.authorization.k8s.io/v1
kind:
RoleBinding
metadata:
  name: web-prod-view
  namespace: web-prod
roleRef:
  apiGroup: rbac.authorization.k8s.io
  kind: Role
  name: web-prod-view
subjects:
- apiGroup: rbac.authorization.k8s.io
  kind: User
  name: web-prod-developer1
```

This final example creates a role with administration permissions (`verbs` is equal to `*`) to all the resources managed by the operator in that namespace (`web-prod`):

```
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: web-prod-admin
  namespace: web-prod
rules:
- apiGroups:
  - postgresql.k8s.enterprisedb.io
  resources:
  -
clusters
  - backups
  -
scheduledbackups
  - poolers
  verbs:
  - '*'
```

Pod Security Standards

EDB Postgres for Kubernetes on OpenShift works with the `restricted-v2` SCC (`SecurityContextConstraints`).

Since the operator has been developed with a security focus from the beginning, in addition to always adhering to the Red Hat Certification process, EDB Postgres for Kubernetes works under the new SCCs introduced in OpenShift 4.11.

By default, EDB Postgres for Kubernetes will drop all capabilities. This ensures that during its lifecycle the operator will never make use of any unsafe capabilities.

On OpenShift we inherit the `SecurityContext.SeccompProfile` for each Pod from the OpenShift deployment, which in turn is set by the Pod Security Admission Controller.

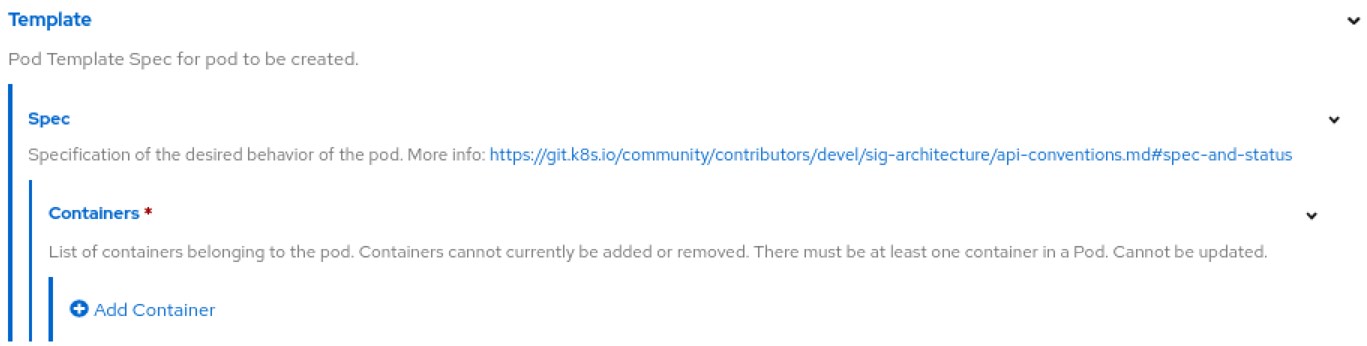
> **Note**
>
> Even if `nonroot-v2` and `hostnetwork-v2` are qualified as less restricted SCCs, we don't run tests on them, and therefore we cannot guarantee that these SCCs will work. That being said, `nonroot-v2` and `hostnetwork-v2` are a subset of rules in `restricted-v2` so there is no reason to believe that they would not work.

Customization of the Pooler image

By default, the `Pooler` resource creates pods having the `pgbouncer` container that runs with the `quay.io/enterprisedb/pgbouncer` image.

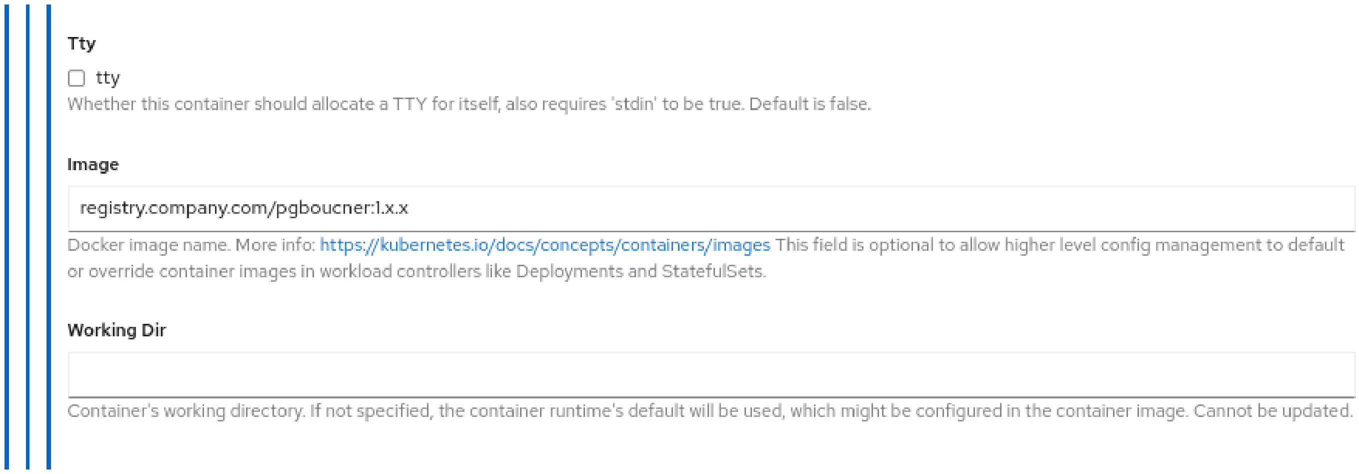
> **There's more**
>
> For more details about pod customization for the pooler, please refer to the"Pod templates" section in the connection pooling documentation.

You can change the image name from the advanced interface, specifically by opening the *"Template"* section, then selecting *"Add container"* under *"Spec > Containers"*:



Then:

- set `pgbouncer` as the name of the container (required field in the pod template)
- set the *"Image"* field as desired (see the image below)



OADP for Velero

The EDB Postgres for Kubernetes operator recommends the use of the Openshift API for Data Protection operator for managing Velero in OpenShift environments. Specific details about how EDB Postgres for Kubernetes integrates with Velero can be found in the Velero section of the Addons documentation. The OADP operator is a community operator that is not directly supported by EDB. The OADP operator is not required to use Velero with EDB Postgres but is a convenient way to install Velero on OpenShift.

Monitoring and metrics

OpenShift includes a Prometheus installation out of the box that can be leveraged for user-defined projects, including EDB Postgres for Kubernetes.

Grafana integration is out of scope for this guide, as Grafana is no longer included with OpenShift.

In this section, we show you how to get started with basic observability, leveraging the default OpenShift installation.

Please refer to the OpenShift monitoring stack overview for further background.

Depending on your OpenShift configuration, you may need to do a bit of setup before you can monitor your EDB Postgres for Kubernetes clusters.

You will need to have your OpenShift configured to enable monitoring for user-defined projects.

You should check, perhaps with your OpenShift administrator, if your installation has the `cluster-monitoring-config` configMap, and if so, if user workload monitoring is enabled.

You can check for the presence of this `configMap` (note that it should be in the `openshift-monitoring` namespace):

```
oc -n openshift-monitoring get configmap cluster-monitoring-
config
```

To enable user workload monitoring, you might want to `oc apply` or `oc edit` the configmap to look something like this:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: cluster-monitoring-
config
  namespace: openshift-monitoring
data:
  config.yaml:
|
    enableUserWorkload:
true
```

After `enableUserWorkload` is set, several monitoring components will be created in the `openshift-user-workload-monitoring` namespace.

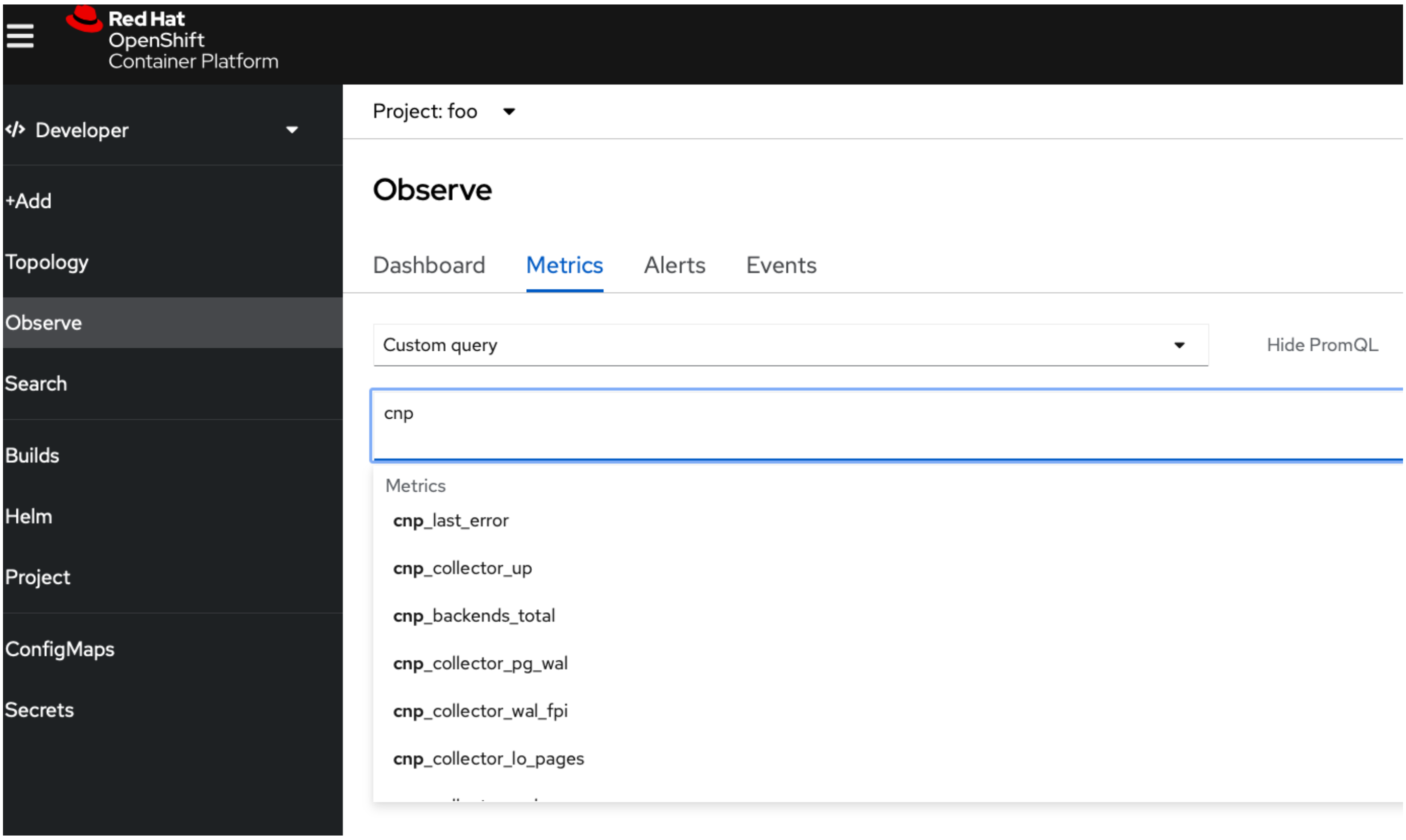
```
$ oc -n openshift-user-workload-monitoring get
pod
NAME                                  READY   STATUS    RESTARTS
AGE
prometheus-operator-58768d7cc-28xb5   2/2     Running   0
5h10m
prometheus-user-workload-0            6/6     Running   0
5h10m
prometheus-user-workload-1            6/6     Running   0
5h10m
thanos-ruler-user-workload-0          3/3     Running   0
5h10m
thanos-ruler-user-workload-1          3/3     Running   0
5h10m
```

You should now be able to see metrics from any cluster enabling them.

For example, we can create the following cluster with monitoring on the `foo` namespace:

```
kubectl apply -n foo -f -
<<EOF
---
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-with-
metrics
spec:
  instances:
3


storage:
    size:
1Gi


monitoring:
    enablePodMonitor: true
EOF
```

You should now be able to query for the default metrics that will be installed with this example. In the `Observe` section of OpenShift (in Developer perspective), you should see a `Metrics` submenu where you can write PromQL queries. Auto-complete is enabled, so you can peek the `cnp_` prefix:



It is easy to define Alerts based on the default metrics as `PrometheusRules`. You can find some examples of rules in the prometheusrule.yaml file, which you can download.

Before applying the rules, again, some OpenShift setup may be necessary.

The `monitoring-rules-edit` or at least `monitoring-rules-view` roles should be assigned for the user wishing to apply and monitor the rules.

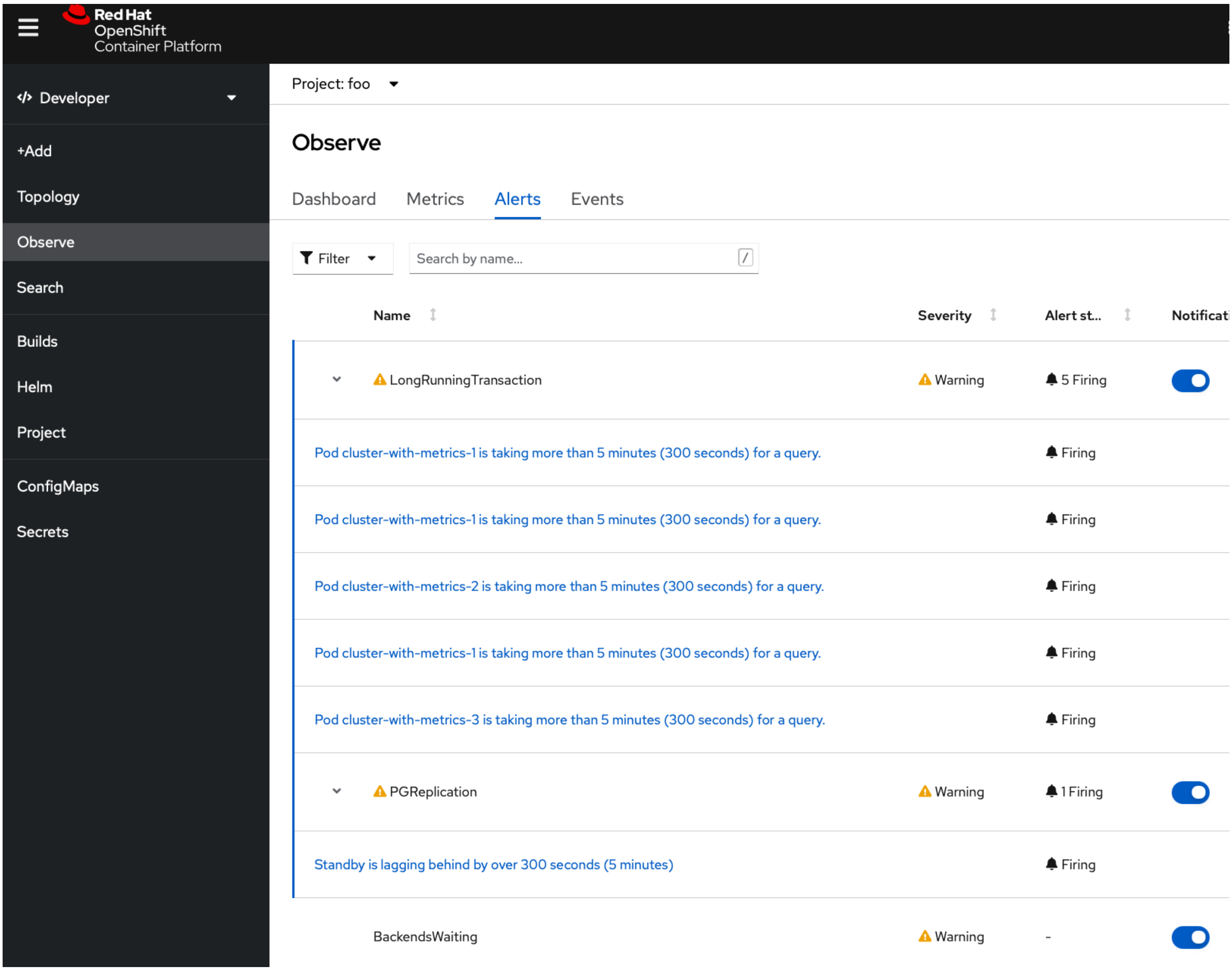
This involves creating a RoleBinding with that permission, for a namespace. Again, refer to the relevant OpenShift document page for further detail. Specifically, the *Granting user permissions by using the web console* section should be of interest.

Note that the RoleBinding for `monitoring-rules-edit` applies to a namespace, so make sure you get the right one/ones.

Suppose that the `cnp-prometheusrule.yaml` file that you have previously downloaded now contains your alerts. You can install those rules as follows:

```
oc apply -n foo -f cnp-
prometheusrule.yaml
```

Now you should be able to see Alerts, if there are any. Note that initially there might be no alerts.



Alert routing and notifications are beyond the scope of this guide.

50 Transparent Data Encryption (TDE)

> **Important**
>
> TDE is available *only* for operands that support it: EPAS and PG Extended, versions 15 and newer.

Transparent Data Encryption, or TDE, is a technology used by several database vendors to **encrypt data at rest**, i.e. database files on disk. TDE does not however encrypt data in use.

TDE is included in EDB Postgres Advanced Server and EDB Postgres Extended Server from version 15, and is supported by the EDB Postgres for Kubernetes operator.

> **Important**
>
> Before you proceed, please take some time to familiarize with the TDE feature in the EPAS documentation.

With TDE activated, both WAL files and files for tables will be encrypted. Data encryption/decryption is entirely transparent to the user, as it is managed by the database without requiring any application changes or updated client drivers.

> **Note**
>
> In the code samples shown below, the `epas` sub-section of `postgresql` in the YAML manifests is used to activate TDE. The `epas` section can be used to enable TDE for PG Extended images as well as for EPAS images.

EDB Postgres for Kubernetes provides 3 ways to use TDE:

- using a secret containing the passphrase
- using a secret containing a custom passphrase command
- using a pair of secrets containing custom wrap/unwrap commands

Passphrase secret

The basic approach is to store the passphrase in a Kubernetes secret. Such a passphrase will be used to encrypt the EPAS binary key.

> **EPAS documentation**
>
> Please refer to the EPAS documentation for details on the EPAS encryption key.

Activating TDE on the operator is simple. In the `epas` section of the manifest, use the `tde` stanza to enable TDE, and set the Kubernetes secret that will hold the TDE encryption key.

For example:

```
[…]
postgresql:
  epas:
    tde:
      enabled: true
      secretKeyRef:
        name: tde-key
        key:
key
```

You can find an example in `cluster-example-tde.yaml`.

> **Note**
>
> This file also contains the definition of the secret to hold the encryption key. Look at the following section for an example on how to create a secret for this purpose.

The key stored in the secret will be used as the pass-phrase to invoke `openssl` to wrap/unwrap the EPAS encryption key.

How to create the secret containing the passphrase

First choose the passphrase. While it is recommended to use a randomly generated passphrase, in this example we will use `PostgresRocks` as passphrase, and rely on `kubectl` to generate for us the secret definition:

```
kubectl create secret generic -o yaml tde-key \
    --from-literal=key=PostgresRocks
```

This should return something like this:

```
apiVersion: v1
data:
  key: UG9zdGdyZXNSb2Nrcw==
kind:
Secret
metadata:
  creationTimestamp: "YYYY-MM-DDTHH:MM:SSZ"
  name: tde-key
  namespace: default
  resourceVersion: ....
  uid: ....
type:
Opaque
```

Remember to run `kubectl apply` or remove the `-o yaml` option to the `create` command above to actually create the secret in the cluster.

Custom passphrase command

Instead of the `secretKeyRef` in the cluster manifest snippet above, it is possible to specify a `passphraseCommand` stored in a secret. The passphrase command can be run to generate a passphrase to be used with `openssl`.

```
  […]
  postgresql:
    epas:
      tde:
        enabled: true
        passphraseCommand:
          name: tde-passphrase
          key: command
```

The passphrase command should write to standard output. For example, we could simply use `echo my-passphrase`.

The passphrase generated by the command will be used the same way the secretKeyRef was used, i.e. as a passphrase argument for `openssl`.

Custom wrap/unwrap commands

It is also possible to specify the wrap and unwrap commands, rather than rely on the default invocation of `openssl`. This can be done by creating secrets containing the custom commands, and declaring those secrets in the `tde` stanza.

The snippet below shows a cluster with TDE enabled using custom commands.

```
[…]
postgresql:
  epas:
    tde:
      enabled: true
      wrapCommand:
        name: tde-wrap-command
        key: command
      unwrapCommand:
        name: tde-unwrap-command
        key: command
```

The custom commands need to obey the following conventions:

1. The custom wrap command should accept input from standard input, which EPAS will use to feed it the binary key. It should write to a file via an explicit argument (not shell redirections). Moreover, the file argument should be given the string "%p", which is a placeholder EPAS will use to pass the file path of the new, wrapped encryption key file.

