
EDB
Postgres for the AI Generation
Postgres on Kubernetes Workshop

Requirements for running the hands-on locally

Have these things ready prior to joining the workshop

- Bash shell available (i.e. either running Linux or Mac, or WSL on Windows laptops)
- Internet access (for downloading code and images)
- Have at least a little knowledge of Kubernetes and Postgres (youtube is a great resource)

- Docker is installed
- Run without errors
  - $ docker ps

- k3d or kind is installed (k3d preferred as it seems there are less issues with it)

- Create local Kubernetes cluster (run either) - remove it again (we don't need it)
  - $ kind create cluster –name cnpg-workshop
  - $ k3d cluster create cnpg-workshop

Agenda

Session
Introduction to Kubernetes
Introduction to Operators and CNPG
DBA Mindset Shift - how Kubernetes changes the game (and doesn't)
Interactive session
Additional functionality in CNPG
A shameless plug + Discussion
Lunch

Introduction to Kubernetes

Cloud Native!!

Let's play Buzzword Bingo!

Benefits:

Increased Agility, Improved Scalability, Higher Resilience, Optimization of resources, Lower Time to Market, …

Core terms:

Decoupling, Configuration as Code, Declarative APIs, Immutable architecture, Services Meshes, Change Management (DevOps procedures)

Implementation details:

Microservices, Containers, Orchestration, Cloud Agnostic, CI/CD, CoDe, … (DevOps???)

From Servers to Containers - why Kubernetes matters

A new way to manage workloads

- From physical servers (bare metal) to virtual servers (VMs)
  - Full operating system
- Containers
  - Shares the OS Kernel with the Host system but runs in complete isolation.

  A container is essentially a package that contain everything an application needs to run.

  (Think of it as a small, self-contained box that holds application code and dependencies)

  Run in isolation using Linux Namespaces and use cgroups to put cap resources.

- Docker - the beginning of it all
  - Which is why the ‘code’ used to create a container is often called a “Dockerfile”
  - The official name today though is “Containerfile”

Example of a Containerfile

This is what creates a ‘Container’

```
# Use the official PostgreSQL image as a base.
FROM postgres:16.2-alpine

# Set environment variables for the database configuration.
ENV POSTGRES_USER=admin
ENV POSTGRES_PASSWORD=securepassword
ENV POSTGRES_DB=mydb

# Copy our custom initialization script and our custom startup script into the container.
# init-schema.sql: Contains your database schema (e.g., CREATE TABLE statements).
# start.sh: Our custom bash script to run the database.
COPY ./init-schema.sql /docker-entrypoint-initdb.d/
COPY ./start.sh /usr/local/bin/

# Make the start.sh script executable.
RUN chmod +x /usr/local/bin/start.sh

# The CMD instruction now tells the container to execute our custom bash script.
CMD ["start.sh"]
```

How Kubernetes uses a container

- How do you run a container on Kubernetes?
  - You create a Pod with the container inside

A Pod is the smallest unit in Kubernetes. It can contain a number of containers.

The containers are either the actual application or something that initialises the application.

Or, they can add functionality, like network filtering, log collection, …

The magic in Kubernetes (Operator or Control Loop)

- Once you have a pod, you can deploy that to Kubernetes
- Either as:

  Single Pod - meaning no special handling

  Deployment - a way to automatically scale to multiple pods and apply HA

If you tell Kubernetes to run a deployment with 3 replicas of your Pod it will do so, and **maintain that count even if one of the pods fail**. Pods are automatically killed and created if Kubernetes detects a problem with a pod.

This is what is known as a Kubernetes Control Loop - also known as an “Operator”.

Kubernetes natively has one Operator - but we can add our own (ones that understand our app)

Why is Kubernetes special?

- Declarative deployments vs. imperative
  - Kubernetes tends to be declarative (though some things *can* be done imperatively - you really shouldn't)
- Leans into Infrastructure as Code, DevOps etc.
- Perfect for MicroServices etc.

But why run the databases in Kubernetes?

- Database workload benefit from Kubernetes as well (automatic healing, scale in/out, etc.)
- One platform to run them all - no "context switch" between different environments

Introduction to CloudNativePG and EDB

What is CNPG?

CNPG = Cloud Native Postgres (https://cloudnative-pg.io/)

CloudNativePG is the Kubernetes operator that covers the full lifecycle of a highly available PostgreSQL database cluster with a primary/standby architecture, using native streaming replication.

- An Operator (Control Loop) plus the images needed to run databases using the operator

CNPG is designed bottom-up for Kubernetes. This sets it apart from other available operators where the architecture is Patroni-based and then patched to ‘work’ in Kubernetes.

Quick facts about CNPG

- Immutable images (secure by default)
  - Only user available when opening a shell: postgres or enterprisedb (depending on database type)
- Heavily tested and vulnerability tested by EDB prior to release
- CNPG uses StorageClass (if you want)
- You don’t manage StatefulSets or Persistent Volumes or even Persistent Volume Claims
  - CNPG does that for you
- Maturity level for CNPG is the highest possible: **AutoPilot**

But wait … which storage provider should we use?

- Databases need fast storage. Until someone builds a Network Storage Solution faster than local storage …

  You should use local storage - but you can configure something else

Interactive session
It’s time to go hands-on!

Use case
The environment

Features shown during the demo

- Installation operator and CNPG plugin
- Check the CloudNativePG operator status
- Installation Backup (Barman) Operator
- Setup PostgreSQL Cluster on Kubernetes
- Create the backup of the PostgreSQL Cluster
- Run out-of-the-place recovery
- Run switchover
- Simulate failover szenario
- Run minor upgrade using rolling upgrade approach
- Scale out/down
- Test fencing
- Test hibernation
- Database migration
- Major upgrade (in-place)
- Monitoring

Last CloudNativePG tested version is 1.26

This demo is in 

https://github.com/EnterpriseDB/cnpg-hands-on

Download or pull the code

cd to the new directory.