2. The custom unwrap command should write to standard output. It should have an explicit file path argument for input (not shell redirections). Again, the file argument should be given the string "%p", which is the placeholder EPAS will fill in with the wrapped encryption key file path.

For example:

- wrap command: `openssl enc -aes-128-cbc -pass pass:temp-pass -e -out %p`
- unwrap command: `openssl enc -aes-128-cbc -pass pass:temp-pass -d -in %p`

Example using HashiCorp Vault

The following example shows how to use HashiCorp Vault to store the encryption key and use it to activate TDE. The `vault` CLI is used to interact with Vault and is included by default in the EDB Postgres Advanced Server (EPAS) image.

First, wherever you have vault running you must enable the Transit secrets engine and create a key:

```
vault secrets enable transit
vault write -f transit/keys/pg-tde
```

Then, create a secret containing the custom wrap/unwrap commands. The wrap and unwrap commands will 'wrap' a binary that is in the EPAS image. The binary will interact with the vault API to encrypt/decrypt the EPAS encryption.

The binary needs 5 flags: `--file`, `--host`, `--secret`, `--key` and `--vault-endpoint`. The `--host` flag is in the format of `http://vault-host:vault-port` and needs to be provided to reach the Vault. The server `--secret` flag is the name of the Kubernetes secret that contains the vault token and the `--key` flag is the key in that secret pointing the vault token. The `--vault-endpoint` flag is the name of the key that was created inside vault; in the example above it is `pg-tde`.

If running the Vault operator in Kubernetes the root token can be obtained from the following two commands:

```
kubectl exec vault-0 -- vault operator init -key-shares=1 -key-threshold=1 -format=json > cluster-keys.json
cat cluster-keys.json | jq -r ".root_token"
```

```
kubectl create secret generic -o yaml vault-token \
    --from-literal=wrap="/bin/vault wrap --file %p --host http://vault:8200 --secret vault-token --key
token --vault-endpoint pg-tde" \
    --from-literal=unwrap="/bin/vault unwrap --file %p --host http://vault:8200 --secret vault-token --key
token --vault-endpoint pg-tde" \
    --from-literal=token="hvs.whatever"
```

You can now create a Cluster that is referencing the secrets:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: hashicorp-vault-tde
spec:
  instances: 3
  storage:
    size:
1Gi
  postgresql:
    epas:
      tde:
        enabled: true
        wrapCommand:
          name: vault-
token
          key: wrap
        unwrapCommand:
          name: vault-
token
          key:
unwrap
```

Enable TLS

The binary needs 6 flags: `--file`, `--host`, `--secret`, `--key`, `--vault-endpoint` and `--enable-tls`. The `--host` flag is in the format of `http://vault-host:vault-port` and needs to be provided to reach the Vault. The server `--secret` flag is the name of the Kubernetes secret that contains the vault token and the `--key` flag is the key in that secret pointing the vault token. The `--vault-endpoint` flag is the name of the key that was created inside vault; in the example above it is `pg-tde`. To enable TLS, `--enable-tls` must be set to `true`.

```
kubectl create secret generic -o yaml vault-token \
    --from-literal=wrap="/bin/vault wrap --file %p --host http://vault:8200 --secret vault-token --key
token --vault-endpoint pg-tde --enable-tls true" \
    --from-literal=unwrap="/bin/vault unwrap --file %p --host http://vault:8200 --secret vault-token --key
token --vault-endpoint pg-tde --enable-tls true" \
    --from-literal=token="hvs.whatever"
```

Then, specify the environment variables in a secret or configMap.

```
kubectl create secret generic -o yaml env-var-secret \
--from-literal=VAULT_CACERT="/projected/certificate/vault-ca.pem" \
```

Reference the secret in the Cluster spec `envFrom` section, so that the binary can use the environment variables, as they will be injected in the pods. If one or more environment variables refers to files, rely on the `projectedVolumeTemplate` to mount custom files, and on secrets or configmaps for their contents. Following the example, the values of the `tls-vault-secret` key `ca` is mounted as file into the path `/projected/certificate/vault-ca.pem`.

```
kubectl create secret generic -o yaml tls-vault-secret --from-file=ca=vault-ca.crt
```

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: hashicorp-vault-tde
spec:
  envFrom:
  - secretRef:
      name: env-var-secret
  projectedVolumeTemplate:
    sources:
      - secret:
          name: tls-vault-secret
          items:
            - key: ca
              path: certificate/vault-ca.pem
  instances: 3
  storage:
    size:
1Gi
  postgresql:
    epas:
      tde:
        enabled: true
        wrapCommand:
          name: vault-
token
          key: wrap
        unwrapCommand:
          name: vault-
token
          key:
unwrap
```

51 Add-ons

EDB Postgres for Kubernetes supports add-ons that can be enabled on a per-cluster basis. These add-ons are:

1. External backup adapter
2. Kasten
3. Velero

> **Info**
>
> If you are planning to use Velero in OpenShift, please refer to theOADP section in the Openshift documentation.

All add-ons will automatically be available to the operator and to be used will need to be enabled at the cluster level via the `k8s.enterprisedb.io/addons` annotation.

External Backup Adapter

The external backup adapter add-ons provide a generic way to integrate EDB Postgres for Kubernetes in a third-party tool for backups through customizable ways to identify via labels and/or annotations:

- which PVC group to backup
- which PVCs to exclude, in case the cluster has one or more active replicas
- the Pod running the PostgreSQL instance that has been selected for the backup (a standby or, if not available, the primary)

You can choose between two add-ons that only differ from each other for the way they allow you to configure the adapter for your backup system:

- `external-backup-adapter` : in case you want to customize the behavior at the operator's configuration level via either a config map or a secret - and share it with all the Postgres clusters that are managed by the operator's deployment (see the `external-backup-adapter` section below)
- `external-backup-adapter-cluster` : in case you want to customize the behavior of the adapter at the Postgres cluster level, through a specific annotation (see the `external-backup-adapter-cluster` section below)

Such add-ons allow you to define the names of the annotations that will contain the commands to be run before or after taking a backup in the pod selected by the operator.

As a result, any third-party backup tool for Kubernetes can rely on the above to coordinate itself with a PostgreSQL cluster, or a set of them.

Recovery simply relies on the operator to reconcile the cluster from an existing PVC group.

> **Important**
>
> The External Backup Adapter is not a tool to perform backups. It simply provides a generic interface that any third-party backup tool in the Kubernetes space can use. Such tools are responsible for safely storing the PVC and/or the content, and make it available at recovery time together with all the necessary resource definitions of your Kubernetes cluster.

Customizing the adapter

As mentioned above, the adapter can be configured in two ways, which then determines the actual `add-on` you need to use in your `Cluster` resource.

If you are planning to define the same behavior for all the Postgres `Cluster` resources managed by the operator, we recommend that you use the `external-backup-adapter` add-on, and configure the annotations/labels in the operator's configuration.

If you are planning to have different behaviors for a subset of the Postgres `Cluster` resources that you have, we recommend that you use the `external-backup-adapter-cluster` add-on.

Both add-ons share the same capabilities in terms of customization, which needs to be defined as a YAML object having the following keys:

- `electedResourcesDecorators`
- `excludedResourcesDecorators`
- `excludedResourcesSelector`
- `backupInstanceDecorators`
- `preBackupHookConfiguration`
- `postBackupHookConfiguration`

Each section is explained below. Further down you'll find the instructions on how to customize each of the two add-ons, with some examples.

The `electedResourcesDecorators` section

This section allows you to configure an array of labels and/or annotations that will be put on every elected PVC group.

Each element of the array must have the following fields:

`key` : the name of the key for the label or annotation

`metadataType` : the type of metadata, either `"label"` or `"annotation"`

`value` : the value that will be assigned to the label or annotation

The `excludedResourcesDecorators` section

This section allows you to configure an array of labels and/or annotations that will be placed on every excluded pod and PVC.

Each element of the array must have the same fields as the `electedResourcesDecorators` section above.

The `excludedResourcesSelector` section

This section selects Pods and PVCs that are applied to the `excludedResourcesDecorators`. It accepts a label selector rule as value. When empty, all the Pods and every PVC that is not elected will be excluded.

The `backupInstanceDecorators` section

This section allows you to configure an array of labels and/or annotations that will be placed on the instance that has been selected for the backup by the operator and which contains the hooks to be run.

Each element of the array must have the same fields as the `electedResourcesDecorators` section above.

The `preBackupHookConfiguration` section

This section allows you to control the names of the annotations in which the operator will place the name of the container, the command to run before taking the backup, and the command to run in case of error/abort on the third-party tool side. Such metadata will be applied on the instance that's been selected by the operator for the backup (see `backupInstanceDecorators` above).

The following fields must be provided:

`container` : Specifies where to place the information about the container that will run the pre-backup command. The container name is a fixed value and cannot be configured. Will be saved in the annotations. To decorate the pod with hooks refer to: `instanceWithHookDecorators` `command` : Specifies where to place the information about the command that will be executed before the backup is taken. The command that will be executed is a fixed value and cannot be configured. Will be saved in the annotations. To decorate the pod with hooks refer to: `instanceWithHookDecorators`

`onError` : Specifies where to place the information about the command that will be executed in case of an error. The command that will be executed is a fixed value and cannot be configured. Will be saved in the annotations. To decorate the pod with hooks refer to: `instanceWithHookDecorators`

The `postBackupHookConfiguration` section

This section allows you to control the names of the annotations in which the operator will place the name of the container and the command to run after taking the backup. Such metadata will be applied on the instance that's been selected by the operator for the backup (see `backupInstanceDecorators` above).

The following fields must be provided:

`container` : Specifies where to place the information about the container that will run the post-backup command. The container name is a fixed value and cannot be configured. Will be saved in the annotations. To decorate the pod with hooks refer to: `instanceWithHookDecorators`

`command` : Specifies where to place the information about the command that will be executed after the backup is taken. The command that will be executed is a fixed value and cannot be configured. Will be saved in the annotations. To decorate the pod with hooks refer to: `instanceWithHookDecorators`

The `external-backup-adapter` add-on

The `external-backup-adapter` add-on can be entirely configured at operator's level via the `EXTERNAL_BACKUP_ADDON_CONFIGURATION` field in the operator's `ConfigMap` / `Secret` .

For more information, please refer to the provided sample file at the end of this section, or the example below:


```yaml
apiVersion: v1
kind: ConfigMap
metadata:
  name: postgresql-operator-controller-manager-
config
  namespace: postgresql-operator-
system
data:
  #
...
  EXTERNAL_BACKUP_ADDON_CONFIGURATION: |-
    electedResourcesDecorators:
      - key:
"app.example.com/elected"
        metadataType: "label"
        value:
"true"
    excludedResourcesSelector:
app=xyz,env=prod
    excludedResourcesDecorators:
      - key:
"app.example.com/excluded"
        metadataType: "label"
        value:
"true"
      - key: "app.example.com/excluded-
reason"
        metadataType: "annotation"
        value: "Not necessary for
backup"
    backupInstanceDecorators:
      - key:
"app.example.com/hasHooks"
        metadataType: "label"
        value:
"true"
    preBackupHookConfiguration:
      container:
        key: "app.example.com/pre-backup-
container"

command:
        key: "app.example.com/pre-backup-
command"

onError:
        key: "app.example.com/pre-backup-on-
error"
    postBackupHookConfiguration:
      container:
        key: "app.example.com/post-backup-container"

command:
        key: "app.example.com/post-backup-command"
```

The add-on can be activated by adding the following annotation to the `Cluster` resource:

```yaml
k8s.enterprisedb.io/addons: '["external-backup-adapter"]'
```

The `external-backup-adapter-cluster` add-on

The `external-backup-adapter-cluster` add-on must be configured in each `Cluster` resource you intend to use it through the `k8s.enterprisedb.io/externalBackupAdapterClusterConfig` annotation - which accepts the YAML object as content - as outlined in the following example:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
  annotations:
    "k8s.enterprisedb.io/addons": '["external-backup-adapter-cluster"]'
    "k8s.enterprisedb.io/externalBackupAdapterClusterConfig": |-
      electedResourcesDecorators:
        - key:
"app.example.com/elected"
          metadataType: "label"
          value:
"true"
      excludedResourcesSelector:
app=xyz,env=prod
      excludedResourcesDecorators:
        - key:
"app.example.com/excluded"
          metadataType: "label"
          value:
"true"
        - key: "app.example.com/excluded-
reason"
          metadataType: "annotation"
          value: "Not necessary for
backup"
      backupInstanceDecorators:
        - key:
"app.example.com/hasHooks"
          metadataType: "label"
          value:
"true"
      preBackupHookConfiguration:
        container:
          key: "app.example.com/pre-backup-
container"

command:
          key: "app.example.com/pre-backup-
command"

onError:
          key: "app.example.com/pre-backup-on-
error"
      postBackupHookConfiguration:
        container:
          key: "app.example.com/post-backup-container"

command:
          key: "app.example.com/post-backup-command"
spec:
  instances: 3
  storage:
    size:
1Gi
```

About the fencing annotation

If the configured external backup adapter backs up annotations, the fencing annotation will be set by the pre-backup hook and persist to the restored cluster. After restoring the cluster, you will need to manually remove the fencing annotation from the `Cluster` object to fix this.

This can be done with the `cnp` plugin for kubectl:

```
kubectl cnp fencing off <cluster-name>
```

Or, if you don't have the `cnp` plugin, you can remove the fencing annotation manually with the following command:

```
kubectl annotate cluster <cluster-name> k8s.enterprisedb.io/fencedInstances-
```

Please refer to the fencing documentation for more information.

Limitations

As far as the backup part is concerned, currently, the EDB Postgres for Kubernetes integration with `external-backup-adapter` and `external-backup-adapter-cluster` supports **cold backups** only. These are also referred to as **offline backups**. This means that the selected replica is temporarily fenced so that `external-backup-adapter` and `external-backup-adapter-cluster` can take a physical snapshot of the PVC group - namely the `PGDATA` volume and, where available, the WAL volume.

In this short timeframe, the standby cannot accept read-only connections. If no standby is available - usually because we're in a single instance cluster - and the annotation `k8s.enterprisedb.io/snapshotAllowColdBackupOnPrimary` is set to true, `external-backup-adapter` and `external-backup-adapter-cluster` will temporarily fence the primary, causing downtime in terms of read-write operations. This use case is normally left to development environments.

Full example of YAML file

Here is a full example of YAML content to be placed in either:

- the `EXTERNAL_BACKUP_ADDON_CONFIGURATION` option as part of the the operator's configuration process described above for the `external-backup-adapter` add-on, or
- in the `k8s.enterprisedb.io/externalBackupAdapterClusterConfig` annotation for the `external-backup-adapter-cluster` add-on

> **Hint**
>
> Copy the content below and paste it inside the `ConfigMap` or `Secret` that you use to configure the operator or the annotation in the `Cluster`, making sure you use the `|` character that YAML reserves for literals, as well as proper indentation. Use the comments to help you customize the options for your tool.

```yaml
# An array of labels and/or annotations that will be
placed
# on the elected PVC
group
electedResourcesDecorators:
  - key: "backup.example.com/elected"
    metadataType: "label"
    value: "true"

# An array of labels and/or annotations that will be
placed
# on every excluded pod and
PVC
excludedResourcesDecorators:
```



```
  - key: "backup.example.com/excluded"
    metadataType: "label"
    value: "true"
  - key: "backup.example.com/excluded-reason"
    metadataType: "annotation"
    value: "Not necessary for
backup"

# A LabelSelector containing the labels being used to filter
Pods
# and PVCs to decorate with
excludedResourcesDecorators.
# It accepts a label selector rule as
value.
# See https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/#label-
selectors
# When empty, all the Pods and every PVC that is not elected will be
excluded.
excludedResourcesSelector:
app=xyz,env=prod

# An array of labels and/or annotations that will be
placed
# on the instance pod that's been selected for the backup
by
# the operator and which contains the
hooks.
# At least one element is
required
backupInstanceDecorators:
  - key: "backup.example.com/hasHooks"
    metadataType: "label"
    value: "true"

# The pre-backup hook configuration allows you to control the
names
# of the annotations in which the operator will place the
container
# name, the command to run before taking the backup, and the
command
# to run in case of error/abort on the third-party tool
side.
# Such metadata will be applied on the instance that's been
selected
# by the operator for the backup (see
`backupInstanceDecorators`)
preBackupHookConfiguration:
  # Where to place the information about the container that will
run
  # the pre-backup command. The container name is a fixed value
and
  # cannot be configured. Will be saved in the
annotations.
  # To decorate the pod with the hooks refer to:
instanceWithHookDecorators
  container:
    key: "app.example.com/pre-backup-container"
  # Where to place the information about the command that will
be
  # executed before the backup is taken. The command is a fixed
value
  # and cannot be configured. Will be saved in the
annotations.
  # To decorate the pod with the hooks refer to:
instanceWithHookDecorators
  command:
    key: "app.example.com/pre-backup-command"
```



```
  # Where to place the information about the command that will
be
  # executed in case of an error on the third-party tool
side.
  # The command is a fixed value and cannot be
configured.
  # Will be saved in the
annotations.
  # To decorate the pod with the hooks refer to:
instanceWithHookDecorators
  onError:
    key: "app.example.com/pre-backup-on-error"

# The post-backup hook configuration allows you to control the
names
# of the annotations in which the operator will place the
container
# name and the command to run after taking the
backup.
# Such metadata will be applied on the instance that's been selected
by
# the operator for the backup (see
`backupInstanceDecorators`).
postBackupHookConfiguration:
  # Where to place the information about the container that will
run
  # the post-backup command. The container name is a fixed value
and
  # cannot be configured. Will be saved in the
annotations.
  # To decorate the pod with hooks refer to:
instanceWithHookDecorators
  container:
    key: "app.example.com/post-backup-container"
  # Where to place the information about the command that will
be
  # executed after the backup is taken. The command is a fixed
value
  # and cannot be configured. Will be saved in the
annotations.
  # To decorate the pod with hooks refer to:
instanceWithHookDecorators
  command:
    key: "app.example.com/post-backup-command"
```

Kasten

Kasten is a very popular data protection tool for Kubernetes, enabling backup and restore, disaster recovery, and application mobility for Kubernetes applications. For more information, see the Kasten website and the Kasten by Veeam Implementation Guide

In brief, to enable transparent integration with Kasten on an EDB Postgres for Kubernetes Cluster, you just need to add the `kasten` value to the `k8s.enterprisedb.io/addons` annotation in a `Cluster` spec. For example:


```
kind: Cluster
metadata:
  name: one-instance
  annotations:
     k8s.enterprisedb.io/addons: '["kasten"]'
     k8s.enterprisedb.io/snapshotAllowColdBackupOnPrimary: enabled
spec:
  instances: 1
  storage:
    size:
1Gi
  walStorage:
    size:
1Gi
```

Once the cluster is created and healthy, the operator will select the farthest ahead replica instance to be the designated backup and will add Kasten-specific backup hooks through annotations and labels to that instance.

> **Important**
>
> The operator will refuse to shut down a primary instance to take a cold backup unless the Cluster is annotated with `k8s.enterprisedb.io/snapshotAllowColdBackupOnPrimary: enabled`

For further guidance on how to configure and use Kasten, see the Implementation Guide'sConfiguration and Using sections.

Limitations

As far as the backup part is concerned, currently, the EDB Postgres for Kubernetes integration with Kasten supports**cold backups** only. These are also referred to as **offline backups**. This means that the selected replica is temporarily fenced so that external-backup-adapter can take a physical snapshot of the PVC group - namely the `PGDATA` volume and, where available, the WAL volume.

In this short timeframe, the standby cannot accept read-only connections. If no standby is available - usually because we're in a single instance cluster - and the annotation `k8s.enterprisedb.io/snapshotAllowColdBackupOnPrimary` is set to true, Kasten will temporarily fence the primary, causing downtime in terms of read-write operations. This use case is normally left to development environments.

In terms of recovery, the integration with Kasten supports snapshot recovery only. No Point-in-Time Recovery (PITR) is available at the moment with the Kasten add-on, and RPO is determined by the frequency of the snapshots in your Kasten environment. If your organization relies on Kasten, this usually is acceptable, but if you need PITR we recommend you look at the native continuous backup method on object stores.

Velero

Velero is an open-source tool to safely back up, restore, perform disaster recovery, and migrate Kubernetes cluster resources and persistent volumes. For more information, see the Velero documentation. To enable Velero compatibility with an EDB Postgres for Kubernetes Cluster, add the `velero` value to the `k8s.enterprisedb.io/addons` annotation in a Cluster spec. For example:

```yaml
kind: Cluster
metadata:
  name: one-instance
  annotations:
     k8s.enterprisedb.io/addons: '["velero"]'
     k8s.enterprisedb.io/snapshotAllowColdBackupOnPrimary: enabled
spec:
  instances: 1
  storage:
    size:
1Gi
  walStorage:
    size:
1Gi
```

Once the cluster is created and healthy, the operator will select the farthest ahead replica instance to be the designated backup and will add Velero-specific backup hooks as annotations to that instance.

These annotations are used by Velero to run the commands to prepare the Postgres instance to be backed up.

> **Important**
>
> The operator will refuse to shut down a primary instance to take a cold backup unless the Cluster is annotated with `k8s.enterprisedb.io/snapshotAllowColdBackupOnPrimary: enabled`

Limitations

As far as the backup part is concerned, currently, the EDB Postgres for Kubernetes integration with Velero supports**cold backups** only. These are also referred to as **offline backups**. This means that the selected replica is temporarily fenced so that external-backup-adapter can take a physical snapshot of the PVC group - namely the `PGDATA` volume and, where available, the WAL volume.

In this short timeframe, the standby cannot accept read-only connections. If no standby is available - usually because we're in a single instance cluster - and the annotation `k8s.enterprisedb.io/snapshotAllowColdBackupOnPrimary` is set to true, Velero will temporarily fence the primary, causing downtime in terms of read-write operations. This use case is normally left to development environments.

In terms of recovery, the integration with Velero supports snapshot recovery only, for now. No Point-in-Time Recovery (PITR) is available at the moment with the Velero add-on, and RPO is determined by the frequency of the snapshots in your Velero environment. If your organization relies on Velero, this usually is acceptable, but if you need PITR we recommend you look at the native continuous backup method on object stores.

Backup

By design, EDB Postgres for Kubernetes offloads as much of the backup functionality to Velero as possible, with the only requirement to make available the previously mentioned backup hooks. Since EDB Postgres for Kubernetes transparently sets all the needed configurations, and the rest is standard Velero, using Velero to backup a Postgres cluster is as straightforward as it would be for any other object. For example:

```sh
velero backup create mybackup
\
  --include-namespaces mynamespace
\
  -n velero-install-namespace
```

This command will create a standard Velero backup using the configured object storage and the configured Snapshot API.

> **Important**
>
> By default, the Velero add-on exclude only a few resources from the backup operation, namely pods and PVCs of the instances that have not been selected (as you recall, the operator tries to backup the PVCs of the first replica). However, you can use the options for the `velero backup` command to fine tune the resources you want to be part of your backup.

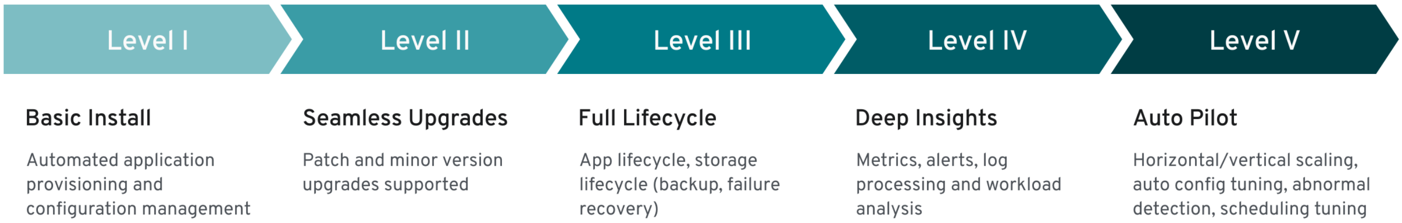
Restore

As with backup, the recovery process is a standard Velero procedure. The command to restore from a backup created with the above parameters would be:

```
velero create restore myrestore
\
  --from-backup mybackup
\
  -n velero-install-namespace
```

52 Operator capability levels

These capabilities were implemented by EDB Postgres for Kubernetes, classified using the Operator SDK definition of Capability Levels framework.



> **Important**
>
> Based on the Operator Capability Levels model, you can expect a "Level V - Auto Pilot" set of capabilities from the EDB Postgres for Kubernetes operator.

Each capability level is associated with a certain set of management features the operator offers:

1. Basic install
2. Seamless upgrades
3. Full lifecycle
4. Deep insights
5. Auto pilot

> **Note**
>
> We consider this framework as a guide for future work and implementations in the operator.

Level 1: Basic install

Capability level 1 involves installing and configuring the operator. This category includes usability and user experience enhancements, such as improvements in how you interact with the operator and a PostgreSQL cluster configuration.

> **Important**
>
> We consider information security part of this level.

Operator deployment via declarative configuration

The operator is installed in a declarative way using a Kubernetes manifest that defines four major `CustomResourceDefinition` objects: `Cluster`, `Pooler`, `Backup`, and `ScheduledBackup`.

PostgreSQL cluster deployment via declarative configuration

You define a PostgreSQL cluster (operand) using the `Cluster` custom resource in a fully declarative way. The PostgreSQL version is determined by the operand container image defined in the CR, which is automatically fetched from the requested registry. When deploying an operand, the operator also creates the following resources: `Pod`, `Service`, `Secret`, `ConfigMap`, `PersistentVolumeClaim`, `PodDisruptionBudget`, `ServiceAccount`, `RoleBinding`, and `Role`.

Override of operand images through the CRD

The operator is designed to support any operand container image with PostgreSQL inside. By default, the operator uses the latest available minor version of the latest stable major version supported by the PostgreSQL Community and published on quay.io by EDB. You can use any compatible image of PostgreSQL supporting the primary/standby architecture directly by setting the `imageName` attribute in the CR. The operator also supports `imagePullSecrets` to access private container registries, and it supports digests and tags for finer control of container image immutability. If you prefer not to specify an image name, you can leverage image catalogs by simply referencing the PostgreSQL major version. Moreover, image catalogs enable you to effortlessly create custom catalogs, directing to images based on your specific requirements.

Labels and annotations

You can configure the operator to support inheriting labels and annotations that are defined in a cluster's metadata. The goal is to improve the organization of the EDB Postgres for Kubernetes deployment in your Kubernetes infrastructure.

Self-contained instance manager

Instead of relying on an external tool to coordinate PostgreSQL instances in the Kubernetes cluster pods, such as Patroni or Stolon, the operator injects the operator executable inside each pod, in a file named `/controller/manager`. The application is used to control the underlying PostgreSQL instance and to reconcile the pod status with the instance based on the PostgreSQL cluster topology. The instance manager also starts a web server that's invoked by the `kubelet` for probes. Unix signals invoked by the `kubelet` are filtered by the instance manager. Where appropriate, they're forwarded to the `postgres` process for fast and controlled reactions to external events. The instance manager is written in Go and has no external dependencies.

Storage configuration

Storage is a critical component in a database workload. Taking advantage of the Kubernetes native capabilities and resources in terms of storage, the operator gives you enough flexibility to choose the right storage for your workload requirements, based on what the underlying Kubernetes environment can offer. This implies choosing a particular storage class in a public cloud environment or fine-tuning the generated PVC through a PVC template in the CR's `storage` parameter.

For better performance and finer control, you can also choose to host your cluster's write-ahead log (WAL, also known as `pg_wal`) on a separate volume, preferably on different storage. The "Benchmarking" section of the documentation provides detailed instructions on benchmarking both storage and the database before production. It relies on the `cnp` plugin to ensure optimal performance and reliability.

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.

EDB Postgres for Kubernetes manages replication slots for all the replicas in the HA cluster. The implementation is inspired by the previously proposed patch for PostgreSQL, called failover slots, and also supports user defined physical replication slots on the primary.

Service Configuration

By default, EDB Postgres for Kubernetes creates three Kubernetes services 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.

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 and schemas.

Although no configuration is required to run the cluster, you can customize both PostgreSQL runtime configuration and PostgreSQL host-based authentication rules in the `postgresql` section of the CR.

Configuration of Postgres roles, users, and groups

EDB Postgres for Kubernetes supports management of PostgreSQL roles, users, and groups through declarative configuration using the `.spec.managed.roles` stanza.

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.

On Red Hat OpenShift, Cloud Native PostgreSQL runs in `restricted` security context constraint (SCC), the most restrictive one - with the goal to limit the execution of a pod to a namespace allocated UID and SELinux context.

Affinity

The cluster's `affinity` section enables fine-tuning of how pods and related resources, such as persistent volumes, are scheduled across the nodes of a Kubernetes cluster. In particular, the operator supports:

- Pod affinity and anti-affinity
- Node selector
- Taints and tolerations

Topology spread constraints

The cluster's `topologySpreadConstraints` section enables additional control of scheduling pods across topologies, enhancing what affinity and anti-affinity can offer.

License keys

The operator comes with support for license keys, with the possibility to programmatically define a default behavior in case of the absence of a key. Cloud Native PostgreSQL has been programmed to create an implicit 30-day trial license for every deployed cluster. License keys are signed strings that the operator can verify using an asymmetric key technique. The content is a JSON object that includes the type, the product, the expiration date, and, if required, the cluster identifiers (namespace and name), the number of instances, the credentials to be used as a secret by the operator to pull down an image from a protected container registry. Beyond the expiration date, the operator will stop any reconciliation process until the license key is restored.

Command line interface

EDB Postgres for Kubernetes doesn't have its own command-line interface. It relies on the best command-line interface for Kubernetes, kubectl, by providing a plugin called `cnp`. This plugin enhances and simplifies your PostgreSQL cluster management experience.

Current status of the cluster

The operator continuously updates the status section of the CR with the observed status of the cluster. The entire PostgreSQL cluster status is continuously monitored by the instance manager running in each pod. The instance manager is responsible for applying the required changes to the controlled PostgreSQL instance to converge to the required status of the cluster. (For example, if the cluster status reports that pod `-1` is the primary, pod `-1` needs to promote itself while the other pods need to follow pod `-1`.) The same status is used by the `cnp` plugin for kubectl to provide details.

Operator's certification authority

The operator creates a certification authority for itself. It creates and signs with the operator certification authority a leaf certificate for the webhook server to use. This certificate ensures safe communication between the Kubernetes API server and the operator.

Cluster's certification authority

The operator creates a certification authority for every PostgreSQL cluster. This certification authority is used to issue and renew TLS certificates for clients' authentication, including streaming replication standby servers (instead of passwords). Support for a custom certification authority for client certificates is available through secrets, which also includes integration with cert-manager. Certificates can be issued with the `cnp` plugin for kubectl.

TLS connections

The operator transparently and natively supports TLS/SSL connections to encrypt client/server communications for increased security using the cluster's certification authority. Support for custom server certificates is available through secrets, which also includes integration with cert-manager.

Certificate authentication for streaming replication

To authorize streaming replication connections from the standby servers, the operator relies on TLS client certificate authentication. This method is used instead of relying on a password (and therefore a secret).

Continuous configuration management

The operator enables you to apply changes to the `Cluster` resource YAML section of the PostgreSQL configuration. Depending on the configuration option, it also makes sure that all instances are properly reloaded or restarted.

> **Current limitation**
>
> Changes with `ALTER SYSTEM` aren't detected, meaning that the cluster state isn't enforced.

Import of existing PostgreSQL databases

The operator provides a declarative way to import existing Postgres databases in a new EDB Postgres for Kubernetes cluster in Kubernetes, using offline migrations. The same feature also covers offline major upgrades of PostgreSQL databases. Offline means that applications must stop their write operations at the source until the database is imported. The feature extends the `initdb` bootstrap method to create a new PostgreSQL cluster using a logical snapshot of the data available in another PostgreSQL database. This data can be accessed by way of the network through a superuser connection. Import is from any supported version of Postgres. It relies on `pg_dump` and `pg_restore` being executed from the new cluster primary for all databases that are part of the operation and, if requested, for roles.

PostGIS clusters

EDB Postgres for Kubernetes supports the installation of clusters with the PostGIS open source extension for geographical databases. This extension is one of the most popular extensions for PostgreSQL.

Basic LDAP authentication for PostgreSQL

The operator allows you to configure LDAP authentication for your PostgreSQL clients, using either the *simple bind* or *search+bind* mode, as described in the LDAP authentication section of the PostgreSQL documentation.

Multiple installation methods

The operator can be installed through a Kubernetes manifest by way of `kubectl apply`, to be used in a traditional Kubernetes installation in public and private cloud environments. Additionally, it can be deployed through the OpenShift Container Platform by Red Hat. A Helm Chart for the operator is also available.

Convention over configuration

The operator supports the convention-over-configuration paradigm, deciding standard default values while allowing you to override them and customize them. You can specify a deployment of a PostgreSQL cluster using the `Cluster` CRD in a couple of lines of YAML code.

Level 2: Seamless upgrades

Capability level 2 is about enabling updates of the operator and the actual workload, in this case PostgreSQL servers. This includes PostgreSQL minor release updates (security and bug fixes normally) as well as major online upgrades.

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, EDB Postgres for Kubernetes 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.

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 `cnp` 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.

Offline In-Place Major Upgrades of PostgreSQL

EDB Postgres for Kubernetes 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.

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`.

Level 3: Full lifecycle

Capability level 3 requires the operator to manage aspects of business continuity and scalability.

Disaster recovery is a business continuity component that requires that both backup and recovery of a database work correctly. While as a starting point, the goal is to achieve RPO < 5 minutes, the long-term goal is to implement RPO=0 backup solutions. *High availability* is the other important component of business continuity. Through PostgreSQL native physical replication and hot standby replicas, it allows the operator to perform failover and switchover operations. This area includes enhancements in:

- Control of PostgreSQL physical replication, such as synchronous replication, (cascading) replication clusters, and so on
- Connection pooling, to improve performance and control through a connection pooling layer with pgBouncer

PostgreSQL WAL archive

The operator supports PostgreSQL continuous archiving of WAL files to an object store (AWS S3 and S3-compatible, Azure Blob Storage, Google Cloud Storage, and gateways like MinIO).

WAL archiving is defined at the cluster level, declaratively, through the `backup` parameter in the cluster definition. This is done by specifying an S3 protocol destination URL (for example, to point to a specific folder in an AWS S3 bucket) and, optionally, a generic endpoint URL.

WAL archiving, a prerequisite for continuous backup, doesn't require any further user action. The operator transparently sets the `archive_command` to rely on `barman-cloud-wal-archive` to ship WAL files to the defined endpoint. You can decide the compression algorithm, as well as the number of parallel jobs to concurrently upload WAL files in the archive. In addition, `Instance Manager` checks the correctness of the archive destination by performing the `barman-cloud-check-wal-archive` command before beginning to ship the first set of WAL files.

PostgreSQL Backups

EDB Postgres for Kubernetes provides a pluggable interface (CNP-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 EDB Postgres for Kubernetes 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 CNP-I plugins, EDB Postgres for Kubernetes 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 CNP-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.

Backups from a standby

The operator supports offloading base backups onto a standby without impacting theRPO of the database. This allows resources to be preserved on the primary, in particular I/O, for standard database operations.

Full restore from a backup

The operator enables you to bootstrap a new cluster (with its settings) starting from an existing and accessible backup, either on a volume snapshot, or in an object store, 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. EDB Postgres for Kubernetes supports parallel WAL fetching from the archive.

Point-in-time recovery (PITR) from a backup

The operator enables you to create a new PostgreSQL cluster by recovering an existing backup to a specific point in time, defined with a timestamp, a label, or a transaction ID. This capability is built on top of the full restore one and supports all the options available in PostgreSQL for PITR.

Zero-Data-Loss Clusters Through Synchronous Replication

Achieve *zero data loss*(RPO=0) in your local high-availability EDB Postgres for Kubernetes 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.

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) in the event of unintended errors, such as incorrect `DELETE` or `UPDATE` SQL operations.

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 EDB Postgres for Kubernetes, 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 EDB Postgres for Kubernetes, ensuring robust data integrity and continuity across diverse environments.

Tablespace support

EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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.

Customizable Startup, Liveness, and Readiness Probes

EDB Postgres for Kubernetes 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.

Rolling deployments

The operator supports rolling deployments to minimize the downtime. If a PostgreSQL cluster is exposed publicly, the service load-balances the read-only traffic only to available pods during the initialization or the update.

Scale up and down of replicas

The operator allows you to scale up and down the number of instances in a PostgreSQL cluster. New replicas are started up from the primary server and participate in the cluster's HA infrastructure. The CRD declares a "scale" subresource that allows you to use the `kubectl scale` command.

Maintenance window and PodDisruptionBudget for Kubernetes nodes

The operator creates a `PodDisruptionBudget` resource to limit the number of concurrent disruptions to one primary instance. This configuration prevents the maintenance operation from deleting all the pods in a cluster, allowing the specified number of instances to be created. The PodDisruptionBudget is applied during the node-draining operation, preventing any disruption of the cluster service.

While this strategy is correct for Kubernetes clusters where storage is shared among all the worker nodes, it might not be the best solution for clusters using local storage or for clusters installed in a private cloud. The operator allows you to specify a maintenance window and configure the reaction to any underlying node eviction. The `ReusePVC` option in the maintenance window section enables to specify the strategy to use. Allocate new storage in a different PVC for the evicted instance, or wait for the underlying node to be available again.

Fencing

Fencing is the process of protecting the data in one, more, or even all instances of a PostgreSQL cluster when they appear to be malfunctioning. When an instance is fenced, the PostgreSQL server process is guaranteed to be shut down, while the pod is kept running. This ensures that, until the fence is lifted, data on the pod isn't modified by PostgreSQL and that you can investigate file system for debugging and troubleshooting purposes.

Hibernation

EDB Postgres for Kubernetes supports hibernation of a running PostgreSQL cluster in a declarative manner, through the `k8s.enterprisedb.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.

Reuse of persistent volumes storage in pods

When the operator needs to create a pod that was deleted by the user or was evicted by a Kubernetes maintenance operation, it reuses the `PersistentVolumeClaim`, if available. This ability avoids the need to clone the data from the primary again.

CPU and memory requests and limits

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

Connection pooling with PgBouncer

EDB Postgres for Kubernetes provides native support for connection pooling with PgBouncer, 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.

Logical Replication

EDB Postgres for Kubernetes 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.

Integration with external backup tools for Kubernetes

EDB Postgres for Kubernetes provides add-ons to integrate with:

- Kasten, a very popular data protection tool for Kubernetes, enabling backup and restore, disaster recovery, and application mobility for cloud native applications
- Velero, a very popular open source tool to back up and restore Kubernetes resources and persistent volumes and OpenShift API for Data Protection (OADP)

Moreover, the external backup adapter add-on provides a generic interface to integrate EDB Postgres for Kubernetes in any third-party tool for backups.

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.

EDB Postgres for Kubernetes was designed to provide everything needed to easily integrate with industry-standard and community-accepted tools for flexible monitoring and logging.

Prometheus exporter with configurable queries

The instance manager provides a pluggable framework. By way of its own web server listening on the `metrics` port (9187), it exposes an endpoint to export metrics for the Prometheus monitoring and alerting tool. The operator supports custom monitoring queries defined as `ConfigMap` or `Secret` objects using a syntax that's compatible with `postgres_exporter` for Prometheus. EDB Postgres for Kubernetes provides a set of basic monitoring queries for PostgreSQL that can be integrated and adapted to your context.

Grafana dashboard

EDB Postgres for Kubernetes comes with a Grafana dashboard that you can use as a base to monitor all critical aspects of a PostgreSQL cluster, and customize.

Standard output logging of PostgreSQL error messages in JSON format

Every log message is delivered to standard output in JSON format. The first level is the definition of the timestamp, the log level, and the type of log entry, such as `postgres` for the canonical PostgreSQL error message channel. As a result, every pod managed by EDB Postgres for Kubernetes can be easily and directly integrated with any downstream log processing stack that supports JSON as source data type.

Real-time query monitoring

EDB Postgres for Kubernetes transparently and natively supports:

- The essential `pg_stat_statements` extension, which enables tracking of planning and execution statistics of all SQL statements executed by a PostgreSQL server
- The `auto_explain` extension, 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` extension, which makes logical replication slots usable across a physical failover, ensuring resilience in change data capture (CDC) contexts based on PostgreSQL's native logical replication

Audit

EDB Postgres for Kubernetes allows database and security administrators, auditors, and operators to track and analyze database activities using PGAudit for PostgreSQL and the EDB Audit Logging functionality (for EDB Postgres Advanced). Such activities flow directly in the JSON log and can be properly routed to the correct downstream target using common log brokers like Fluentd.

Kubernetes events

Record major events as expected by the Kubernetes API, such as creating resources, removing nodes, and upgrading. Events can be displayed by using the `kubectl describe` and `kubectl get events` commands.

Level 5: Auto pilot

Capability level 5 is focused on automated scaling, healing, and tuning through the discovery of anomalies and insights that emerged from the observability layer.

Automated failover for self-healing

In case of detected failure on the primary, the operator changes the status of the cluster by setting the most aligned replica as the new target primary. As a consequence, the instance manager in each alive pod initiates the required procedures to align itself with the requested status of the cluster. It does this by either becoming the new primary or by following it. In case the former primary comes back up, the same mechanism avoids a split-brain by preventing applications from reaching it, running `pg_rewind` on the server and restarting it as a standby.

Automated recreation of a standby

If the pod hosting a standby is removed, the operator initiates the procedure to re-create a standby server.

53 Frequently Asked Questions (FAQ)

Running PostgreSQL in Kubernetes

Everyone knows that stateful workloads like PostgreSQL cannot run in Kubernetes. Why do you say the contrary?

An *independent research survey commissioned by the Data on Kubernetes Community* in September 2021 revealed that half of the respondents run most of their production workloads on Kubernetes. 90% of them believe that Kubernetes is ready for stateful workloads, and 70% of them run databases in production. Databases like Postgres. However, according to them, significant challenges remain, such as the knowledge gap (Kubernetes and Cloud Native, in general, have a steep learning curve) and the quality of Kubernetes operators. The latter is the reason why we believe that an operator like EDB Postgres for Kubernetes 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*.

EDB Postgres for Kubernetes 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/EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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.

EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes before making your decision. You will see that most of these operators use an external failover management tool (Patroni or similar) and rely on StatefulSets.

Here is a non exhaustive list, in chronological order from their publication on GitHub:

- Crunchy Data Postgres Operator (2017)
- Zalando Postgres Operator (2017)
- Stackgres (2020)
- Percona Operator for PostgreSQL (2021)
- Kubegres (2021)

Feel free to report any relevant missing entry as a PR.

> **Info**
>
> 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 EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes using StatefulSets?

EDB Postgres for Kubernetes does not rely on `StatefulSet` resources, and instead manages the underlying PVCs directly by leveraging the selected storage class for dynamic provisioning. Please refer to the "Custom Pod Controller" section for details and reasons behind this decision.

High availability

What happens to the PostgreSQL clusters when the operator pod dies or it is not available for a certain amount of time?

The EDB Postgres for Kubernetes 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 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 EDB Postgres for Kubernetes not relying on a failover management tool like Patroni, repmgr, or Stolon?

Although part of the team that develops EDB Postgres for Kubernetes 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 instance manager
- control the Kubernetes services, that is the entry points for your applications

Should I manually resync a former primary with the new one following a failover?

No. The operator does that automatically for you, and relies on `pg_rewind` to synchronize the former primary with the new one.

Database management

Why should I use PostgreSQL?

We believe that PostgreSQL is the equivalent in the database area of what Linux represents in the operating system space. The current latest major version of Postgres is version 16, which ships out of the box:

- native streaming replication, both physical and logical
- continuous hot backup and point in time recovery
- declarative partitioning for horizontal table partitioning, which is a very well-known technique in the database area to improve vertical scalability on a single instance
- extensibility, with extensions like PostGIS for geographical databases
- parallel queries for vertical scalability
- JSON support, unleashing the multi-model hybrid database for both structured and unstructured data queried via standard SQL

And so on ...

How many databases should be hosted in a single PostgreSQL instance?

Our recommendation is to dedicate a single PostgreSQL cluster (intended as primary and multiple standby servers) to a single database, entirely managed by a single microservice application. However, by leveraging the "postgres" superuser, it is possible to create as many users and databases as desired (subject to the available resources).

The reason for this recommendation lies in the Cloud Native concept, based on microservices. In a pure microservice architecture, the microservice itself should own the data it manages exclusively. These could be flat files, queues, key-value stores, or, in our case, a PostgreSQL relational database containing both structured and unstructured data. The general idea is that only the microservice can access the database, including schema management and migrations.

EDB Postgres for Kubernetes has been designed to work this way out of the box, by default creating an application user and an application database owned by the aforementioned application user.

Reserving a PostgreSQL instance to a single microservice owned database, enhances:

- resource management: in PostgreSQL, CPU, and memory constrained resources are generally handled at the instance level, not the database level, making it easier to integrate it with Kubernetes resource management policies at the pod level
- physical continuous backup and Point-In-Time-Recovery (PITR): given that PostgreSQL handles continuous backup and recovery at the instance level, having one database per instance simplifies PITR operations, differentiates retention policy management, and increases data protection of backups
- application updates: enable each application to decide their update policies without impacting other databases owned by different applications
- database updates: each application can decide which PostgreSQL version to use, and independently, when to upgrade to a different major version of PostgreSQL and at what conditions (e.g., cutover time)

Is there an upper limit in database size for not considering Kubernetes?

No, as Kubernetes is no different from virtual machines and bare metal as far as this is regarded. Practically, however, it depends on the available resources of your Kubernetes cluster. Our advice with very large databases (VLDB) is to consider a shared nothing architecture, where a Kubernetes worker node is dedicated to a single Postgres instance, with dedicated storage. We proved that this extreme architectural pattern works when we benchmarked running PostgreSQL on bare metal Kubernetes with local persistent volumes. Tablespaces and horizontal partitioning are data modeling techniques that you can use to improve the vertical scalability of you databases.

How can I specify a time zone in the PostgreSQL cluster?

PostgreSQL has an extensive support for time zones, as explained in the official documentation:

- Date time data types
- Client connections config options

Although time zones can even be used at session, transaction and even as part of a query in PostgreSQL, a very common way is to set them up globally. With EDB Postgres for Kubernetes you can configure the cluster level time zone in the `.spec.postgresql.parameters` section as in the following example:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: pg-
italy
spec:
  instances: 1

  postgresql:
    parameters:
      timezone: "Europe/Rome"

  storage:
    size:
1Gi
```