Setup the demo env

Call to action: Create k3d cluster

- In the terminal 1:
  - Go to the directory /home/workshop/workshop/cnp-demo:
  - Create the k3d cluster - run the script:

    ./00_start_infra.sh
  - Check the cluster:

    kubectl get nodes

    kubectl get pods

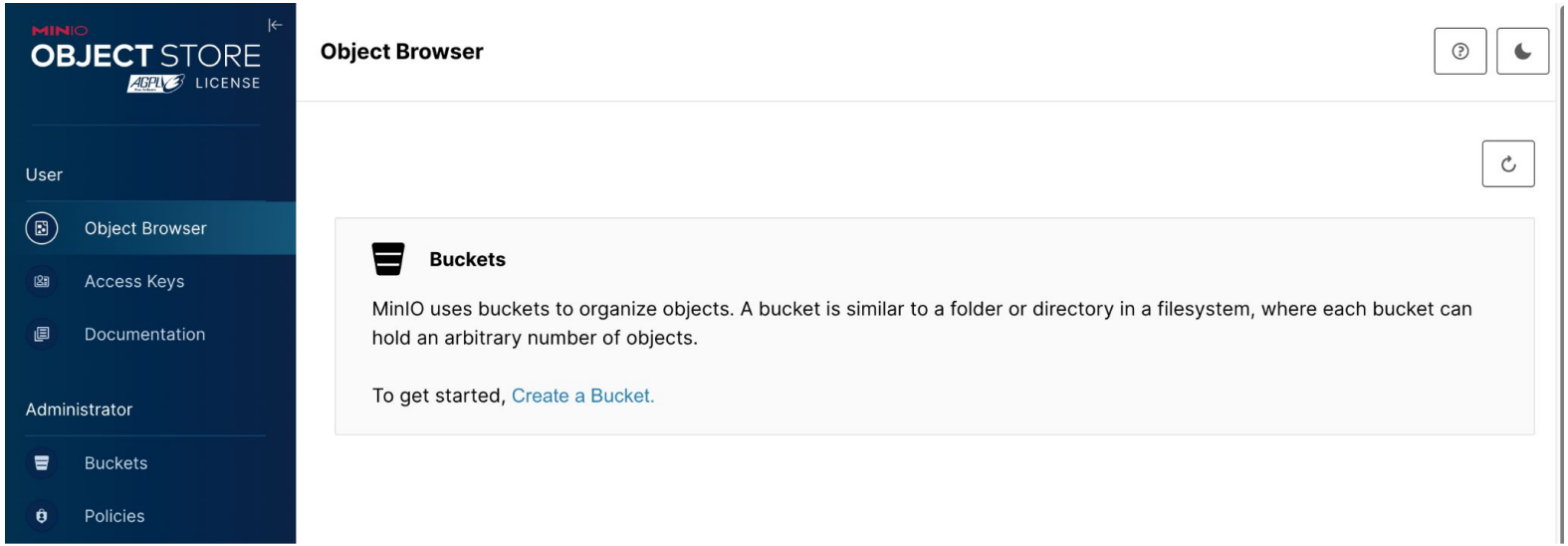
Call to action: Run Minio server

- In the terminal 1:
  - Go to the directory /home/workshop/workshop/cnp-demo:
  - Start MinIO:

    ./start_minio_docker_server.sh &

Call to action: Connect to the MinIO server

- In the browser open the new tab and go to
  - http://localhost:9001
  - Connect as user **admin** with the password: **password**
- The page will appear:



Use case
Plug-in installation

The “cnpg” plugin for kubectl

- The official CLI for CloudNativePG
  - Available also as RPM or Deb package
- Extends the ‘kubectl’ command:
  - Customize the installation of the operator
  - Status of a cluster
  - Perform a manual switchover (promote a standby) or a restart of a node
  - Issue TLS certificates for client authentication
  - Declare start and stop of a Kubernetes node maintenance
  - Destroy a cluster and all its PVC
  - Fence a cluster or a set of the instances
  - Hibernate a cluster
  - Generate jobs for benchmarking via pgbench and fio
  - Issue a new backup
  - Start pgadmin

```
Name:               cluster-example
Namespace:          default
System ID:          7100921006673293335
PostgreSQL Image:   ghcr.io/cloudnative-pg/postgresql:14.3
Primary instance:   cluster-example-2
Status:             Cluster in healthy state
Instances:          3
Ready instances:    3
Current Write LSN:  0/C000060 (Timeline: 4 - WAL File: 0000000400000000000000C)

Certificates Status
Certificate Name            Expiration Date               Days Left Until Expiration
----------------            ---------------               --------------------------
cluster-example-replication  2022-08-21 13:15:00 +0000 UTC  89.95
cluster-example-server       2022-08-21 13:15:00 +0000 UTC  89.95
cluster-example-ca           2022-08-21 13:15:00 +0000 UTC  89.95

Continuous Backup status
First Point of Recoverability:  2022-05-23T13:37:08Z
Working WAL archiving:          OK
WALs waiting to be archived:    0
Last Archived WAL:              0000000400000000000000B   @   2022-05-23T13:42:09.37537Z
Last Failed WAL:                -

Streaming Replication status
Name              Sent LSN   Write LSN  Flush LSN  Replay LSN  Write Lag  Flush Lag  Replay Lag  State      Sync State  Sync Priority
----              --------   ---------  ---------  ----------  ---------  ---------  ----------  -----      ----------  -------------
cluster-example-3  0/C000060  0/C000060  0/C000060  0/C000060   00:00:00   00:00:00   00:00:00    streaming  async       0
cluster-example-1  0/C000060  0/C000060  0/C000060  0/C000060   00:00:00   00:00:00   00:00:00    streaming  async       0

Instances status
Name              Database Size  Current LSN  Replication role  Status  QoS         Manager Version
----              -------------  -----------  ----------------  ------  ---         ---------------
cluster-example-3  33 MB          0/C000060    Standby (async)   OK      BestEffort  1.15.0
cluster-example-2  33 MB          0/C000060    Primary           OK      BestEffort  1.15.0
cluster-example-1  33 MB          0/C000060    Standby (async)   OK      BestEffort  1.15.0
```

Call to action: Install CNPG plugin

- In the terminal 1:
  - Install CNPG plugin:

```
./01_install_plugin.sh
```

Use case
Operator installation

Call to action: Operator Installation

- Install the operator
- Check the installed CNPG Operator in the terminal
- In the terminal 1:
  - Install CNPG operator:

    ./02_install_operator.sh

  - Check the installation of the operator (try several times):

    ./03_check_operator_installed.sh

Use case
Backup Configuration

Backup and Recovery - Backup Methods

- CNPG Operator supports several backup methods:
  - Database backup to the object storage - with Barman tool
    - Until the CNPG Version 1.26 barman was integrated into CNPG Operator
    - From version 1.26 onwards, EDB has provided a comprehensive backup operator based on Barman technology - **Barman Cloud Plugin**
  - Native volume snapshots
- WAL management
  - Object store
- Physical Base backups
  - Object store
  - Kubernetes level backup integration (Velero/OADP, Veem Kasten K10, generic interface)*
  - Kubernetes Volume Snapshots

Backup and Recovery - Choosing the best ssolution

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

Backup and Recovery - Barman Cloud Plugin

- Features:
  - Continuous physical backup on “backup object stores”
  - Scheduled and on-demand base backups
  - Continuous WAL archiving (including parallel)
  - Backup can be created from primary or a standby
  - Backups encryption
  - Support for parallel backups
  - Support for backup compression
  - Support for recovery window retention policies (e.g. 30 days)

Backup and Recovery - Barman Cloud Plugin

- Recovery means creating a new cluster starting from a “recovery object store”
  - Then pull WAL files (including in parallel) and replay them
  - Full (End of the WAL) or PITR
- Both rely on Barman Cloud technology
  - AWS S3
  - Azure Storage compatible
  - Google Cloud Storage
  - MinIO

Backup and Recovery - Kubernetes Volume Snapshot

- Transparent support for:
  - Incremental backup and recovery at block level
  - Differential backup and recovery at block level
  - Based on copy on write
- Leverage the storage class to manage the snapshots, including:
  - Data mobility across network (availability zones, Kubernetes clusters, regions)
  - Relay files on a secondary location in a different region, or any subsequent one
  - Encryption
- Enhances Very Large Databases (VLDB) adoption

Backup & Recovery via Snapshots: some numbers

Let's now talk about some initial benchmarks I have performed on volume snapshots using 3 `r5.4xlarge` nodes on AWS EKS with the `gp3` storage class. I have defined 4 different database size categories (tiny, small, medium, and large), as follows:

Cluster name	Database size	pgbench init scale	PGDATA volume size	WAL volume size	pgbench init duration
tiny	4.5 GB	300	8 GB	1 GB	67s
small	44 GB	3,000	80 GB	10 GB	10m 50s
medium	438 GB	3,0000	800 GB	100 GB	3h 15m 34s
large	4,381 GB	300,000	8,000 GB	200 GB	32h 47m 47s

The table below shows the results of both backup and recovery for each of them.

Cluster name	1st backup duration	2nd backup duration after 1hr of pgbench	Full recovery time
tiny	2m 43s	4m 16s	31s
small	20m 38s	16m 45s	27s
medium	2h 42m	2h 34m	48s
large	3h 54m 6s	2h 3s	2m 2s

https://www.enterprisedb.com/postgresql-disaster-recovery-with-kubernetes-volume-snapshots-using-cloudnativepg

Call to action: Barman Cloud Operator configuration

- Installing cert-manager
- Installing Barman Cloud operator
- Configuration the storage in MinIO

- In the terminal 1:
  - Install Barman Operator:

    ./04_install_barman_plugin.sh

  - Check the installation of the barman operator:

    ./05_check_barman_plugin.sh

Use case
Create the postgres cluster

Synchronizing the state of a Postgres database

- Being a DBMS, PostgreSQL is a stateful workload in Kubernetes
- Stateless workloads achieve HA and DR mainly through traffic redirection
- Stateful workloads require the state to be replicated in multiple locations:
  - **Storage-level** replication
  - **Application-level** replication (in our case, application = Postgres)
- Postgres has a very robust and powerful native replication system
  - We've built it
  - Founded on the Write Ahead Log
  - Read-only standby servers
  - Supports also synchronous replication controlled at the transaction level
- **We recommend application-level** over storage-level replication for Postgres

Bootstrap - different ways of creating a cluster

- Create a new cluster from scratch
  - “initdb”: named after the standard “initdb” process in PostgreSQL that initializes an instance
- Create a new cluster from an existing one:
  - Directly (“pg_basebackup”), using physical streaming replication
  - Directly (logical backup/restore) using pg_dump and pg_restore
  - Indirectly (“recovery”), from an object store
    - To the end of the WAL
      - Can be used to start independent replica clusters in continuous recovery
    - Using PITR