The time zone can be verified with:

```
$ kubectl exec -ti pg-italy-1 -c postgres -- psql -x -c "SHOW timezone"
-[ RECORD 1 ]---------
TimeZone | Europe/Rome
```

What is the recommended architecture for best business continuity outcomes?

As covered in the "Architecture" section, the main recommendation is to adopt shared nothing architectures as much as possible, by leveraging the native capabilities and resources that Kubernetes provides in a single cluster, namely:

- availability zones: spread your instances across different availability zones in the same Kubernetes cluster
- worker nodes: as a consequence, make sure that your Postgres instances reside on different Kubernetes worker nodes
- storage: use dedicated storage for each worker node running Postgres

Use at least one standby, preferably at least two, so that you can configure synchronous replication in the cluster, introducingRPO=0 for high availability.

If you do not have availability zones - normally the case of on-premise installations - separate on worker nodes and storage.

Properly setup continuous backup on a local/regional object store.

The same architecture that is in a single Kubernetes cluster can be replicated in another Kubernetes cluster (normally in another geographical area or region) through the replica cluster feature, providing disaster recovery and high availability at global scale.

You can use the WAL archive in the primary object store to feed the replica in the other region, without having to provide a streaming connection, if the default maximum RPO of 5 minutes is enough for you.

How can instances be stopped or started?

Please look at "Fencing" or "Hibernation".

What are the global objects such as roles and databases that are automatically created by EDB Postgres for Kubernetes?

The operator automatically creates a user for the application (by default called `app`) and a database for the application (by default called `app`) which is owned by the aforementioned user.

This way, the database is ready for a microservice adoption, as developers can control migrations using the `app` user, without requiring *superuser* access.

Teams can then create another user for read-write operations through the "Declarative role management" feature and assign the required `GRANT` to the tables.

54 Troubleshooting

In this page, you can find some basic information on how to troubleshoot EDB Postgres for Kubernetes in your Kubernetes cluster deployment.

> **Hint**
>
> As a Kubernetes administrator, you should have the `kubectl` Cheat Sheet page bookmarked!

Before you start

Kubernetes environment

What can make a difference in a troubleshooting activity is to provide clear information about the underlying Kubernetes system.

Make sure you know:

- the Kubernetes distribution and version you are using
- the specifications of the nodes where PostgreSQL is running
- as much as you can about the actual storage, including storage class and benchmarks you have done before going into production.
- which relevant Kubernetes applications you are using in your cluster (i.e. Prometheus, Grafana, Istio, Certmanager, ...)
- the situation of continuous backup, in particular if it's in place and working correctly: in case it is not, make sure you take an emergency backup before performing any potential disrupting operation

Useful utilities

On top of the mandatory `kubectl` utility, for troubleshooting, we recommend the following plugins/utilities to be available in your system:

- `cnp` plugin for `kubectl`
- `jq` , a lightweight and flexible command-line JSON processor
- `grep` , searches one or more input files for lines containing a match to a specified pattern. It is already available in most *nix distros. If you are on Windows OS, you can use `findstr` as an alternative to `grep` or directly use `wsl` and install your preferred *nix distro and use the tools mentioned above.

First steps

To quickly get an overview of the cluster or installation, the `kubectl` plugin is the primary tool to use:

1. the status subcommand provides an overview of a cluster
2. the report subcommand provides the manifests for clusters and the operator deployment. It can also include logs using the `--logs` option. The report generated via the plugin will include the full cluster manifest.

The plugin can be installed on air-gapped systems via packages. Please refer to the plugin document for complete instructions.

Are there backups?

After getting the cluster manifest with the plugin, you should verify if backups are set up and working.

In a cluster with backups set up, you will find, in the cluster Status, the fields `lastSuccessfulBackup` and `firstRecoverabilityPoint`. You should make sure there is a recent `lastSuccessfulBackup`.

A cluster lacking the `.spec.backup` stanza won't have backups. An insistent message will appear in the PostgreSQL logs:

```
Backup not configured, skip WAL archiving.
```

Before proceeding with troubleshooting operations, it may be advisable to perform an emergency backup depending on your findings regarding backups. Refer to the following section for instructions.

It is **extremely risky** to operate a production database without keeping regular backups.

Emergency backup

In some emergency situations, you might need to take an emergency logical backup of the main `app` database.

> **Important**
>
> The instructions you find below must be executed only in emergency situations and the temporary backup files kept under the data protection policies that are effective in your organization. The dump file is indeed stored in the client machine that runs the `kubectl` command, so make sure that all protections are in place and you have enough space to store the backup file.

The following example shows how to take a logical backup of the `app` database in the `cluster-example` Postgres cluster, from the `cluster-example-1` pod:

```
kubectl exec cluster-example-1 -c postgres
\
  -- pg_dump -Fc -d app >
app.dump
```

> **Note**
>
> You can easily adapt the above command to backup your cluster, by providing the names of the objects you have used in your environment.

The above command issues a `pg_dump` command in custom format, which is the most versatile way to take logical backups in PostgreSQL.

The next step is to restore the database. We assume that you are operating on a new PostgreSQL cluster that's been just initialized (so the `app` database is empty).

The following example shows how to restore the above logical backup in the `app` database of the `new-cluster-example` Postgres cluster, by connecting to the primary (`new-cluster-example-1` pod):

```
kubectl exec -i new-cluster-example-1 -c postgres
\
  -- pg_restore --no-owner --role=app -d app --verbose <
app.dump
```

> **Important**
>
> The example in this section assumes that you have no other global objects (databases and roles) to dump and restore, as per our recommendation. In case you have multiple roles, make sure you have taken a backup using `pg_dumpall -g` and you manually restore them in the new cluster. In case you have multiple databases, you need to repeat the above operation one database at a time, making sure you assign the right ownership. If you are not familiar with PostgreSQL, we advise that you do these critical operations under the guidance of a professional support company.

The above steps might be integrated into the `cnp` plugin at some stage in the future.

Logs

All resources created and managed by EDB Postgres for Kubernetes log to standard output in accordance with Kubernetes conventions, using JSON format.

While logs are typically processed at the infrastructure level and include those from EDB Postgres for Kubernetes, 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 cnp logs` command for EDB Postgres for Kubernetes-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 `k8s.enterprisedb.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 EDB Postgres for Kubernetes.

Operator information

By default, the EDB Postgres for Kubernetes operator is installed in the `postgresql-operator-system` namespace in Kubernetes as a `Deployment` (see the "Details about the deployment" section for details).

You can get a list of the operator pods by running:

```
kubectl get pods -n postgresql-operator-system
```

> **Note**
>
> Under normal circumstances, you should have one pod where the operator is running, identified by a name starting with `postgresql-operator-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 `postgresql-operator-controller-manager`.

Collect the relevant information about the operator that is running in pod `<POD>` with:

```
kubectl describe pod -n postgresql-operator-system <POD>
```

Then get the logs from the same pod by running:

```
kubectl logs -n postgresql-operator-system <POD>
```

Gather more information about the operator

Get logs from all pods in EDB Postgres for Kubernetes operator Deployment (in case you have a multi operator deployment) by running:

```
kubectl logs -n postgresql-operator-system \
  deployment/postgresql-operator-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 postgresql-operator-system \
  deployment/postgresql-operator-controller-manager --all-containers=true | \
  jq -r . > cnp_logs.json
```

Get EDB Postgres for Kubernetes operator version by using `kubectl-cnp` plugin:

```
kubectl-cnp status <CLUSTER>
```

Output:

```
Cluster in healthy state
Name:               cluster-example
Namespace:          default
System ID:          7044925089871458324
PostgreSQL Image:   quay.io/enterprisedb/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
```

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 `cnp` plugin:

```
kubectl cnp status -n <NAMESPACE> <CLUSTER>
```

> **Tip**
>
> You can print more information by adding the `--verbose` option.

Get EDB PostgreSQL Advanced Server (EPAS) / PostgreSQL container image version:

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

Output:

```
  Image Name:    quay.io/enterprisedb/postgresql:17.5-3
```

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

Pod information

You can retrieve the list of instances that belong to a given PostgreSQL cluster with:

```
# using labels available from CNP 1.12.0
kubectl get pod -l k8s.enterprisedb.io/cluster=<CLUSTER> -L role -n <NAMESPACE>
# using legacy labels
kubectl get pod -l postgresql=<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'
```

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
```

Backup information

You can list the backups that have been created for a named cluster with:

```
kubectl get backup -l k8s.enterprisedb.io/cluster=<CLUSTER>
```

Storage information

Sometimes is useful to double-check the StorageClass used by the cluster to have some more context during investigations or troubleshooting, like this:

```
STORAGECLASS=$(kubectl get pvc <POD> -o jsonpath='{.spec.storageClassName}')
kubectl get storageclasses $STORAGECLASS -o yaml
```

We are taking the StorageClass from one of the cluster pod here since often clusters are created using the default StorageClass.

Node information

Kubernetes nodes is where ultimately PostgreSQL pods will be running. It's strategically important to know as much as we can about them.

You can get the list of nodes in your Kubernetes cluster with:

```
# look at the worker nodes and their status
kubectl get nodes -o wide
```

Additionally, you can gather the list of nodes where the pods of a given cluster are running with:

```
kubectl get pod -l k8s.enterprisedb.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 usingaffinity/anti-affinity rules and/or tolerations.

Conditions

Like many native kubernetes objects like here, Cluster exposes `status.conditions` as well. This allows one to 'wait' for a particular event to occur instead of relying on the overall cluster health state. Available conditions as of now are:

- LastBackupSucceeded
- ContinuousArchiving
- Ready

`LastBackupSucceeded` is reporting the status of the latest backup. If set to `True` the last backup has been taken correctly, it is set to `False` otherwise.

`ContinuousArchiving` is reporting the status of the WAL archiving. If set to `True` the last WAL archival process has been terminated correctly, it is set to `False` otherwise.

`Ready` is `True` when the cluster has the number of instances specified by the user and the primary instance is ready. This condition can be used in scripts to wait for the cluster to be created.

How to wait for a particular condition

- Backup:

```
$ kubectl wait --for=condition=LastBackupSucceeded cluster/<CLUSTER-NAME> -n
<NAMESPACE>
```

- ContinuousArchiving:

```
$ kubectl wait --for=condition=ContinuousArchiving cluster/<CLUSTER-NAME> -n
<NAMESPACE>
```

- Ready (Cluster is ready or not):

```
$ kubectl wait --for=condition=Ready cluster/<CLUSTER-NAME> -n
<NAMESPACE>
```

Below is a snippet of a `cluster.status` that contains a failing condition.

```
$ kubectl get cluster/<cluster-name> -o
yaml
.
.
.
  status:
    conditions:
    - message: 'unexpected failure invoking barman-cloud-wal-archive: exit
status
        2'
      reason:
ContinuousArchivingFailing
      status: "False"
      type:
ContinuousArchiving

    - message: exit status
2
      reason:
LastBackupFailed
      status: "False"
      type:
LastBackupSucceeded

    - message: Cluster Is Not
Ready
      reason: ClusterIsNotReady
      status: "False"
      type: Ready
```

Networking

EDB Postgres for Kubernetes requires basic networking and connectivity in place. You can find more information in the networking section.

If installing EDB Postgres for Kubernetes 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        k8s.enterprisedb.io/cluster=cluster-example
47m
default-deny-ingress    <none>
57m
```

PostgreSQL core dumps

Although rare, PostgreSQL can sometimes crash and generate a core dump in the `PGDATA` folder. When that happens, normally it is a bug in PostgreSQL (and most likely it has already been solved - this is why it is important to always run the latest minor version of PostgreSQL).

EDB Postgres for Kubernetes allows you to control what to include in the core dump through the `k8s.enterprisedb.io/coredumpFilter` annotation.

> **Info**
>
> Please refer to "Labels and annotations" for more details on the standard annotations that EDB Postgres for Kubernetes provides.

By default, the `k8s.enterprisedb.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.

> **Info**
>
> Please refer to "Core dump filtering settings" section of "The `/proc` Filesystem" page of the Linux Kernel documentation. for more details on how to set the bitmask that controls the core dump filter.

> **Important**
>
> Beware that this setting only takes effect during Pod startup and that changing the annotation doesn't trigger an automated rollout of the instances.

Although you might not personally be involved in inspecting core dumps, you might be asked to provide them so that a Postgres expert can look into them. First, verify that you have a core dump in the `PGDATA` directory with the following command (please run it against the correct pod where the Postgres instance is running):

```
kubectl exec -ti POD -c postgres
\
  -- find /var/lib/postgresql/data/pgdata -name
'core.*'
```

Under normal circumstances, this should return an empty set. Suppose, for example, that we have a core dump file:

```
/var/lib/postgresql/data/pgdata/core.14177
```

Once you have verified the space on disk is sufficient, you can collect the core dump on your machine through `kubectl cp` as follows:

```
kubectl cp POD:/var/lib/postgresql/data/pgdata/core.14177 core.14177
```

You now have the file. Make sure you free the space on the server by removing the core dumps.

Some known issues

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" section 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" section

Pods are stuck in `Pending` state

In case a Cluster's instance is stuck in the `Pending` phase, you should check the pod's `Events` section to get an idea of the reasons behind this:

```
kubectl describe pod -n <NAMESPACE> <POD>
```

Some of the possible causes for this are:

- No nodes are matching the `nodeSelector`
- Tolerations are not correctly configured to match the nodes' taints
- No nodes are available at all: this could also be related to `cluster-autoscaler` hitting some limits, or having some temporary issues

In this case, it could also be useful to check events in the namespace:

```
kubectl get events -n <NAMESPACE>
# list events in chronological order
kubectl get events -n <NAMESPACE> --sort-by=.metadata.creationTimestamp
```

Replicas out of sync when no backup is configured

Sometimes replicas might be switched off for a bit of time due to maintenance reasons (think of when a Kubernetes nodes is drained). In case your cluster does not have backup configured, when replicas come back up, they might require a WAL file that is not present anymore on the primary (having been already recycled according to the WAL management policies as mentioned in "The `postgresql` section"), and fall out of synchronization.

Similarly, when `pg_rewind` might require a WAL file that is not present anymore in the former primary, reporting `pg_rewind: error: could not open file`.

In these cases, pods cannot become ready anymore, and you are required to delete the PVC and let the operator rebuild the replica.

If you rely on dynamically provisioned Persistent Volumes, and you are confident in deleting the PV itself, you can do so with:

```
PODNAME=<POD>
VOLNAME=$(kubectl get pv -o json | \
  jq -r '.items[]|select(.spec.claimRef.name=='\"$PODNAME\"')|.metadata.name')

kubectl delete pod/$PODNAME pvc/$PODNAME pvc/$PODNAME-wal pv/$VOLNAME
```

Cluster stuck in `Creating new replica`

Cluster is stuck in "Creating a new replica", while pod logs don't show relevant problems. This has been found to be related to the next issue on connectivity. Networking issues are reflected in the status column as follows:

```
Instance Status Extraction Error: HTTP communication issue
```

Networking is impaired by installed Network Policies

As pointed out in the networking section, local network policies could prevent some of the required connectivity.

A tell-tale sign that connectivity is impaired is the presence in the operator logs of messages like:

```
"Cannot extract Pod status", […snipped…] "Get \"http://<pod IP>:8000/pg/status\": dial tcp <pod IP>:8000:
i/o timeout"
```

You should list the network policies, and look for any policies restricting connectivity.

```
$ kubectl get networkpolicies
NAME                   POD-SELECTOR
AGE
allow-prometheus       k8s.enterprisedb.io/cluster=cluster-example
47m
default-deny-ingress   <none>
57m
```

For example, in the listing above, `default-deny-ingress` seems a likely culprit. You can drill into it:

```
$ kubectl get networkpolicies default-deny-ingress -o
yaml
<…snipped…>
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

In the networking page you can find a network policy file that you can customize to create a `NetworkPolicy` explicitly allowing the operator to connect cross-namespace to cluster pods.

Error while bootstrapping the data directory

If your Cluster's initialization job crashes with a "Bus error (core dumped) child process exited with exit code 135", you likely need to fix the Cluster hugepages settings.

The reason is the incomplete support of hugepages in the cgroup v1 that should be fixed in v2. For more information, check the PostgreSQLBUG #17757: Not honoring huge_pages setting during initdb causes DB crash in Kubernetes.

To check whether hugepages are enabled, run `grep HugePages /proc/meminfo` on the Kubernetes node and check if hugepages are present, their size, and how many are free.

If the hugepages are present, you need to configure how much hugepages memory every PostgreSQL pod should have available.

For example:

```
  postgresql:
    parameters:
      shared_buffers: "128MB"

  resources:
    requests:
      memory: "512Mi"
    limits:
      hugepages-2Mi: "512Mi"
```

Please remember that you must have enough hugepages memory available to schedule every Pod in the Cluster (in the example above, at least 512MiB per Pod must be free).

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.

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.

55 API Reference - v1.26.0

Package v1 contains API Schema definitions for the postgresql v1 API group

Resource Types

- Backup
- Cluster
- ClusterImageCatalog
- Database
- ImageCatalog
- Pooler
- Publication
- ScheduledBackup
- Subscription

Backup

A Backup resource is a request for a PostgreSQL backup by the user.

Field	Description
`apiVersion` **[Required]** string	`postgresql.k8s.enterprisedb.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

Cluster

Cluster is the Schema for the PostgreSQL API

Field	Description
`apiVersion` **[Required]** string	`postgresql.k8s.enterprisedb.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.

Field	Description
`spec` **[Required]** *ClusterSpec*	Specification of the desired behavior of the cluster. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status
`status` *ClusterStatus*	Most recently observed status of the cluster. This data may not be up to date. Populated by the system. Read-only. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

ClusterImageCatalog

ClusterImageCatalog is the Schema for the clusterimagecatalogs API

Field	Description
`apiVersion` **[Required]** string	`postgresql.k8s.enterprisedb.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

Database

Database is the Schema for the databases API

Field	Description
`apiVersion` **[Required]** string	`postgresql.k8s.enterprisedb.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

ImageCatalog

ImageCatalog is the Schema for the imagecatalogs API

Field	Description
`apiVersion` **[Required]** string	`postgresql.k8s.enterprisedb.io/v1`
`kind` **[Required]** string	`ImageCatalog`
`metadata` **[Required]** *meta/v1.ObjectMeta*	No description provided.Refer to the Kubernetes API documentation for the fields of the `metadata` field.
`spec` **[Required]** *ImageCatalogSpec*	Specification of the desired behavior of the ImageCatalog. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

Pooler

Pooler is the Schema for the poolers API

Field	Description
`apiVersion` **[Required]** string	`postgresql.k8s.enterprisedb.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

Publication

Publication is the Schema for the publications API

Field	Description
`apiVersion` **[Required]** string	`postgresql.k8s.enterprisedb.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.

Field	Description
`spec` **[Required]** *PublicationSpec*	No description provided.
`status` **[Required]** *PublicationStatus*	No description provided.

ScheduledBackup

ScheduledBackup is the Schema for the scheduledbackups API

Field	Description
`apiVersion` **[Required]** string	`postgresql.k8s.enterprisedb.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

Subscription

Subscription is the Schema for the subscriptions API

Field	Description
`apiVersion` **[Required]** string	`postgresql.k8s.enterprisedb.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.

AffinityConfiguration

Appears in:

- ClusterSpec

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

Field	Description
`enablePodAntiAffinity` *bool*	Activates anti-affinity for the pods. The operator will define pods anti-affinity unless this field is explicitly set to false
`topologyKey` *string*	TopologyKey to use for anti-affinity configuration. See k8s documentation for more info on that
`nodeSelector` *map[string]string*	NodeSelector is map of key-value pairs used to define the nodes on which the pods can run. More info: https://kubernetes.io/docs/concepts/configuration/assign-pod-node/
`nodeAffinity` *core/v1.NodeAffinity*	NodeAffinity describes node affinity scheduling rules for the pod. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#node-affinity
`tolerations` *[]core/v1.Toleration*	Tolerations is a list of Tolerations that should be set for all the pods, in order to allow them to run on tainted nodes. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/taint-and-toleration/
`podAntiAffinityType` *string*	PodAntiAffinityType allows the user to decide whether pod anti-affinity between cluster instance has to be considered a strong requirement during scheduling or not. Allowed values are: "preferred" (default if empty) or "required". Setting it to "required", could lead to instances remaining pending until new kubernetes nodes are added if all the existing nodes don't match the required pod anti-affinity rule. More info: https://kubernetes.io/docs/concepts/scheduling-eviction/assign-pod-node/#inter-pod-affinity-and-anti-affinity
`additionalPodAntiAffinity` *core/v1.PodAntiAffinity*	AdditionalPodAntiAffinity allows to specify pod anti-affinity terms to be added to the ones generated by the operator if EnablePodAntiAffinity is set to true (default) or to be used exclusively if set to false.
`additionalPodAffinity` *core/v1.PodAffinity*	AdditionalPodAffinity allows to specify pod affinity terms to be passed to all the cluster's pods.

AvailableArchitecture

Appears in:

- ClusterStatus

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

BackupConfiguration

Appears in:

- 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/pkg/api.BarmanObjectStoreConfiguration*	The configuration for the barman-cloud tool suite
`retentionPolicy` *string*	RetentionPolicy is the retention policy to be used for backups and WALs (i.e. '60d'). The retention policy is expressed in the form of `XXu` where `XX` is a positive integer and `u` is in `[dwm]` - days, weeks, months. It's currently only applicable when using the BarmanObjectStore method.
`target` *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.

BackupMethod

(Alias of `string`)

Appears in:

- BackupSpec
- BackupStatus
- ScheduledBackupSpec

BackupMethod defines the way of executing the physical base backups of the selected PostgreSQL instance

BackupPhase

(Alias of `string`)

Appears in:

- BackupStatus

BackupPhase is the phase of the backup

BackupPluginConfiguration

Appears in:

- BackupSpec
- ScheduledBackupSpec

BackupPluginConfiguration contains the backup configuration used by the backup plugin

Field	Description
`name` **[Required]** *string*	Name is the name of the plugin managing this backup
`parameters` *map[string]string*	Parameters are the configuration parameters passed to the backup plugin for this backup

BackupSnapshotElementStatus

Appears in:

- 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

BackupSnapshotStatus

Appears in:

- BackupStatus

BackupSnapshotStatus the fields exclusive to the volumeSnapshot method backup

Field	Description
`elements` *[]BackupSnapshotElementStatus*	The elements list, populated with the gathered volume snapshots

BackupSource

Appears in:

- 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/pkg/api.LocalObjectReference*	(Members of `LocalObjectReference` are embedded into this type.) No description provided.
`endpointCA` *github.com/cloudnative-pg/machinery/pkg/api.SecretKeySelector*	EndpointCA store the CA bundle of the barman endpoint. Useful when using self-signed certificates to avoid errors with certificate issuer and barman-cloud-wal-archive.

BackupSpec

Appears in:

- Backup

BackupSpec defines the desired state of Backup

Field	Description
`cluster` **[Required]** *github.com/cloudnative-pg/machinery/pkg/api.LocalObjectReference*	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.

Field	Description
`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

BackupStatus

Appears in:

- Backup

BackupStatus defines the observed state of Backup

Field	Description
`BarmanCredentials` *github.com/cloudnative-pg/barman-cloud/pkg/api.BarmanCredentials*	(Members of `BarmanCredentials` are embedded into this type.) The potential credentials for each cloud provider
`endpointCA` *github.com/cloudnative-pg/machinery/pkg/api.SecretKeySelector*	EndpointCA store the CA bundle of the barman endpoint. Useful when using self-signed certificates to avoid errors with certificate issuer and barman-cloud-wal-archive.
`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

Field	Description
`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
`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`)

Field	Description
`pluginMetadata` *map[string]string*	A map containing the plugin metadata

BackupTarget

(Alias of `string`)

Appears in:

- BackupConfiguration
- BackupSpec
- ScheduledBackupSpec

BackupTarget describes the preferred targets for a backup

BootstrapConfiguration

Appears in:

- 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

BootstrapInitDB

Appears in:

- 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/machinery/pkg/api.LocalObjectReference*	Name of the secret containing the initial credentials for the owner of the user database. If empty a new secret will be created from scratch
`redwood` *bool*	If we need to enable/disable Redwood compatibility. Requires EPAS and for EPAS defaults to true
`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)

Field	Description
`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)
`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)

BootstrapPgBaseBackup

Appears in:

- 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` .

Field	Description
`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/machinery/pkg/api.LocalObjectReference*	Name of the secret containing the initial credentials for the owner of the user database. If empty a new secret will be created from scratch

BootstrapRecovery

Appears in:

- BootstrapConfiguration

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`.
`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 EDB Postgres for Kubernetes, 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 default, the recovery process applies all the available WAL files in the archive (full recovery). However, you can also end the recovery as soon as a consistent state is reached or recover to a point-in-time (PITR) by specifying a `RecoveryTarget` object, as expected by PostgreSQL (i.e., timestamp, transaction Id, LSN, ...). More info: https://www.postgresql.org/docs/current/runtime-config-wal.html#RUNTIME-CONFIG-WAL-RECOVERY-TARGET
`database` *string*	Name of the database used by the application. Default: `app`.
`owner` *string*	Name of the owner of the database in the instance to be used by applications. Defaults to the value of the `database` key.

Field	Description
`secret` *github.com/cloudnative-pg/machinery/pkg/api.LocalObjectReference*	Name of the secret containing the initial credentials for the owner of the user database. If empty a new secret will be created from scratch

CatalogImage

Appears in:

- ImageCatalogSpec

CatalogImage defines the image and major version

Field	Description
`image` **[Required]** *string*	The image reference
`major` **[Required]** *int*	The PostgreSQL major version of the image. Must be unique within the catalog.

CertificatesConfiguration

Appears in:

- CertificatesStatus
- ClusterSpec

CertificatesConfiguration contains the needed configurations to handle server certificates.

Field	Description
`serverCASecret` *string*	The secret containing the Server CA certificate. If not defined, a new secret will be created with a self-signed CA and will be used to generate the TLS certificate ServerTLSSecret. Contains: • `ca.crt` : CA that should be used to validate the server certificate, used as `sslrootcert` in client connection strings. • `ca.key` : key used to generate Server SSL certs, if ServerTLSSecret is provided, this can be omitted.
`serverTLSSecret` *string*	The secret of type kubernetes.io/tls containing the server TLS certificate and key that will be set as `ssl_cert_file` and `ssl_key_file` so that clients can connect to postgres securely. If not defined, ServerCASecret must provide also `ca.key` and a new secret will be created using the provided CA.

Field	Description
`replicationTLSSecret` *string*	The secret of type kubernetes.io/tls containing the client certificate to authenticate as the `streaming_replica` user. If not defined, ClientCASecret must provide also `ca.key`, and a new secret will be created using the provided CA.
`clientCASecret` *string*	The secret containing the Client CA certificate. If not defined, a new secret will be created with a self-signed CA and will be used to generate all the client certificates. Contains: • `ca.crt`: CA that should be used to validate the client certificates, used as `ssl_ca_file` of all the instances. • `ca.key`: key used to generate client certificates, if ReplicationTLSSecret is provided, this can be omitted.
`serverAltDNSNames` *[]string*	The list of the server alternative DNS names to be added to the generated server TLS certificates, when required.

CertificatesStatus

Appears in:

- ClusterStatus

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.

ClusterMonitoringTLSConfiguration

Appears in:

- MonitoringConfiguration

ClusterMonitoringTLSConfiguration is the type containing the TLS configuration for the cluster's monitoring

Field	Description

Field	Description
`enabled` *bool*	Enable TLS for the monitoring endpoint. Changing this option will force a rollout of all instances.

ClusterSpec

Appears in:

- Cluster

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/containers/images#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.

Field	Description
`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/pkg/api.LocalObjectReference*	The secret containing the superuser password. If not defined a new secret will be created with a randomly generated password
`enableSuperuserAccess` *bool*	When this option is enabled, the operator will use the `SuperuserSecret` to update the `postgres` user password (if the secret is not present, the operator will automatically create one). When this option is disabled, the operator will ignore the `SuperuserSecret` content, delete it when automatically created, and then blank the password of the `postgres` user by setting it to `NULL`. Disabled by default.
`certificates` *CertificatesConfiguration*	The configuration for the CA and related certificates
`imagePullSecrets` *[]github.com/cloudnative-pg/machinery/pkg/api.LocalObjectReference*	The list of pull secrets to be used to pull the images. If the license key contains a pull secret that secret will be automatically included.
`storage` *StorageConfiguration*	Configuration of the storage of the instances
`serviceAccountTemplate` *ServiceAccountTemplate*	Configure the generation of the service account
`walStorage` *StorageConfiguration*	Configuration of the storage for PostgreSQL WAL (Write-Ahead Log)
`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)

Field	Description
`smartStopDelay` *int32*	Deprecated: please use SmartShutdownTimeout instead
`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*	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*	Deployment strategy to follow to upgrade the primary server during a rolling update procedure, after all replicas have been successfully updated: it can be automated (`unsupervised` - default) or manual (`supervised`)
`primaryUpdateMethod` *PrimaryUpdateMethod*	Method to follow to upgrade the primary server during a rolling update procedure, after all replicas have been successfully updated: it can be with a switchover (`switchover`) or in-place (`restart` - default)
`backup` *BackupConfiguration*	The configuration to be used for backups

Field	Description
`nodeMaintenanceWindow` *NodeMaintenanceWindow*	Define a maintenance window for the Kubernetes nodes
`licenseKey` *string*	The license key of the cluster. When empty, the cluster operates in trial mode and after the expiry date (default 30 days) the operator will cease any reconciliation attempt. For details, please refer to the license agreement that comes with the operator.
`licenseKeySecret` *core/v1.SecretKeySelector*	The reference to the license key. When this is set it take precedence over LicenseKey.
`monitoring` *MonitoringConfiguration*	The configuration of the monitoring infrastructure of this cluster
`externalClusters` *[]ExternalCluster*	The list of external clusters which are used in the configuration
`logLevel` *string*	The instances' log level, one of the following values: error, warning, info (default), debug, trace
`projectedVolumeTemplate` *core/v1.ProjectedVolumeSource*	Template to be used to define projected volumes, projected volumes will be mounted under `/projected` base folder
`env` *[]core/v1.EnvVar*	Env follows the Env format to pass environment variables to the pods created in the cluster
`envFrom` *[]core/v1.EnvFromSource*	EnvFrom follows the EnvFrom format to pass environment variables sources to the pods to be used by Env
`managed` *ManagedConfiguration*	The configuration that is used by the portions of PostgreSQL that are managed by the instance manager
`seccompProfile` *core/v1.SeccompProfile*	The SeccompProfile applied to every Pod and Container. Defaults to: `RuntimeDefault`
`tablespaces` *[]TablespaceConfiguration*	The tablespaces configuration
`enablePDB` *bool*	Manage the `PodDisruptionBudget` resources within the cluster. When configured as `true` (default setting), the pod disruption budgets will safeguard the primary node from being terminated. Conversely, setting it to `false` will result in the absence of any `PodDisruptionBudget` resource, permitting the shutdown of all nodes hosting the PostgreSQL cluster. This latter configuration is advisable for any PostgreSQL cluster employed for development/staging purposes.
`plugins` *[]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.

ClusterStatus

Appears in:

- Cluster

ClusterStatus defines the observed state of Cluster

Field	Description
`instances` *int*	The total number of PVC Groups detected in the cluster. It may differ from the number of existing instance pods.
`readyInstances` *int*	The total number of ready instances in the cluster. It is equal to the number of ready instance pods.
`instancesStatus` *map[PodStatus][]string*	InstancesStatus indicates in which status the instances are
`instancesReportedState` *map[PodName]InstanceReportedState*	The reported state of the instances during the last reconciliation loop
`managedRolesStatus` *ManagedRoles*	ManagedRolesStatus reports the state of the managed roles in the cluster
`tablespacesStatus` *[]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

Field	Description
`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
`licenseStatus` *github.com/EnterpriseDB/cloud-native-postgres/pkg/licensekey.Status*	Status of the license
`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
`firstRecoverabilityPointByMethod` *map[BackupMethod]meta/v1.Time*	The first recoverability point, stored as a date in RFC3339 format, per backup method type
`lastSuccessfulBackup` *string*	Last successful backup, stored as a date in RFC3339 format This field is calculated from the content of LastSuccessfulBackupByMethod
`lastSuccessfulBackupByMethod` *map[BackupMethod]meta/v1.Time*	Last successful backup, stored as a date in RFC3339 format, per backup method type

Field	Description
`lastFailedBackup` *string*	Stored as a date in RFC3339 format
`cloudNativePostgresqlCommitHash` *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
`cloudNativePostgresqlOperatorHash` *string*	The hash of the binary of the operator
`availableArchitectures` *[]AvailableArchitecture*	AvailableArchitectures reports the available architectures of a cluster
`conditions` *[]meta/v1.Condition*	Conditions for cluster object
`instanceNames` *[]string*	List of instance names in the cluster
`onlineUpdateEnabled` *bool*	OnlineUpdateEnabled shows if the online upgrade is enabled inside the cluster
`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 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

ConfigMapResourceVersion

Appears in:

- 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

DataDurabilityLevel

(Alias of `string`)

Appears in:

- SynchronousReplicaConfiguration

DataDurabilityLevel specifies how strictly to enforce synchronous replication when cluster instances are unavailable. Options are `required` or `preferred`.

DataSource

Appears in:

- 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*	Configuration of the storage for PostgreSQL tablespaces

DatabaseObjectSpec

Appears in:

- ExtensionSpec
- SchemaSpec

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.

DatabaseObjectStatus

Appears in:

- 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

DatabaseReclaimPolicy

(Alias of `string`)

Appears in:

- DatabaseSpec

DatabaseReclaimPolicy describes a policy for end-of-life maintenance of databases.

DatabaseRoleRef

Appears in:

- TablespaceConfiguration

DatabaseRoleRef is a reference an a role available inside PostgreSQL

Field	Description
`name` *string*	No description provided.

DatabaseSpec

Appears in:

- 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.
`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.

Field	Description
`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*	The list of schemas to be managed in the database
`extensions` *[]ExtensionSpec*	The list of extensions to be managed in the database

DatabaseStatus

Appears in:

- 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*	Schemas is the status of the managed schemas
`extensions` *[]DatabaseObjectStatus*	Extensions is the status of the managed extensions

EPASConfiguration

Appears in:

- PostgresConfiguration

EPASConfiguration contains EDB Postgres Advanced Server specific configurations

Field	Description
`audit` *bool*	If true enables edb_audit logging
`tde` *TDEConfiguration*	TDE configuration

EmbeddedObjectMetadata

Appears in:

- 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.

EnsureOption

(Alias of `string`)

Appears in:

- DatabaseObjectSpec
- DatabaseSpec
- RoleConfiguration

EnsureOption represents whether we should enforce the presence or absence of a Role in a PostgreSQL instance

EphemeralVolumesSizeLimitConfiguration

Appears in:

- 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

ExtensionSpec

Appears in:

- DatabaseSpec

ExtensionSpec configures an extension in a database

Field	Description

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.

ExternalCluster

Appears in:

- 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, EDB Postgres for Kubernetes 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/pkg/api.BarmanObjectStoreConfiguration*	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

ImageCatalogRef

Appears in:

- ClusterSpec

ImageCatalogRef defines the reference to a major version in an ImageCatalog

Field	Description
`TypedLocalObjectReference` *core/v1.TypedLocalObjectReference*	(Members of `TypedLocalObjectReference` are embedded into this type.) No description provided.
`major` **[Required]** *int*	The major version of PostgreSQL we want to use from the ImageCatalog

ImageCatalogSpec

Appears in:

- ClusterImageCatalog
- ImageCatalog

ImageCatalogSpec defines the desired ImageCatalog

Field	Description
`images` **[Required]** *[]CatalogImage*	List of CatalogImages available in the catalog

ImageInfo

Appears in:

- 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

Import

Appears in:

- 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.