Storage management

- Storage is the most critical component for a database
- Direct support for Persistent Volume Claims (PVC)
  - We deliberately do not use Statefulsets
- The PVC storing the PGDATA is central to CloudNativePG
  - Our motto is: “PGDATA is worth a 1000 pods”
- Storage agnostic
- Freedom of choice
  - Local storage
  - Network storage
- Automated generation of PVC
- Support for PVC templates
  - Storage classes

Call to action: Configure and Install the Postgres cluster

- In the terminal 1:
  - Create the yaml file:

    ./06_get_cluster_config_file.sh
  - Create the postgres cluster:

    ./07_install_cluster.sh

- In the terminal **2** - Check the Postgres cluster status::

  ./08_show_status.sh

Call to action: Create table test with 1000 rows

- Once cluster is running … (minimum the primary) insert the data - run the script:

```
./09_insert_data.sh
```

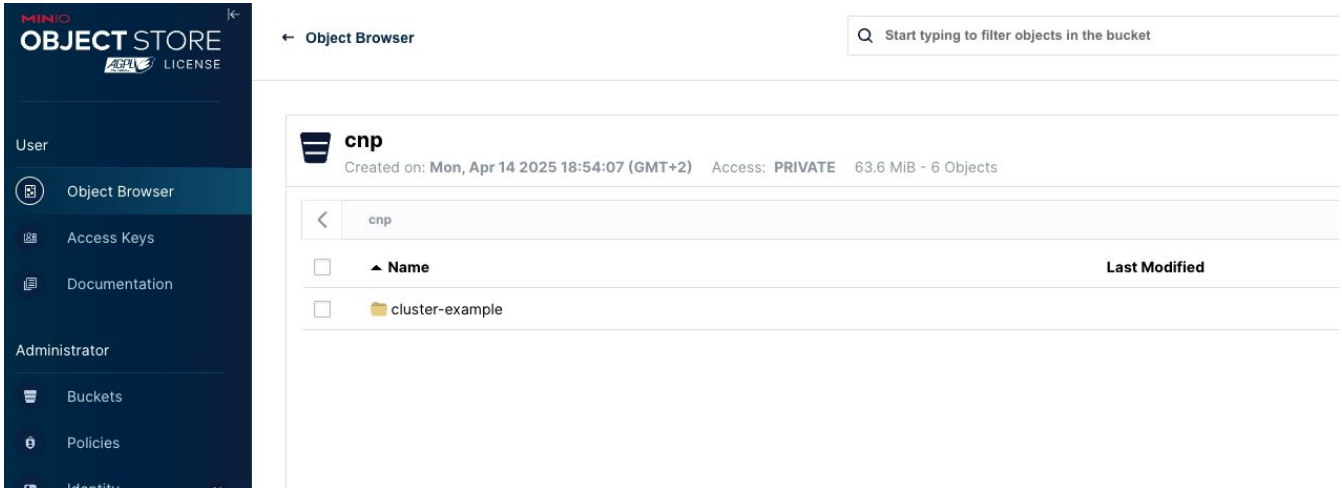
Use case
Backup & Restore

Backup demonstration

- Create the full backup
- Check Backup in MinIO UI
- Restore the database from the backup

Call to action: Connect to the MinIO server

- In the browser open the new tab and go to
  - http://<vm ip>:9001
  - Connect as user **admin** with the password: **password**
- The page will appear:



- Click on cluster-example and check the backup of WAL files

Call to action: Create the full backup

- With this step we will:
  - Create the full backup of the postgres cluster in the MinIO storage:
- In the terminal 1:
  - Run the script:

    ./10_backup_cluster.sh

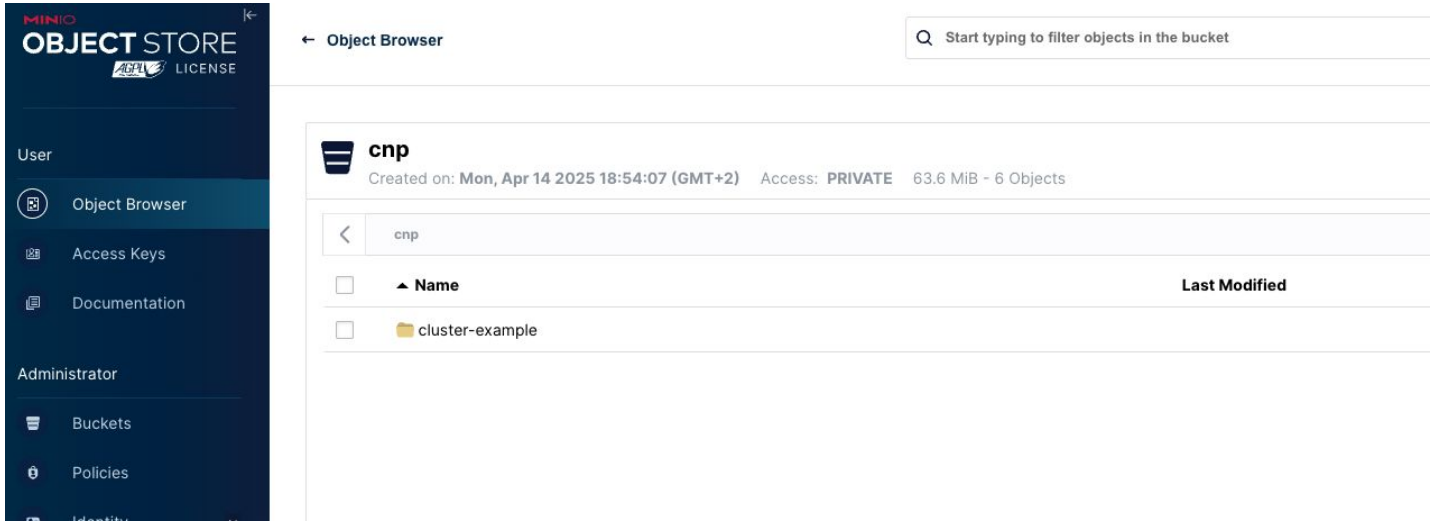
  - Check the backup status:

    ./11_backup_describe.sh

  - Check the created backup in the MinIO GUI

Call to action: Check Backup in MinIO UI

- Check the created backup in the MinIO GUI



Call to action: Restore the database from the backup

- With this step we will:
  - Create the new cluster cluster-restore
  - Restore the full backup created in the previous step in the new cluster:
- In the terminal 1:
  - Run the restore:

    ./12_restore_cluster.sh
  - Check the creation status:

    kubectl get pods -w # after creation stop the execution with <ctrl>+c
  - Check the table test in the cluster-restore, run the script:

    ./13_check_restore.sh
  - Delete the cluster-restore to avoid resource problems during the workshop:

    kubectl delete cluster cluster-restore

Use case
Promote & Failover

Call to action: Promote standby to the primary

- With this step we will:
  - promote a pod in the cluster to primary, so you can start with maintenance work or test a switch-over situation in your cluster
  - Promote is the option of the cnpg plugin

- In the terminal 1:
  - Run the script:

    ./14_promote.sh

- In the terminal **2**:
  - Check the failover cluster status:

    ./08_show_status.sh

Call to action: Run failover test

- With this step we will:
  - Delete the primary database of the cluster cluster-example
  - The operator will:
    - detect failure of the primary
    - promote standby to the new primary and create the new standby according to the configuration (3 instances)
- In the terminal 1:
  - Run the script:

    ./15_failover.sh
- In the terminal **2**:
  - Check the failover cluster status:

    ./08_show_status.sh

Use case
Minor Upgrade

Rolling updates

- Update of a deployment with ~zero downtime
  - Standby servers are updated first
  - Then the primary:
    - supervised / unsupervised
    - switchover / restart
- When they are triggered:
  - Security update of Postgres images
  - Minor update of PostgreSQL
  - Configuration changes when restart is required
  - Update of the operator
    - Unless in-place upgrade is enabled

Call to action: Run the Promote and Upgrade

- With this step we will:
  - Run the postgres minor update from the version 16.4 to 16.5
- In the web terminal **2** (prepare a terminal for status - and one to run the admin-commands):
  - Check the upgrade status:

    ./08_show_status.sh
- In the terminal **1**:
  - Run the script:

    ./16_minor_upgrade.sh
- In terminal **2**: (prepare a terminal for status - and one to run the admin-commands):
  - check Postgres version: PostgreSQL Image: quay.io/enterprisedb/postgresql:16.5

Use case
Scale-out and scale-down

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.

Call to action: Scale-out the postgres cluster

- With this step we will:
  - Add the 1 standby to the cluster
- In the web terminal 1:
  - Run the script:

    ./18_scale_out.sh (using –replicas=X… another way would be to update the YAML)

- In the web terminal **2**:
  - Check the cluster status:

    ./08_show_status.sh

Call to action: Scale-down the postgres cluster

- With this step we will:
  - Remove 2 standby pods from the cluster

- In the web terminal 1:
  - Run the script:

    ./19_scale_down.sh

- In the web terminal **2**:
  - Check the cluster status:

    ./08_show_status.sh

Use Case Fencing

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.

Call to action: Stop postgres process on the pod

- In the web terminal 1:
  - Run the script:

    ./20_fencing.sh on

- In the web terminal **2**:
  - Check the cluster status:

    ./08_show_status.sh

Call to action: Start the postgres process on the pod

- In the terminal 1:
  - Run the script:

    ./20_fencing.sh off

- In the terminal **2**:
  - Check the cluster status:

    ./08_show_status.sh

Use case Hibernation

Hibernation

- CloudNativePG supports **hibernation** of a running PostgreSQL cluster in a declarative manner
  - through the cnpg.io/hibernation annotation
- Hibernation enables saving CPU power by removing the database pods while keeping the database PVCs
- This feature simulates scaling to 0 instances.

Call to action: Stop the postgres cluster

- In the terminal 1:
  - Run the script:

    ./21_hibernation.sh on

- In the terminal **2**:
  - Check the cluster status:

    ./08_show_status.sh

Call to action: Start the postgres cluster

- In the terminal 1:
  - Run the script:

    ./21_hibernation.sh off

- In the terminal **2**:
  - Check the cluster status:

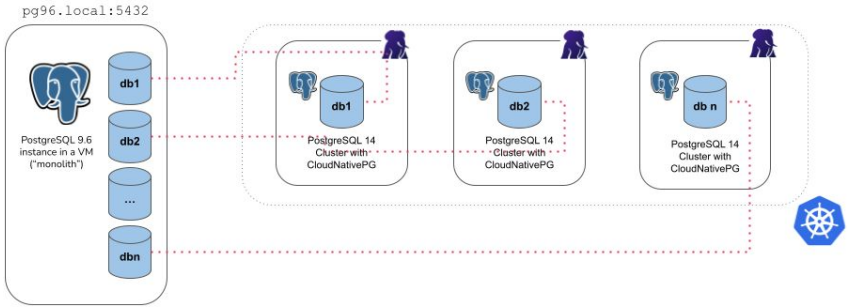
    ./08_show_status.sh

Use case
Database Migration

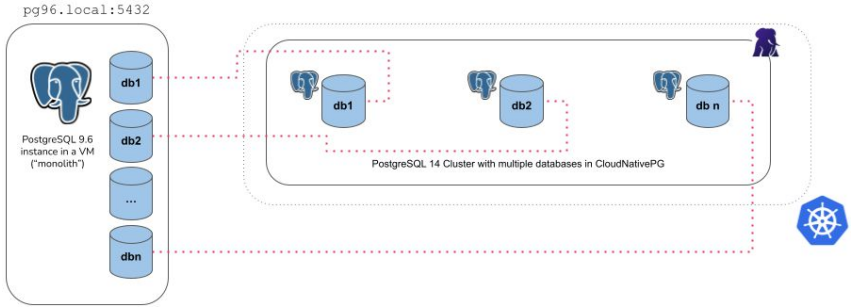
Database Migration

- In this step we will migrate the app database from our existing cluster to the new cluster
- We will create the yaml file with the setting “import” in the bootstrap section
- The operator uses internally postgres tools pg_dump and pg_restore
- This method can be used to **migrate** another database or to run **out-of-the-place upgrade**
- Possible settings:

Microservices:



Monolith:



Call to action: Migrate the cluster example to the new cluster

- In the terminal 1:
  - Create the new cluster v17 and migrate the database app from the cluster cluster-example::

    ./22_major_upgrade_by_copy.sh
  - Connect to the migrated cluster and check version and data::

    ./23_verify_data_migrated_16_17.sh

Use case
Major Upgrade

Database Major Upgrade

- CNPG supports 3 types of major upgrade:
  - **In-place**: you should test carefully this method before you migrate your production database. You should create the fallback scenario in case of failure
  - **Copy**: we discussed this topic in “Database migration”
  - **Logical replication:** you can create the new database in the new cluster and replicate the data between two clusters.
    - CloudNativePG enhances this capability by providing declarative support for key PostgreSQL logical replication objects:
      - Publications via the Publication resource
      - Subscriptions via the Subscription resource

Call to action: In-place major upgrade

- In this step we will:
  - upgrade the existing cluster cluster-example from the Postgres version 16 to 17
  - choose in-place method

Note: For this action the downtime should be planned: during the upgrade process postgres instances are not available!

- In the terminal 1:
  - Run in-place major upgrade for the cluster cluster-example:

    ./24_major_upgrade_in_place.sh
  - Connect to the upgraded cluster and check version:

    ./25_verify_major_upgrade_16_17.sh

Use case
Monitoring

Call to action: Setup monitoring

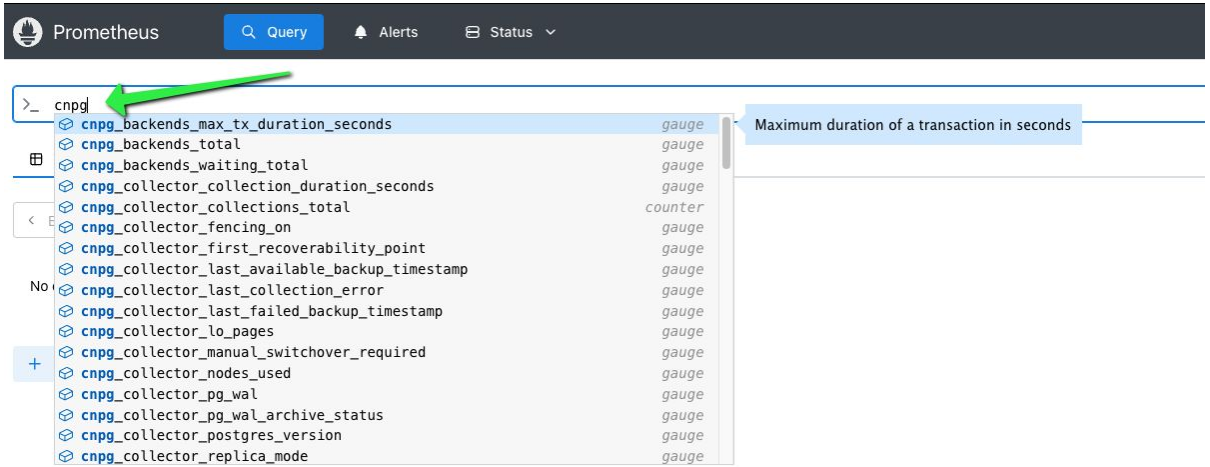
- In the terminal 1:
  - Change the directory::
  - Install the prometheus rules:

    ./monitoring/01_prometheus_rules.sh
  - Start port forwarding for prometheus and grafana:

    ./monitoring/02_port_forwarding_prometheus_grafana.sh
- Download the Grafana Dashboard to your laptop:
  - https://github.com/cloudnative-pg/grafana-dashboards/blob/main/charts/cluster/grafana-dashboard.json

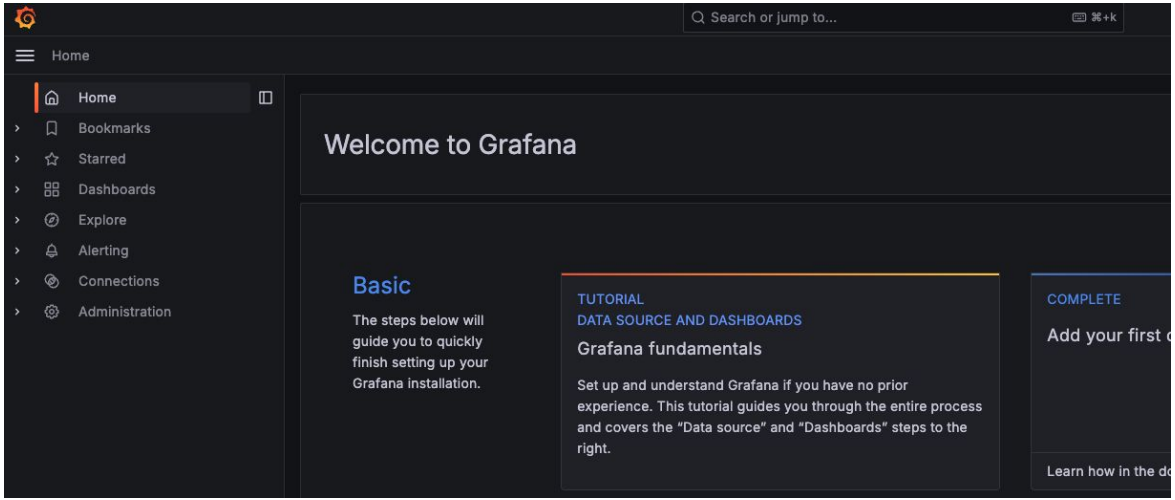
Call to action: Explore Prometheus

- In the browser open the new tab and go to
  - http://<vm ip>:9090
- The prometheus page will appear - search for “cnpg” metrics:



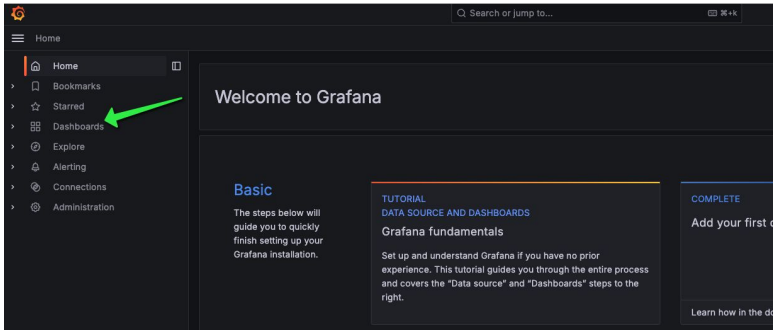
Call to action: Access the Grafana page

- In the browser open the new tab and go to
  - http://<vm ip>:3000
  - Connect as user **admin** with the password: **prom-operator**
- The grafana page will appear

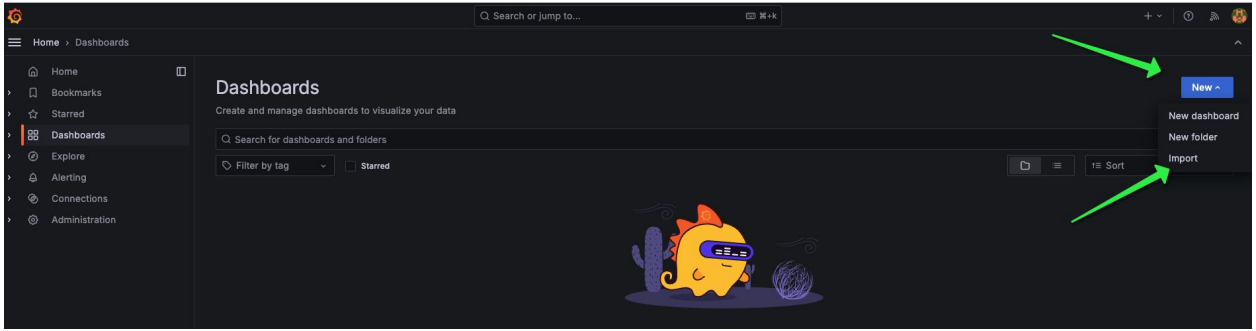


Call to action: Configure Grafana

- Go to Dashboards

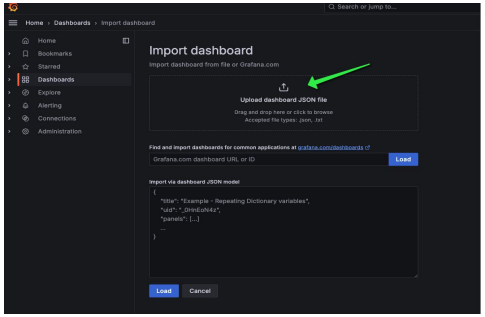


- Press "New", then "Import":

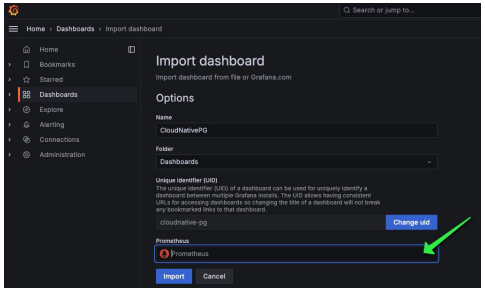


Call to action: Configure Grafana - continued

- Upload the Dashboard json file:



- Upload grafana-dashboard.json file
- Select Prometheus as the data source:



Call to action: Explore CNPG Dashboard- continued

- Explore the CNPG Dashboard:

What more?

(some additional features from EDB)

What we didn’t show you today ….