ImportSource

Appears in:

- Import

ImportSource describes the source for the logical snapshot

Field	Description
`externalCluster` **[Required]** *string*	The name of the externalCluster used for import

InstanceID

Appears in:

- BackupStatus

InstanceID contains the information to identify an instance

Field	Description
`podName` *string*	The pod name
`ContainerID` *string*	The container ID

InstanceReportedState

Appears in:

- 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

LDAPBindAsAuth

Appears in:

- LDAPConfig

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

Field	Description
`prefix` *string*	Prefix for the bind authentication option

Field	Description
`suffix` *string*	Suffix for the bind authentication option

LDAPBindSearchAuth

Appears in:

- LDAPConfig

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

Field	Description
`baseDN` *string*	Root DN to begin the user search
`bindDN` *string*	DN of the user to bind to the directory
`bindPassword` *core/v1.SecretKeySelector*	Secret with the password for the user to bind to the directory
`searchAttribute` *string*	Attribute to match against the username
`searchFilter` *string*	Search filter to use when doing the search+bind authentication

LDAPConfig

Appears in:

- PostgresConfiguration

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`

Field	Description
`bindAsAuth` *LDAPBindAsAuth*	Bind as authentication configuration
`bindSearchAuth` *LDAPBindSearchAuth*	Bind+Search authentication configuration
`tls` *bool*	Set to 'true' to enable LDAP over TLS. 'false' is default

LDAPScheme

(Alias of `string`)

Appears in:

- LDAPConfig

LDAPScheme defines the possible schemes for LDAP

ManagedConfiguration

Appears in:

- ClusterSpec

ManagedConfiguration represents the portions of PostgreSQL that are managed by the instance manager

Field	Description
`roles` *[]RoleConfiguration*	Database roles managed by the `Cluster`
`services` *ManagedServices*	Services roles managed by the `Cluster`

ManagedRoles

Appears in:

- ClusterStatus

ManagedRoles tracks the status of a cluster's managed roles

Field	Description
`byStatus` *map[RoleStatus][]string*	ByStatus gives the list of roles in each state
`cannotReconcile` *map[string][]string*	CannotReconcile lists roles that cannot be reconciled in PostgreSQL, with an explanation of the cause
`passwordStatus` *map[string]PasswordState*	PasswordStatus gives the last transaction id and password secret version for each managed role

ManagedService

Appears in:

- 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.

ManagedServices

Appears in:

- ManagedConfiguration

ManagedServices represents the services managed by the cluster.

Field	Description
`disabledDefaultServices` *[]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*	Additional is a list of additional managed services specified by the user.

Metadata

Appears in:

- PodTemplateSpec
- ServiceAccountTemplate
- ServiceTemplateSpec

Metadata is a structure similar to the metav1.ObjectMeta, but still parseable by controller-gen to create a suitable CRD for the user. The comment of PodTemplateSpec has an explanation of why we are not using the core data types.

Field	Description
`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

MonitoringConfiguration

Appears in:

- 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/api.ConfigMapKeySelector*	The list of config maps containing the custom queries
`customQueriesSecret` *[]github.com/cloudnative-pg/machinery/pkg/api.SecretKeySelector*	The list of secrets containing the custom queries
`enablePodMonitor` *bool*	Enable or disable the `PodMonitor`

Field	Description
`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/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/prometheus-operator/pkg/apis/monitoring/v1.RelabelConfig*	The list of relabelings for the `PodMonitor`. Applied to samples before scraping.

NodeMaintenanceWindow

Appears in:

- ClusterSpec

NodeMaintenanceWindow contains information that the operator will use while upgrading the underlying node.

This option is only useful when the chosen storage prevents the Pods from being freely moved across nodes.

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

OnlineConfiguration

Appears in:

- BackupSpec
- ScheduledBackupSpec
- VolumeSnapshotConfiguration

OnlineConfiguration contains the configuration parameters for the online volume snapshot

Field	Description

Field	Description
`waitForArchive` *bool*	If false, the function will return immediately after the backup is completed, without waiting for WAL to be archived. This behavior is only useful with backup software that independently monitors WAL archiving. Otherwise, WAL required to make the backup consistent might be missing and make the backup useless. By default, or when this parameter is true, pg_backup_stop will wait for WAL to be archived when archiving is enabled. On a standby, this means that it will wait only when archive_mode = always. If write activity on the primary is low, it may be useful to run pg_switch_wal on the primary in order to trigger an immediate segment switch.
`immediateCheckpoint` *bool*	Control whether the I/O workload for the backup initial checkpoint will be limited, according to the `checkpoint_completion_target` setting on the PostgreSQL server. If set to true, an immediate checkpoint will be used, meaning PostgreSQL will complete the checkpoint as soon as possible. `false` by default.

PasswordState

Appears in:

- ManagedRoles

PasswordState represents the state of the password of a managed RoleConfiguration

Field	Description
`transactionID` *int64*	the last transaction ID to affect the role definition in PostgreSQL
`resourceVersion` *string*	the resource version of the password secret

PgBouncerIntegrationStatus

Appears in:

- PoolerIntegrations

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

Field	Description
`secrets` *[]string*	No description provided.

PgBouncerPoolMode

(Alias of `string`)

Appears in:

- PgBouncerSpec

PgBouncerPoolMode is the mode of PgBouncer

PgBouncerSecrets

Appears in:

- PoolerSecrets

PgBouncerSecrets contains the versions of the secrets used by pgbouncer

Field	Description
`authQuery` *SecretVersion*	The auth query secret version

PgBouncerSpec

Appears in:

- PoolerSpec

PgBouncerSpec defines how to configure PgBouncer

Field	Description
`poolMode` *PgBouncerPoolMode*	The pool mode. Default: `session`.
`authQuerySecret` *github.com/cloudnative-pg/machinery/pkg/api.LocalObjectReference*	The credentials of the user that need to be used for the authentication query. In case it is specified, also an AuthQuery (e.g. "SELECT usename, passwd FROM pg_catalog.pg_shadow WHERE usename=$1") has to be specified and no automatic CNP 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 CNP Cluster integration will be triggered.
`parameters` *map[string]string*	Additional parameters to be passed to PgBouncer - please check the CNP documentation for a list of options you can configure

Field	Description
`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.

PluginConfiguration

Appears in:

- ClusterSpec
- ExternalCluster

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

PluginStatus

Appears in:

- 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

Field	Description
`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

PodTemplateSpec

Appears in:

- PoolerSpec

PodTemplateSpec is a structure allowing the user to set a template for Pod generation.

Unfortunately we can't use the corev1.PodTemplateSpec type because the generated CRD won't have the field for the metadata section.

References: https://github.com/kubernetes-sigs/controller-tools/issues/385 https://github.com/kubernetes-sigs/controller-tools/issues/448 https://github.com/prometheus-operator/prometheus-operator/issues/3041

Field	Description
`metadata` *Metadata*	Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata
`spec` *core/v1.PodSpec*	Specification of the desired behavior of the pod. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

PodTopologyLabels

(Alias of `map[string]string`)

Appears in:

- Topology

PodTopologyLabels represent the topology of a Pod. map[labelName]labelValue

PoolerIntegrations

Appears in:

- ClusterStatus

PoolerIntegrations encapsulates the needed integration for the poolers referencing the cluster

Field	Description
`pgBouncerIntegration` *PgBouncerIntegrationStatus*	No description provided.

PoolerMonitoringConfiguration

Appears in:

- 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/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/prometheus-operator/pkg/apis/monitoring/v1.RelabelConfig*	The list of relabelings for the `PodMonitor`. Applied to samples before scraping.

PoolerSecrets

Appears in:

- PoolerStatus

PoolerSecrets contains the versions of all the secrets used

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

PoolerSpec

Appears in:

- Pooler

PoolerSpec defines the desired state of Pooler

Field	Description
`cluster` **[Required]** *github.com/cloudnative-pg/machinery/pkg/api.LocalObjectReference*	This is the cluster reference on which the Pooler will work. Pooler name should never match with any cluster name within the same namespace.
`type` *PoolerType*	Type of service to forward traffic to. Default: `rw` .
`instances` *int32*	The number of replicas we want. Default: 1.
`template` *PodTemplateSpec*	The template of the Pod to be created
`pgbouncer` **[Required]** *PgBouncerSpec*	The PgBouncer configuration
`deploymentStrategy` *apps/v1.DeploymentStrategy*	The deployment strategy to use for pgbouncer to replace existing pods with new ones
`monitoring` *PoolerMonitoringConfiguration*	The configuration of the monitoring infrastructure of this pooler.
`serviceTemplate` *ServiceTemplateSpec*	Template for the Service to be created

PoolerStatus

Appears in:

- Pooler

PoolerStatus defines the observed state of Pooler

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

PoolerType

(Alias of `string`)

Appears in:

- PoolerSpec

PoolerType is the type of the connection pool, meaning the service we are targeting. Allowed values are `rw` and `ro`.

PostgresConfiguration

Appears in:

- 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
`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)

Field	Description
`epas` *EPASConfiguration*	EDB Postgres Advanced Server specific configurations
`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 EDB Postgres for Kubernetes Cluster. This should only be used for debugging and troubleshooting. Defaults to false.

PrimaryUpdateMethod

(Alias of `string`)

Appears in:

- ClusterSpec

PrimaryUpdateMethod contains the method to use when upgrading the primary server of the cluster as part of rolling updates

PrimaryUpdateStrategy

(Alias of `string`)

Appears in:

- ClusterSpec

PrimaryUpdateStrategy contains the strategy to follow when upgrading the primary server of the cluster as part of rolling updates

Probe

Appears in:

- ProbeWithStrategy
- ProbesConfiguration

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.

ProbeStrategyType

(Alias of `string`)

Appears in:

- ProbeWithStrategy

ProbeStrategyType is the type of the strategy used to declare a PostgreSQL instance ready

ProbeWithStrategy

Appears in:

- 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

ProbesConfiguration

Appears in:

- 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]** *Probe*	The liveness probe configuration
`readiness` **[Required]** *ProbeWithStrategy*	The readiness probe configuration

PublicationReclaimPolicy

(Alias of `string`)

Appears in:

- PublicationSpec

PublicationReclaimPolicy defines a policy for end-of-life maintenance of Publications.

PublicationSpec

Appears in:

- 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

PublicationStatus

Appears in:

- 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

PublicationTarget

Appears in:

- 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*	Just the following schema objects

PublicationTargetObject

Appears in:

- 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.

PublicationTargetTable

Appears in:

- 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
`columns` *[]string*	The columns to publish

RecoveryTarget

Appears in:

- BootstrapRecovery

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

Field	Description
`backupID` *string*	The ID of the backup from which to start the recovery process. If empty (default) the operator will automatically detect the backup based on targetTime or targetLSN if specified. Otherwise use the latest available backup in chronological order.
`targetTLI` *string*	The target timeline ("latest" or a positive integer)
`targetXID` *string*	The target transaction ID
`targetName` *string*	The target name (to be previously created with `pg_create_restore_point`)
`targetLSN` *string*	The target LSN (Log Sequence Number)
`targetTime` *string*	The target time as a timestamp in the RFC3339 standard
`targetImmediate` *bool*	End recovery as soon as a consistent state is reached
`exclusive` *bool*	Set the target to be exclusive. If omitted, defaults to false, so that in Postgres, `recovery_target_inclusive` will be true

ReplicaClusterConfiguration

Appears in:

- ClusterSpec

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`

Field	Description
`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.
`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.

ReplicationSlotsConfiguration

Appears in:

- ClusterSpec

ReplicationSlotsConfiguration encapsulates the configuration of replication slots

Field	Description
`highAvailability` *ReplicationSlotsHAConfiguration*	Replication slots for high availability configuration
`updateInterval` *int*	Standby will update the status of the local replication slots every `updateInterval` seconds (default 30).
`synchronizeReplicas` *SynchronizeReplicasConfiguration*	Configures the synchronization of the user defined physical replication slots

ReplicationSlotsHAConfiguration

Appears in:

- ReplicationSlotsConfiguration

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 `_cnp_`.

RoleConfiguration

Appears in:

- 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/pkg/api.LocalObjectReference*	Secret containing the password of the role (if present) If null, the password will be ignored unless DisablePassword is set
`connectionLimit` *int64*	If the role can log in, this specifies how many concurrent connections the role can make. `-1` (the default) means no limit.
`validUntil` *meta/v1.Time*	Date and time after which the role's password is no longer valid. When omitted, the password will never expire (default).
`inRoles` *[]string*	List of one or more existing roles to which this role will be immediately added as a new member. Default empty.
`inherit` *bool*	Whether a role "inherits" the privileges of roles it is a member of. Defaults is `true`.

Field	Description
`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` .
`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` .

SQLRefs

Appears in:

- 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/api.SecretKeySelector*	SecretRefs holds a list of references to Secrets
`configMapRefs` *[]github.com/cloudnative-pg/machinery/pkg/api.ConfigMapKeySelector*	ConfigMapRefs holds a list of references to ConfigMaps

ScheduledBackupSpec

Appears in:

- 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/pkg/api.LocalObjectReference*	The cluster to backup
`backupOwnerReference` *string*	Indicates which ownerReference should be put inside the created backup resources. • none: no owner reference for created backup objects (same behavior as before the field was introduced) • self: sets the Scheduled backup object as owner of the backup • cluster: set the cluster as owner of the backup
`target` *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

ScheduledBackupStatus

Appears in:

- 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

SchemaSpec

Appears in:

- DatabaseSpec

SchemaSpec configures a schema in a database

Field	Description
`DatabaseObjectSpec` *DatabaseObjectSpec*	(Members of `DatabaseObjectSpec` are embedded into this type.) Common fields
`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`.

SecretVersion

Appears in:

- PgBouncerSecrets
- PoolerSecrets

SecretVersion contains a secret name and its ResourceVersion

Field	Description
`name` *string*	The name of the secret
`version` *string*	The ResourceVersion of the secret

SecretsResourceVersion

Appears in:

- ClusterStatus

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

ServiceAccountTemplate

Appears in:

- 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

ServiceSelectorType

(Alias of `string`)

Appears in:

- ManagedService
- ManagedServices

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

ServiceTemplateSpec

Appears in:

- ManagedService
- PoolerSpec

ServiceTemplateSpec is a structure allowing the user to set a template for Service generation.

Field	Description
`metadata` *Metadata*	Standard object's metadata. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#metadata
`spec` *core/v1.ServiceSpec*	Specification of the desired behavior of the service. More info: https://git.k8s.io/community/contributors/devel/sig-architecture/api-conventions.md#spec-and-status

ServiceUpdateStrategy

(Alias of `string`)

Appears in:

- ManagedService

ServiceUpdateStrategy describes how the changes to the managed service should be handled

SnapshotOwnerReference

(Alias of `string`)

Appears in:

- VolumeSnapshotConfiguration

SnapshotOwnerReference defines the reference type for the owner of the snapshot. This specifies which owner the processed resources should relate to.

SnapshotType

(Alias of `string`)

Appears in:

- Import

SnapshotType is a type of allowed import

StorageConfiguration

Appears in:

- ClusterSpec
- TablespaceConfiguration

StorageConfiguration is the configuration used to create and reconcile PVCs, usable for WAL volumes, PGDATA volumes, or tablespaces

Field	Description
`storageClass` *string*	StorageClass to use for PVCs. Applied after evaluating the PVC template, if available. If not specified, the generated PVCs will use the default storage class
`size` *string*	Size of the storage. Required if not already specified in the PVC template. Changes to this field are automatically reapplied to the created PVCs. Size cannot be decreased.

Field	Description
`resizeInUseVolumes` *bool*	Resize existent PVCs, defaults to true
`pvcTemplate` *core/v1.PersistentVolumeClaimSpec*	Template to be used to generate the Persistent Volume Claim

SubscriptionReclaimPolicy

(Alias of `string`)

Appears in:

- SubscriptionSpec

SubscriptionReclaimPolicy describes a policy for end-of-life maintenance of Subscriptions.

SubscriptionSpec

Appears in:

- 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
`dbname` **[Required]** *string*	The name of the database where the publication will be installed in the "subscriber" cluster
`parameters` *map[string]string*	Subscription parameters part of the `WITH` clause as expected by PostgreSQL `CREATE SUBSCRIPTION` command
`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")

Field	Description
`subscriptionReclaimPolicy` *SubscriptionReclaimPolicy*	The policy for end-of-life maintenance of this subscription

SubscriptionStatus

Appears in:

- 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

SwitchReplicaClusterStatus

Appears in:

- ClusterStatus

SwitchReplicaClusterStatus contains all the statuses regarding the switch of a cluster to a replica cluster

Field	Description
`inProgress` *bool*	InProgress indicates if there is an ongoing procedure of switching a cluster to a replica cluster.

SyncReplicaElectionConstraints

Appears in:

- PostgresConfiguration

SyncReplicaElectionConstraints contains the constraints for sync replicas election.

For anti-affinity parameters two instances are considered in the same location if all the labels values match.

In future synchronous replica election restriction by name will be supported.

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

SynchronizeReplicasConfiguration

Appears in:

- ReplicationSlotsConfiguration

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)

SynchronousReplicaConfiguration

Appears in:

- 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]** *SynchronousReplicaConfigurationMethod*	Method to select synchronous replication standbys from the listed servers, accepting 'any' (quorum-based synchronous replication) or 'first' (priority-based synchronous replication) 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.

Field	Description
`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).
`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).

SynchronousReplicaConfigurationMethod

(Alias of `string`)

Appears in:

- SynchronousReplicaConfiguration

SynchronousReplicaConfigurationMethod configures whether to use quorum based replication or a priority list

TDEConfiguration

Appears in:

- EPASConfiguration

TDEConfiguration contains the Transparent Data Encryption configuration

Field	Description
`enabled` *bool*	True if we want to have TDE enabled
`secretKeyRef` *core/v1.SecretKeySelector*	Reference to the secret that contains the encryption key
`wrapCommand` *core/v1.SecretKeySelector*	WrapCommand is the encrypt command provided by the user
`unwrapCommand` *core/v1.SecretKeySelector*	UnwrapCommand is the decryption command provided by the user

Field	Description
`passphraseCommand` *core/v1.SecretKeySelector*	PassphraseCommand is the command executed to get the passphrase that will be passed to the OpenSSL command to encrypt and decrypt

TablespaceConfiguration

Appears in:

- 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.

TablespaceState

Appears in:

- ClusterStatus

TablespaceState represents the state of a tablespace in a cluster

Field	Description
`name` **[Required]** *string*	Name is the name of the tablespace
`owner` *string*	Owner is the PostgreSQL user owning the tablespace
`state` **[Required]** *TablespaceStatus*	State is the latest reconciliation state

Field	Description
`error` *string*	Error is the reconciliation error, if any

TablespaceStatus

(Alias of `string`)

Appears in:

- TablespaceState

TablespaceStatus represents the status of a tablespace in the cluster

Topology

Appears in:

- ClusterStatus

Topology contains the cluster topology

Field	Description
`instances` *map[PodName]PodTopologyLabels*	Instances contains the pod topology of the instances
`nodesUsed` *int32*	NodesUsed represents the count of distinct nodes accommodating the instances. A value of '1' suggests that all instances are hosted on a single node, implying the absence of High Availability (HA). Ideally, this value should be the same as the number of instances in the Postgres HA cluster, implying shared nothing architecture on the compute side.
`successfullyExtracted` *bool*	SuccessfullyExtracted indicates if the topology data was extract. It is useful to enact fallback behaviors in synchronous replica election in case of failures

VolumeSnapshotConfiguration

Appears in:

- BackupConfiguration

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

56 Backup and Recovery

Backup and recovery are in two separate sections.

57 Appendix C - Common object stores for backups

> **Warning**
>
> As of EDB Postgres for Kubernetes 1.26, **native Barman Cloud support is deprecated** in favor of the **Barman Cloud Plugin**. While the native integration remains functional for now, we strongly recommend beginning a gradual migration to the plugin-based interface after appropriate testing. The Barman Cloud Plugin documentation describes how to use common object stores.

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

- Amazon S3
- Microsoft Azure Blob Storage
- Google Cloud Storage

You can also use any compatible implementation of the supported services.

The required setup depends on the chosen storage provider and is discussed in the following sections.

AWS S3

AWS Simple Storage Service (S3) is a very popular object storage service offered by Amazon.

As far as EDB Postgres for Kubernetes backup is concerned, you can define the permissions to store backups in S3 buckets in two ways:

- If EDB Postgres for Kubernetes is running in EKS. you may want to use theIRSA authentication method
- Alternatively, you can use the `ACCESS_KEY_ID` and `ACCESS_SECRET_KEY` credentials

AWS Access key

You will need the following information about your environment:

- `ACCESS_KEY_ID` : the ID of the access key that will be used to upload files into S3

- `ACCESS_SECRET_KEY` : the secret part of the access key mentioned above

- `ACCESS_SESSION_TOKEN` : the optional session token, in case it is required

The access key used must have permission to upload files into the bucket. Given that, you must create a Kubernetes secret with the credentials, and you can do that with the following command:

```
kubectl create secret generic aws-creds
\
  --from-literal=ACCESS_KEY_ID=<access key here>
\
  --from-literal=ACCESS_SECRET_KEY=<secret key
here>
# --from-literal=ACCESS_SESSION_TOKEN=<session token here> # if
required
```

The credentials will be stored inside Kubernetes and will be encrypted if encryption at rest is configured in your installation.

Once that secret has been created, you can configure your cluster like in the following example:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path
here>"
      s3Credentials:
        accessKeyId:
          name: aws-creds
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: aws-creds
          key: ACCESS_SECRET_KEY
```

The destination path can be any URL pointing to a folder where the instance can upload the WAL files, e.g. `s3://BUCKET_NAME/path/to/folder`.

IAM Role for Service Account (IRSA)

In order to use IRSA you need to set an `annotation` in the `ServiceAccount` of the Postgres cluster.

We can configure EDB Postgres for Kubernetes to inject them using the `serviceAccountTemplate` stanza:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
[...]
spec:
  serviceAccountTemplate:
    metadata:
      annotations:
        eks.amazonaws.com/role-arn: arn:[...]
        [...]
```

S3 lifecycle policy

Barman Cloud writes objects to S3, then does not update them until they are deleted by the Barman Cloud retention policy. A recommended approach for an S3 lifecycle policy is to expire the current version of objects a few days longer than the Barman retention policy, enable object versioning, and expire non-current versions after a number of days. Such a policy protects against accidental deletion, and also allows for restricting permissions to the EDB Postgres for Kubernetes workload so that it may delete objects from S3 without granting permissions to permanently delete objects.

Other S3-compatible Object Storages providers

In case you're using S3-compatible object storage, like **MinIO** or **Linode Object Storage**, you can specify an endpoint instead of using the default S3 one.

In this example, it will use the `bucket` of **Linode** in the region `us-east1`.

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "s3://bucket/"
      endpointURL: "https://us-east1.linodeobjects.com"
      s3Credentials:
        [...]
```

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

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "s3://[your-bucket-name]/[your-backup-folder]/"
      endpointURL: "https://sfo3.digitaloceanspaces.com"
      s3Credentials:
        [...]
```

Using Object Storage with a private CA

Suppose you configure an Object Storage provider which uses a certificate signed with a private CA, for example when using OpenShift or MinIO via HTTPS. In that case, you need to set the option `endpointCA` inside `barmanObjectStore` referring to a secret containing the CA bundle, so that Barman can verify the certificate correctly. You can find instructions on creating a secret using your cert files in the certificates document. Once you have created the secret, you can populate the `endpointCA` as in the following example:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  [...]
  backup:
    barmanObjectStore:
      endpointURL: <myEndpointURL>
      endpointCA:
        name: my-ca-
secret
        key:
ca.crt
```

> **Note**
>
> If you want ConfigMaps and Secrets to be **automatically** reloaded by instances, you can add a label with key `k8s.enterprisedb.io/reload` to the Secrets/ConfigMaps. Otherwise, you will have to reload the instances using the `kubectl cnp reload` subcommand.

Azure Blob Storage

Azure Blob Storage is the object storage service provided by Microsoft.

In order to access your storage account for backup and recovery of EDB Postgres for Kubernetes managed databases, you will need one of the following combinations of credentials:

- Connection String
- Storage account name and Storage account access key
- Storage account name and Storage account SAS Token
- Storage account name and Azure AD Workload Identity properly configured.

Using **Azure AD Workload Identity**, you can avoid saving the credentials into a Kubernetes Secret, and have a Cluster configuration adding the `inheritFromAzureAD` as follows:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path
here>"
      azureCredentials:
        inheritFromAzureAD: true
```

On the other side, using both **Storage account access key** or **Storage account SAS Token**, the credentials need to be stored inside a Kubernetes Secret, adding data entries only when needed. The following command performs that:

```
kubectl create secret generic azure-creds
\
  --from-literal=AZURE_STORAGE_ACCOUNT=<storage account name> \
  --from-literal=AZURE_STORAGE_KEY=<storage account key>
\
  --from-literal=AZURE_STORAGE_SAS_TOKEN=<SAS token> \
  --from-literal=AZURE_STORAGE_CONNECTION_STRING=<connection
string>
```

The credentials will be encrypted at rest, if this feature is enabled in the used Kubernetes cluster.

Given the previous secret, the provided credentials can be injected inside the cluster configuration:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "<destination path
here>"
      azureCredentials:
        connectionString:
          name: azure-
creds
          key: AZURE_CONNECTION_STRING
        storageAccount:
          name: azure-
creds
          key:
AZURE_STORAGE_ACCOUNT
        storageKey:
          name: azure-
creds
          key: AZURE_STORAGE_KEY
        storageSasToken:
          name: azure-
creds
          key: AZURE_STORAGE_SAS_TOKEN
```

When using the Azure Blob Storage, the `destinationPath` fulfills the following structure:

```
<http|https>://<account-name>.<service-name>.core.windows.net/<resource-path>
```

where `<resource-path>` is `<container>/<blob>`. The **account name**, which is also called **storage account name**, is included in the used host name.

Other Azure Blob Storage compatible providers

If you are using a different implementation of the Azure Blob Storage APIs, the `destinationPath` will have the following structure:

```
<http|https>://<local-machine-address>:<port>/<account-name>/<resource-path>
```

In that case, `<account-name>` is the first component of the path.

This is required if you are testing the Azure support via the Azure Storage Emulator orAzurite.

Google Cloud Storage

Currently, the EDB Postgres for Kubernetes operator supports two authentication methods for Google Cloud Storage:

- the first one assumes that the pod is running inside a Google Kubernetes Engine cluster
- the second one leverages the environment variable `GOOGLE_APPLICATION_CREDENTIALS`

Running inside Google Kubernetes Engine

When running inside Google Kubernetes Engine you can configure your backups to simply rely on Workload Identity, without having to set any credentials. In particular, you need to:

- set `.spec.backup.barmanObjectStore.googleCredentials.gkeEnvironment` to `true`
- set the `iam.gke.io/gcp-service-account` annotation in the `serviceAccountTemplate` stanza

Please use the following example as a reference:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  [...]
  backup:
    barmanObjectStore:
      destinationPath: "gs://<destination path
here>"
      googleCredentials:
        gkeEnvironment: true

  serviceAccountTemplate:
    metadata:
      annotations:
        iam.gke.io/gcp-service-account:  [...].iam.gserviceaccount.com
        [...]
```

Using authentication

Following the instruction from Google you will get a JSON file that contains all the required information to authenticate.

The content of the JSON file must be provided using a `Secret` that can be created with the following command:

```
kubectl create secret generic backup-creds --from-file=gcsCredentials=gcs_credentials_file.json
```

This will create the `Secret` with the name `backup-creds` to be used in the yaml file like this:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
[...]
spec:
  backup:
    barmanObjectStore:
      destinationPath: "gs://<destination path
here>"
      googleCredentials:
        applicationCredentials:
          name: backup-creds
          key:
gcsCredentials
```

Now the operator will use the credentials to authenticate against Google Cloud Storage.

> **Important**
>
> This way of authentication will create a JSON file inside the container with all the needed information to access your Google Cloud Storage bucket, meaning that if someone gets access to the pod will also have write permissions to the bucket.

58 External Secrets

External Secrets is a CNCF Sandbox project, accepted in 2022 under the sponsorship of TAG Security.

About

The **External Secrets Operator (ESO)** is a Kubernetes operator that enhances secret management by decoupling the storage of secrets from Kubernetes itself. It enables seamless synchronization between external secret management systems and native Kubernetes `Secret` resources.

ESO supports a wide range of backends, including:

- HashiCorp Vault
- AWS Secrets Manager
- Google Secret Manager
- Azure Key Vault
- IBM Cloud Secrets Manager

...and many more. For a full and up-to-date list of supported providers, refer to the official External Secrets documentation.

Integration with PostgreSQL and EDB Postgres for Kubernetes

When it comes to PostgreSQL databases, External Secrets integrates seamlessly with EDB Postgres for Kubernetes in two major use cases:

- **Automated password management:** ESO can handle the automatic generation and rotation of database user passwords stored in Kubernetes `Secret` resources, ensuring that applications running inside the cluster always have access to up-to-date credentials.

- **Cross-platform secret access:** It enables transparent synchronization of those passwords with an external Key Management Service (KMS) via a `SecretStore` resources. This allows applications and developers outside the Kubernetes cluster—who may not have access to Kubernetes secrets—to retrieve the database credentials directly from the external KMS.

Example: Automated Password Management with External Secrets

Let's walk through how to automatically rotate the password of the `app` user every 24 hours in the `cluster-example` Postgres cluster from the quickstart guide.

> **Important**
>
> Before proceeding, ensure that the `cluster-example` Postgres cluster is up and running in your environment.

By default, EDB Postgres for Kubernetes generates and manages a Kubernetes `Secret` named `cluster-example-app`, which contains the credentials for the `app` user in the `cluster-example` cluster. You can read more about this in the "Connecting from an application" section.

With External Secrets, the goal is to:

1. Define a `Password` generator that specifies how to generate the password.
2. Create an `ExternalSecret` resource that keeps the `cluster-example-app` secret in sync by updating only the `password` and `pgpass` fields.

Creating the Password Generator

The following example creates a `Password` generator resource named `pg-password-generator` in the `default` namespace. You can customize the name and properties to suit your needs:

```yaml
apiVersion: generators.external-secrets.io/v1alpha1
kind: Password
metadata:
  name: pg-password-generator
spec:
  length: 42
  digits: 5
  symbols: 5
  symbolCharacters: "-_$@"
  noUpper: false
  allowRepeat: true
```

This specification defines the characteristics of the generated password, including its length and the inclusion of digits, symbols, and uppercase letters.

Creating the External Secret

The example below creates an `ExternalSecret` resource named `cluster-example-app-secret`, which refreshes the password every 24 hours. It uses a `Merge` policy to update only the specified fields (`password`, `pgpass`, `jdbc-uri` and `uri`) in the `cluster-example-app` secret.

```yaml
apiVersion: external-secrets.io/v1beta1
kind: ExternalSecret
metadata:
  name: cluster-example-app-secret
spec:
  refreshInterval: "24h"
  target:
    name: cluster-example-app
    creationPolicy: Merge
    template:
      metadata:
        labels:
          k8s.enterprisedb.io/reload: "true"
      data:
        password: "{{ .password }}"
        pgpass: "cluster-example-rw:5432:app:app:{{ .password }}"
        jdbc-uri: "jdbc:postgresql://cluster-example-rw.default:5432/app?password={{ .password }}&user=app"
        uri: "postgresql://app:{{ .password }}@cluster-example-rw.default:5432/app"
  dataFrom:
    - sourceRef:
        generatorRef:
          apiVersion: generators.external-secrets.io/v1alpha1
          kind: Password
          name: pg-password-generator
```

The label `k8s.enterprisedb.io/reload: "true"` ensures that EDB Postgres for Kubernetes triggers a reload of the user password in the database when the secret changes.

Verifying the Configuration

To check that the `ExternalSecret` is correctly synchronizing:

```
kubectl get es cluster-example-app-
secret
```

To observe the password being refreshed in real time, temporarily reduce the `refreshInterval` to `30s` and run the following command repeatedly:

```
kubectl get secret cluster-example-app
\
  -o jsonpath="{.data.password}" | base64 -d
```

You should see the password change every 30 seconds, confirming that the rotation is working correctly.

There's More

While the example above focuses on the default `cluster-example-app` secret created by EDB Postgres for Kubernetes, the same approach can be extended to manage any custom secrets or PostgreSQL users you create to regularly rotate their password.

Example: Integration with an External KMS

One of the most widely used Key Management Service (KMS) providers in the CNCF ecosystem is HashiCorp Vault. Although Vault is licensed under the Business Source License (BUSL), a fully compatible and actively maintained open source alternative is available: OpenBao. OpenBao supports all the same interfaces as HashiCorp Vault, making it a true drop-in replacement.

In this example, we'll demonstrate how to integrate EDB Postgres for Kubernetes, External Secrets Operator, and HashiCorp Vault to automatically rotate a PostgreSQL password and securely store it in Vault.

> **Important**
>
> This example assumes that HashiCorp Vault is already installed and properly configured in your environment, and that your team has the necessary expertise to operate it. There are various ways to deploy Vault, and detailing them is outside the scope of EDB Postgres for Kubernetes. While it's possible to run Vault inside Kubernetes, it is more commonly deployed externally. For detailed instructions, consult the HashiCorp Vault documentation.

Continuing from the previous example, we will now create the necessary `SecretStore` and `PushSecret` resources to complete the integration with Vault.

Creating the `SecretStore`

In this example, we assume that HashiCorp Vault is accessible from within the namespace at `http://vault.vault.svc:8200`, and that a Kubernetes `Secret` named `vault-token` exists in the same namespace, containing the token used to authenticate with Vault.

```yaml
apiVersion: external-secrets.io/v1beta1
kind:
SecretStore
metadata:
  name: vault-
backend
spec:
  provider:
    vault:
      server: "http://vault.vault.svc:8200"
      path: "secrets"
      # Specifies the Vault KV secret engine version ("v1" or
"v2").
      # Defaults to "v2" if not
set.
      version: "v2"
      auth:
        # References a Kubernetes Secret that contains the Vault
token.
        # See:
https://www.vaultproject.io/docs/auth/token
        tokenSecretRef:
          name: "vault-token"
          key: "token"
---
apiVersion: v1
kind:
Secret
metadata:
  name: vault-
token
data:
  token: aHZzLioqKioqKio= #
hvs.*******
```

This configuration creates a `SecretStore` resource named `vault-backend`.

> **Important**
>
> This example uses basic token-based authentication, which is suitable for testing API, and CLI use cases. While it is the default method enabled in Vault, it is not recommended for production environments. For production, consider using more secure authentication methods. Refer to the External Secrets Operator documentation for a full list of supported authentication mechanisms.

> **Info**
>
> HashiCorp Vault must have a KV secrets engine enabled at the `secrets` path with version `v2`. If your Vault instance uses a different path or version, be sure to update the `path` and `version` fields accordingly.

Creating the `PushSecret`

The `PushSecret` resource is used to push a Kubernetes `Secret` to HashiCorp Vault. In this simplified example, we'll push the credentials for the `app` user of the sample cluster `cluster-example`.

For more details on configuring `PushSecret`, refer to the External Secrets Operator documentation.

```yaml
apiVersion: external-secrets.io/v1alpha1
kind: PushSecret
metadata:
  name: pushsecret-
example
spec:
  deletionPolicy:
Delete
  refreshInterval:
24h
  secretStoreRefs:
    - name: vault-
backend
      kind:
SecretStore
  selector:
    secret:
      name: cluster-example-
app
  data:
    - match:
        remoteRef:
          remoteKey: cluster-example-
app
```

In this example, the `PushSecret` resource instructs the External Secrets Operator to push the Kubernetes `Secret` named `cluster-example-app` to HashiCorp Vault (from the previous example). The `remoteKey` defines the name under which the secret will be stored in Vault, using the `SecretStore` named `vault-backend`.

Verifying the Configuration

To verify that the `PushSecret` is functioning correctly, navigate to the HashiCorp Vault UI. In the `kv` secrets engine at the path `secrets`, you should find a secret named `cluster-example-app`, corresponding to the `remoteKey` defined above.

60 PostgreSQL Database Management

EDB Postgres for Kubernetes 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" section.

For more advanced use cases, EDB Postgres for Kubernetes 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.

Key Concepts

Scope of Management

> **Important**
>
> EDB Postgres for Kubernetes 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.

Declarative `Database` Manifest

The following example demonstrates how a `Database` resource interacts with a `Cluster`:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind:
Database
metadata:
  name: cluster-example-
one
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.

> **Info**
>
> Please refer to the API reference the full list of attributes you can define for each `Database` object.

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.

> **Info**
>
> The distinction between `metadata.name` and `spec.name` allows multiple `Database` resources to reference databases with the same name across different EDB Postgres for Kubernetes clusters in the same Kubernetes namespace.

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 EDB Postgres for Kubernetes.

> **Important**
>
> Creating a `Database` with `spec.name` set to `postgres` , `template0` , or `template1` is not allowed.

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:

```yaml
apiVersion: postgresql.k8s.enterprisedb.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.

Deleting a Database

EDB Postgres for Kubernetes supports two methods for database deletion:

1. Using the `delete` reclaim policy
2. Declaratively setting the database's `ensure` field to `absent`

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.k8s.enterprisedb.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.

Declaratively Setting `ensure: absent`

To remove a database, set the `ensure` field to `absent` like in the following example:.

```
apiVersion: postgresql.k8s.enterprisedb.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.

Managing Extensions in a Database

> **Info**
>
> While extensions are database-scoped rather than global objects, EDB Postgres for Kubernetes provides a declarative interface for managing them. This approach is necessary because installing certain extensions may require superuser privileges, which EDB Postgres for Kubernetes recommends disabling by default. By leveraging this API, users can efficiently manage extensions in a scalable and controlled manner without requiring elevated privileges.

EDB Postgres for Kubernetes 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:

```yaml
# ...
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.

> **Info**
>
> EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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.

Managing Schemas in a Database

> **Info**
>
> Schema management in PostgreSQL is an exception to EDB Postgres for Kubernetes' primary focus on managing global objects. Since schemas exist within a database, they are typically managed as part of the application development process. However, EDB Postgres for Kubernetes provides a declarative interface for schema management, primarily to complete the support of extensions deployment within schemas.

EDB Postgres for Kubernetes 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.

> **Info**
>
> EDB Postgres for Kubernetes manages schemas using the following PostgreSQL's SQL commands: `CREATE SCHEMA` , `DROP SCHEMA` , `ALTER SCHEMA` .

Limitations and Caveats

Renaming a database

While EDB Postgres for Kubernetes 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.

Creating vs. Altering a Database

- For new databases, EDB Postgres for Kubernetes 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.

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.

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"'
```

Postgres Version Differences

EDB Postgres for Kubernetes 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`.

Manual Changes

EDB Postgres for Kubernetes 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.

61 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`:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: ImageCatalog
metadata:
  name: postgresql
  namespace: default
spec:
  images:
    - major: 15
      image: quay.io/enterprisedb/postgresql:15.6
    - major: 16
      image: quay.io/enterprisedb/postgresql:16.8
    - major: 17
      image: quay.io/enterprisedb/postgresql:17.5
```

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

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind:
ClusterImageCatalog
metadata:
  name: postgresql
spec:
  images:
    - major: 15
      image: quay.io/enterprisedb/postgresql:15.6
    - major: 16
      image: quay.io/enterprisedb/postgresql:16.8
    - major: 17
      image: quay.io/enterprisedb/postgresql:17.5
```

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

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  instances: 3
  imageCatalogRef:
    apiGroup: postgresql.k8s.enterprisedb.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.

EDB Postgres for Kubernetes Catalogs

The EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes project's GitHub repositories, cluster administrators can ensure that their clusters are automatically updated to the latest version within the specified major release.

PostgreSQL Container Images

You can install the latest version of the cluster catalog for the PostgreSQL Container Images (cloudnative-pg/postgres-containers repository) with:

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

PostGIS Container Images

You can install the latest version of the cluster catalog for the PostGIS Container Images (cloudnative-pg/postgis-containers repository) with:

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

62 Iron Bank

EDB Postgres for Kubernetes(PG4K) is available on Iron Bank. As you can read in theoverview page:

> Iron Bank is the DoD's source for hardened containers.
>
> [... snipped ...]
>
> Iron Bank ultimately is for anyone to consume or contribute. However, we specifically target the following personas:
>
> - DoD organizations wishing to consume hardened containers and Iron Banks BoE (Body of Evidence) for each container
> - DoD organizations wishing to help contribute to containers (e.g. bug fixes, new applications, updates)
> - DoD Authorization Officials wishing to understand the risks associated with applications
> - Commercial vendors wishing to bring their application to the DoD

Iron Bank is a part of DoD's Platform One.

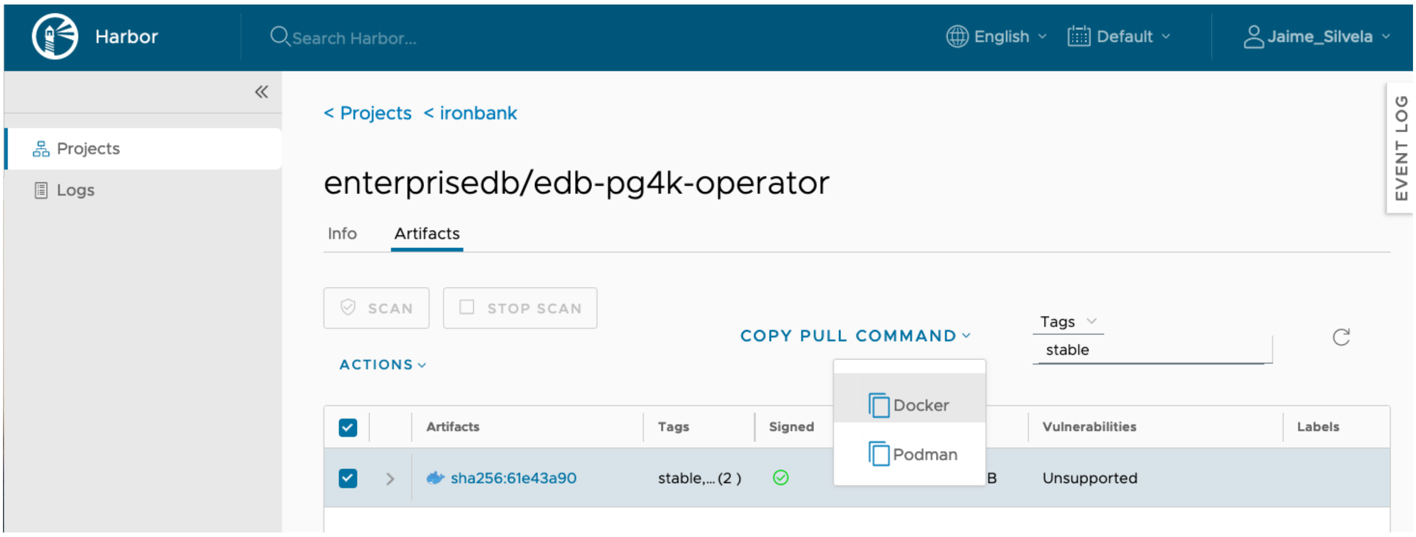
You will need your Iron Bank credentials to access the Iron Bank page for EDB Postgres for Kubernetes.

Pulling the EDB PG4K and operand images from Iron Bank

The images are pulled from the separate Iron Bank container registry. To be able to pull images from the Iron Bank registry, please follow theinstructions from Iron Bank.

Specifically, you will need to use your registry1 credentials to pull images.

To find the desired operator or operand images, we recommend to use the search tool to look with the string `enterprisedb` , and filter by `Tags` , looking for `stable` , as shown in the image. From there, you can get the instruction to pull the image:



For example, to pull the EPAS16 operand from Ironbank, you can run:

```
docker pull registry1.dso.mil/ironbank/enterprisedb/edb-postgres-advanced-
16:16
```

If you want to pick a more specific tag or use a specific SHA, you need to find it from theHarbor page.

Installing the PG4K operator using the Iron Bank image

For installation, you will need a deployment manifest that points to your Iron Bank image. You can take the deployment manifest from the installation instructions for EDB PG4K. For example, for the 1.22.0 release, the manifest is available at `https://get.enterprisedb.io/cnp/postgresql-operator-1.22.0.yaml`. There are a couple of places where you will need to set the image path for the IronBank image.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  labels:
    app.kubernetes.io/name: cloud-native-
postgresql
  name: postgresql-operator-controller-
manager
  namespace: postgresql-operator-
system
spec:
  [… snipped
…]
  template:
    metadata:
      labels:
        app.kubernetes.io/name: cloud-native-
postgresql
    spec:
      containers:
      - args:
        - controller
          [… snipped
…]
        env:
        - name:
PULL_SECRET_NAME
          value: postgresql-operator-pull-
secret
        - name:
OPERATOR_IMAGE_NAME
          value: <INSERT-YOUR-OPERATOR-
IMAGE>

        [… snipped
…]
        image: <INSERT-YOUR-OPERATOR-
IMAGE>
```

If you wish for the operator to be deployed from Iron Bank directly, you will need to create and set the pull secret with the credentials to the registry, as described above.

It may be easier to get the image from Iron Bank with the instructions on the site, and from there, re-tag and publish it to a local registry, or push it directly to your Kubernetes nodes.

Once you have this in place, you can apply your manifest normally with `kubectl apply -f`, as described in the installation instructions.

Deploying clusters with EPAS operands using IronBank images

To deploy a cluster using the EPAS operand you must reference the Ironbank operand image appropriately in the `Cluster` resource YAML. For example, to deploy a PG4K Cluster using the EPAS 16 operand:

1. Create or edit a `Cluster` resource YAML file with the following content:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-example-
full
spec:
  imageName: registry1.dso.mil/ironbank/enterprisedb/edb-postgres-advanced-
17:17
  imagePullSecrets:
    - name: my_ironbank_secret
```

2. Apply the YAML:

```
kubectl apply -f <filename>
```

3. Verify the status of the resource:

```
kubectl get clusters
```

63 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.

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

> **Info**
>
> For more details, examples, and limitations, please refer to theofficial PostgreSQL documentation on Logical Replication.

EDB Postgres for Kubernetes enhances this capability by providing declarative support for key PostgreSQL logical replication objects:

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

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 `cnp publication create` plugin command.

When managing `Cluster` objects with **EDB Postgres for Kubernetes**, PostgreSQL publications can be defined declaratively through the `Publication` resource.

> **Info**
>
> 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:


```yaml
apiVersion: postgresql.k8s.enterprisedb.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 EDB Postgres for Kubernetes API reference for details on declaratively customizing replication targets.

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.

Reconciliation and Status

After creating a `Publication` , EDB Postgres for Kubernetes 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.

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.k8s.enterprisedb.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.

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.

EDB Postgres for Kubernetes simplifies subscription management by enabling you to define them declaratively using the `Subscription` resource.

> **Info**
>
> 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.k8s.enterprisedb.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
```

> **Info**
>
> For more details on configuring the `externalClusters` section, see the "Bootstrap" section 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.

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. EDB Postgres for Kubernetes ensures that the subscription is created or updated on the primary instance of the specified cluster.

Reconciliation and Status

After creating a `Subscription`, EDB Postgres for Kubernetes 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.

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:

```
apiVersion: postgresql.k8s.enterprisedb.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.

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

Addressing Schema Replication

The first limitation, related to schema replication, can be easily addressed using EDB Postgres for Kubernetes' 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.

Handling Sequences

While sequences are not automatically kept in sync through logical replication, EDB Postgres for Kubernetes provides a solution to be used in live migrations. You can use the `cnp` plugin to synchronize sequence values, ensuring consistency between the publisher and subscriber databases.

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 EDB Postgres for Kubernetes, you can easily set up, manage, and evaluate such scenarios in a Kubernetes environment.

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.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: freddie
spec:
  instances: 1

  imageName:
quay.io/enterprisedb/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.k8s.enterprisedb.io/v1
kind:
Publication
metadata:
  name: freddie-
publisher
spec:
  cluster:
    name: freddie
  dbname:
app
  name: publisher
  target:
    allTables: true
```

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.k8s.enterprisedb.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.k8s.enterprisedb.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 `cnp` plugin for simplicity:

```
kubectl cnp 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 `cnp` 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 cnp 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.

64 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)

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.

Upgrading a Minor Version in EDB Postgres for Kubernetes

To upgrade to a newer minor version, simply update the PostgreSQL container image reference in your cluster definition, either directly or via image catalogs. EDB Postgres for Kubernetes 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.

Major Version Upgrades

Major PostgreSQL releases introduce changes to the internal data storage format, requiring a more structured upgrade process.

EDB Postgres for Kubernetes 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" section 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.

Offline In-Place Major Upgrades

EDB Postgres for Kubernetes 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.

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**
>
> EDB Postgres for Kubernetes 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.

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.

> **Info**
>
> For detailed guidance on `pg_upgrade`, refer to the official PostgreSQL documentation.

Post-Upgrade Actions

If the upgrade is successful, EDB Postgres for Kubernetes:

- 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 EDB Postgres for Kubernetes 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.

Example: Performing a Major Upgrade

Consider the following PostgreSQL cluster running version 16:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  imageName: quay.io/enterprisedb/postgresql:16-minimal-bookworm
  instances: 3
  storage:
    size:
1Gi
```

You can check the current PostgreSQL version using the following command:

```
kubectl cnp 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.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: cluster-
example
spec:
  imageName: quay.io/enterprisedb/postgresql:17-minimal-bookworm
  instances: 3
  storage:
    size:
1Gi
```

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 cnp 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 cnp psql cluster-example -- app -c
'ANALYZE'
```

65 Preview Versions

EDB Postgres for Kubernetes 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**
>
> EDB Postgres for Kubernetes release candidates are not intended for use in production systems.

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 EDB Postgres for Kubernetes, additional changes may be implemented before the GA release.

Community Involvement

The stability of each EDB Postgres for Kubernetes 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.

Usage Advisory

The EDB Postgres for Kubernetes Community strongly advises against using preview versions of EDB Postgres for Kubernetes in production environments or active development projects. Although EDB Postgres for Kubernetes 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.

Current Preview Version

The current preview version is **1.26.0-rc3**.

The preview version is available on OpenShift.

There are two different manifests available depending on your subscription plan:

- Standard: The release candidate for the standard operator manifest.
- Enterprise: The release candidate for the enterprise operator manifest.

66 EDB private container registries

The images for the *EDB Postgres for Kubernetes* operator, as well as various operands, are kept in private container image registries under `docker.enterprisedb.com`.

> **Important**
>
> Access to the private registries requires an account with EDB and is reserved to EDB customers with a valid subscription plan. Credentials will be funneled through your EDB account.

> **Important**
>
> There is a bandwidth quota of 10GB/month per registry.

> **Note**
>
> When installing the operator and operands from the private registry, the license keys are not needed.

Which repository to choose?

EDB Postgres for Kubernetes is available with either of "EDB Enterprise Plan" or "EDB Standard Plan".

Depending on your subscription plan, EDB Postgres for Kubernetes will be in one of the following repositories, as described in the table below:

Plan	Repository
EDB Standard Plan	`k8s_standard`
EDB EnterpriseDB Plan	`k8s_enterprise`

The name of the repository shall be used as the *Username* when you try to login to the EDB container registry, for example through `docker login` or a `kubernetes.io/dockerconfigjson` pull secret.

> **Important**
>
> Each repository contains all the images you can access with your plan. You don't need to connect to different repositories to access different images, such as operator or operand images.

How to retrieve the token

In the repositories page in EDB, you'll find an *EDB Repos 2.0* section where a `Repo Token` is shown, obscured. The same token is also be available in your Account profile, labeled as `Repos 2.0 token`.

Next to the token you'll find a button to copy the token, and an eye icon in case you want to view the content of the token as clear text. The token shall be used as the *Password* when you try to access the EDB container registry.

Example with `docker login`

You should be able to login via Docker from your terminal. We suggest you copy the Repo Token using the `Copy Token` button. The `docker` command below will prompt you for a username and a password.

As explained above, the username should be the repository you are trying to access while the password is the token you just copied.

```
$ docker login docker.enterprisedb.com
Username:
k8s_enterprise
Password:
Login Succeeded
```

Operand images

EDB Postgres for Kubernetes supports various PostgreSQL distributions that have images available from the same private registries:

- EDB Postgres Advanced (EPAS)
- EDB Postgres Extended (PGE)

> **Note**
>
> PostgreSQL images are not available in the private registries, but are readily available on `quay.io/enterprisedb/postgresql` or `ghcr.io/enterprisedb/postgresql`.

These images follow the requirements and the conventions described in the "Container image requirements" page of the EDB Postgres for Kubernetes documentation.

In the table below you can find the image name prefix for each Postgres distribution:

Postgres distribution	Image name	Repositories
EDB Postgres Extended (PGE)	`edb-postgres-extended`	`k8s_standard`, `k8s_enterprise`
EDB Postgres Advanced (EPAS)	`edb-postgres-advanced`	`k8s_enterprise`

How to deploy clusters with EPAS or PGE operands

If you have already installed the EDB Postgres for Kubernetes operator from the private registry, you must have already set up an image pull secret. If you haven't, the next section may be of interest to you.

If you have an existing installation of the operator, in order to pull images for EPAS or PGE from the private registry, you will need to create a `kubernetes.io/dockerconfigjson` pull secret.

You can create a pull secret from credentials.

```
kubectl create secret docker-registry registry-pullsecret
\
  -n <CLUSTER-NAMESPACE> --docker-server=docker.enterprisedb.com
\
  --docker-username=<REGISTRY-NAME> \
  --docker-password=<TOKEN>
```

As mentioned above, the `docker-username` is the name of your registry, i.e. `k8s_standard` or `k8s_enterprise`. The `docker-password` is the token retrieved from the EDB portal.

Once your pull secret is created, remember to set the `imagePullSecrets` field in the cluster manifest in addition to the `imageName`. The manifest below will create a cluster running PG Extended from the `k8s_enterprise` repository.

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: postgresql-extended-
cluster
spec:
  instances: 3
  imageName: docker.enterprisedb.com/k8s_enterprise/edb-postgres-extended:16.2
  imagePullSecrets:
  - name: registry-pullsecret

  storage:
    storageClass:
standard
    size:
1Gi
```

How to install the operator using the EDB private registry

As mentioned above, the *username* for docker is the name of your repository, and the token is the *password*. The same credentials can be used for kubernetes to access the registry by setting up a `kubernetes.io/dockerconfigjson` pull secret.

As mentioned in the installation document, there are several different ways to install the operator.

If you are going to install using images from the private registry, you will need to create a pull secret, as we have mentioned, and also customize the `OPERATOR_IMAGE_NAME` parameter in the deployment manifest.

We suggest to use the Helm chart for installation, which will take care of creating the pull secret and customizing the operator image repository for you.

You can find more information in the Helm chart page.

As an example, the following command (provided the token) will install the PG4K operator when using the repository from the EDB EnterpriseDB Plan:

```
helm upgrade --install edb-pg4k \
  --namespace postgresql-operator-system
\
  --create-namespace \
  --set image.repository=docker.enterprisedb.com/k8s_enterprise/edb-postgres-for-kubernetes
\
  --set image.imageCredentials.username=k8s_enterprise \
  --set image.imageCredentials.password=<ENTITLEMENT_TOKEN> \
  --set image.imageCredentials.create=true
\
  --set "imagePullSecrets[0].name"=postgresql-operator-pull-secret \
  --set config.data.PULL_SECRET_NAME=postgresql-operator-pull-secret \
  edb/edb-postgres-for-
kubernetes
```


67 Service Management

A PostgreSQL cluster should only be accessed via standard Kubernetes network services directly managed by EDB Postgres for Kubernetes. For more details, refer to the "Service" page of the Kubernetes Documentation.

EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes usage.

While this setup covers most use cases for accessing PostgreSQL within the same Kubernetes cluster, EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes 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.

Disabling Default Services

You can disable any or all of the `ro` and `r` default services through the `managed.services.disabledDefaultServices` option.

> **Important**
>
> The `rw` service is essential and cannot be disabled because EDB Postgres for Kubernetes 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"]
```

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` stanza 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. EDB Postgres for Kubernetes 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:

```yaml
#
<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.

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.

68 Tablespaces

A tablespace is a robust and widely embraced feature in database management systems. It offers a powerful means to enhance the vertical scalability of a database by decoupling the physical and logical modeling of data. Essentially, it serves as a technique for physical database modeling, enabling the efficient distribution of I/O operations across multiple volumes on distinct storage. It thereby optimizes performance through parallel on-disk read/write operations.

In the context of the database industry, tablespaces play a strategic role, particularly when paired with table partitioning, a logical database modeling technique. They prove instrumental in managing large-scale databases and are also used for tasks such as separating tables from indexes or executing temporary operations.

Tablespaces in PostgreSQL have been playing a pivotal role since 2005 (version 8.0), while declarative partitioning was introduced in 2017 (version 10). Consequently, tablespaces are seamlessly integrated into all supported releases of PostgreSQL. Quoting from the PostgreSQL documentation on tablespaces:

> By using tablespaces, an administrator can control the disk layout of a PostgreSQL installation. This is useful in at least two ways.
>
> - First, if the partition or volume on which the cluster was initialized runs out of space and cannot be extended, a tablespace can be created on a different partition and used until the system can be reconfigured.
> - Second, tablespaces allow an administrator to use knowledge of the usage pattern of database objects to optimize performance.

Declarative tablespaces

EDB Postgres for Kubernetes 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, EDB Postgres for Kubernetes' 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. EDB Postgres for Kubernetes 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 EDB Postgres for Kubernetes.

Using declarative tablespaces

Using declarative tablespaces is straightforward. You can find a full example in `cluster-example-with-tablespaces.yaml`.

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.

EDB Postgres for Kubernetes 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:

```yaml
status:

  <- snipped -
>
  tablespacesStatus:
  - name:
atablespace
    state: reconciled
  - name: another_tablespace
    state: reconciled
  - name: tablespace1
    state: reconciled
```

Storage classes and tablespaces

You can use different storage classes for your tablespaces, just as you can for PGDATA and WAL volumes. This is a convenient way of optimizing your resources, balancing performance and costs of your storage based on data access usage and expectations.

This example helps to explain the feature:

```yaml
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: yardbirds
spec:
  instances: 3

  storage:
    size: 10Gi
  walStorage:
    size: 10Gi
  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:

```sql
CREATE TABLE facts_202312 PARTITION OF facts
    FOR VALUES FROM ('2023-12-01') TO ('2024-01-
01')
    TABLESPACE current;
```

> **Important**
>
> This example assumes you're familiar with PostgreSQL declarative partitioning.

Tablespace ownership

By default, unless otherwise specified, tablespaces are owned by the `app` application user, as defined in `.spec.bootstrap.initdb.owner`. See Bootstrap a new cluster for details. This default behavior works in most microservice database use cases.

You can set the owner of a tablespace in the `owner` stanza, for example the `postgres` user, like in the following excerpt:

```
  #
...
  tablespaces:
    - name: clapton
      owner:
        name:
postgres
      storage:
        size:
1Gi
```

> **Important**
>
> If you change the ownership of a tablespace, make sure that you're using an existing role. Otherwise, the status of the cluster reports the issue and stops reconciling tablespaces until fixed. It's your responsibility to monitor the status and the log and to promptly intervene by fixing the issue.

If you define a tablespace with an owner that doesn't exist, EDB Postgres for Kubernetes 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
```

Backup and recovery

EDB Postgres for Kubernetes handles backup of tablespaces (and the relative tablespace map) both on object stores and volume snapshots.

> **Warning**
>
> By default, backups are taken from replica nodes. A backup taken immediately after creating tablespaces in a cluster can result in an incomplete view of the tablespaces from the replica and thus an incomplete backup. The lag will be resolved in a maximum of 5 minutes, with the next reconciliation.

Once a cluster with tablespaces has a base backup, you can restore a new cluster from it. When it comes to the recovery side, it's your responsibility to ensure that the `Cluster` definition of the recovered database contains the exact list of tablespaces.

Replica clusters

Replica clusters must have the same tablespace definition as their origin. The reason is that tablespace management commands like `CREATE TABLESPACE` are WAL logged and are replayed by any physical replication client (streaming or by way of WAL shipping).

It's your responsibility to ensure that replica clusters have the same list of tablespaces, with the same name. Storage class and size might vary.

For example:

```
spec:

  #
...
  bootstrap:
    recovery:
      # ... your selected recovery
method

  tablespaces:
    - name: tbs1
      storage:
        size:
1Gi
    - name: tbs2
      storage:
        size:
2Gi
    - name: tbs3
      storage:
        size:
2Gi
```

Temporary tablespaces

PostgreSQL allows you to define one or more temporary tablespaces to create temporary objects (temporary tables and indexes on temporary tables) when a `CREATE` command doesn't explicitly specify a tablespace, and to create temporary files for purposes such as sorting large data sets. When no temporary tablespace is specified, PostgreSQL uses the default tablespace of a database, which is currently the main `PGDATA` volume.

When you specify more than one temporary tablespace, PostgreSQL randomly picks one the first time a temporary object needs to be created in a transaction. Then it sequentially iterates through the list.

Temporary tablespaces also work like regular tablespaces with regard to backups.

EDB Postgres for Kubernetes provides the `.spec.tablespaces[*].temporary` option to determine whether to add a tablespace to the `temp_tablespaces` PostgreSQL parameter and thus become eligible to store temporary data that doesn't have an explicit tablespace assignment.

```
spec:
  [...]
  tablespaces:
    - name:
atablespace
      storage:
        size:
1Gi
      temporary: true
```

They can be created at initialization time or added later, requiring a rolling update. The `temporary: true/false` option adds or removes the tablespace name to or from the list of tablespaces in the `temp_tablespaces` option. This change doesn't require a restart of PostgreSQL.

Although temporary tablespaces can also work as regular tablespaces (meaning that users can also host regular data on them while using them for temporary operations), we recommend that you don't mix the two workloads.

See the PostgreSQL documentation on `temp_tablespaces` for details.

kubectl plugin support

The kubectl status plugin includes a section dedicated to tablespaces that offers a convenient overview, including tablespace status, owner, temporary flag, and any errors:

```
[...]

Tablespaces
status
Tablespace          Owner  Status      Temporary
Error
----------          -----  ------      ---------  ----
-
atablespace         app    reconciled  true
another_tablespace  app    reconciled  true
tablespacea1        app    reconciled  false

Instances
status
[...]
```

Limitations

Currently, you can't remove tablespaces from an existing EDB Postgres for Kubernetes cluster.