- PgBouncer (Pooler) integration
  - Create a PgBouncer deployment and automatically configure to the cluster.
- TLS
  - CNPG supports TLS/SSL for encrypted connections etc.
  - The cluster you’re running right now uses certificates for streaming replication
- RBAC
  - The operator uses a dedicated service account ‘cnpg-manager’ to interact with K8s
  - Of course, within the database, everything is as ‘normal’
- Auditing
  - PGAudit can be configured by changing the parameters .spec.postgresql.parameters.pgaudit.*
  - The operators automatically load PGAudit and runs “CREATE EXTENSION”

Advanced Security

Password policy management

DBA managed password profiles, compatible with Oracle profiles

Audit compliance

Track and analyze database activities and user connections

Virtual private databases

Fine grained access control limits user views

EDB/SQL protect

SQL firewall, screens queries for common attack profiles

Data redaction

Protect sensitive information for GDPR, PCI and HIPAA compliance

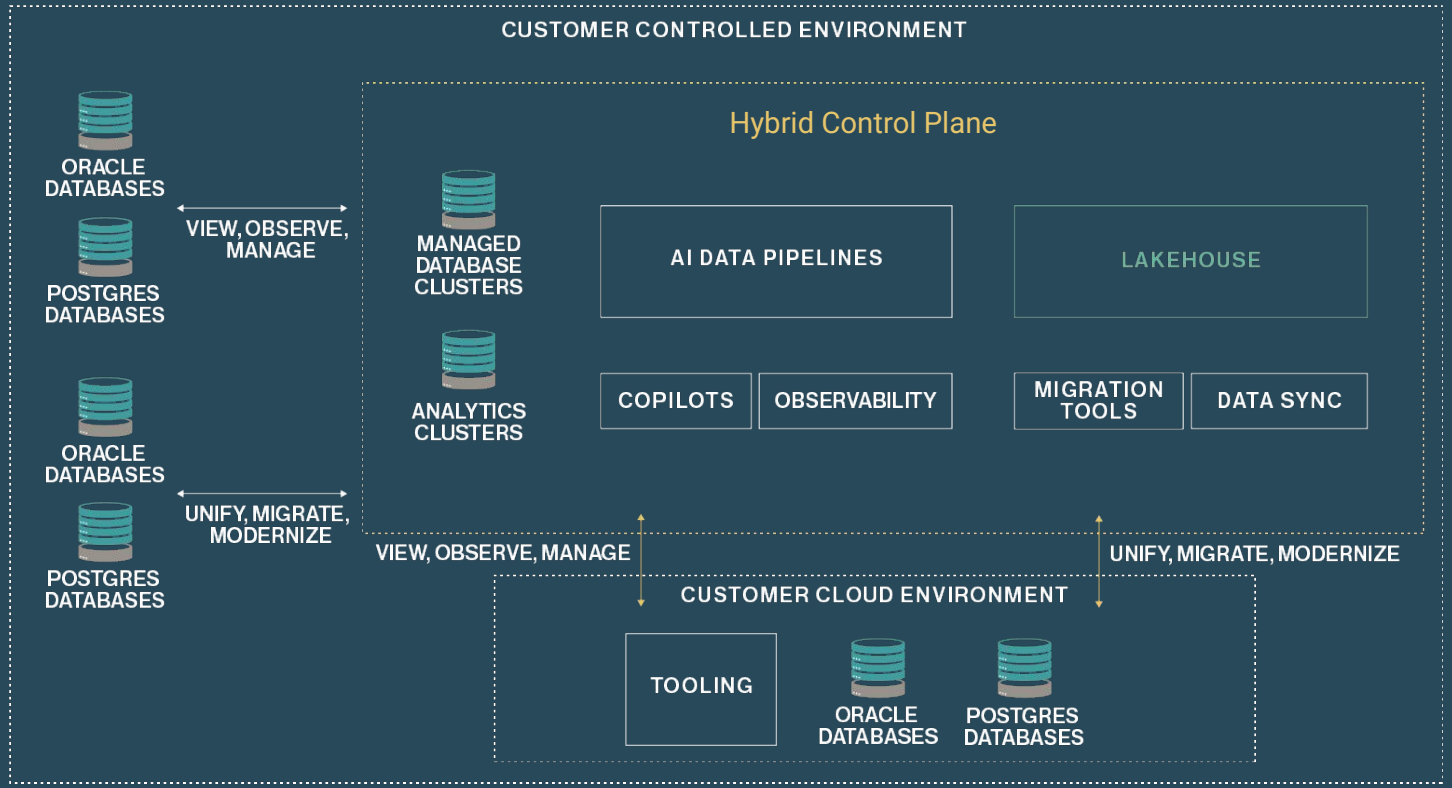
Code protection

Protects sensitive IP, algorithms or financial policies

Transparent Data Encryption (EDB-only features)

- Transparent Data Encryption (TDE) is a feature of EDB Postgres Advanced Server and EDB Postgres Extended Server that prevents unauthorized viewing of data in operating system files on the database server and on backup storage
- Data encryption and decryption is managed by the database and does not require application changes or updated client drivers
- EDB Postgres Advanced Server and EDB Postgres Extended Server provide hooks to key management that is external to the database allowing for simple passphrase encrypt/decrypt or integration with enterprise key management solutions, with initial support for:
  - Amazon AWS Key Management Service (KMS)
  - Google Cloud - Cloud Kay Management Service
  - Microsoft Azure Key Vault
  - HashiCorp Vault (KMIP Secrets Engine and Transit Secrets Engine)
  - Thales CipherTrust Manager
- Data will be unintelligible for unauthorized users if stolen or misplaced

Hybrid Control Plane at a glance



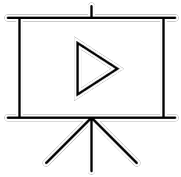
EDB Postgres® AI (PG:AI)

What if you could do all this, in a single install?

- **Database Observability**
  - Monitor all your Postgres databases using internal API, Prometheus and Grafana
- **Database migrations**
  - Migrate Oracle database directly into the platform
- **Database Provisioning**
  - Run Postgres databases as simple as point and click
- **Analytics Accelerator**
  - 30x performance on analytics queries compared to Postgres
  - Query data through Postgres (include both live-data, analytics and Delta Table content)
  - Automatically manage live data -> analytics storage -> delta table
- **AI Accelerator**
  - Manage and deploy AI models
  - Build low-code AI bots (and more advanced ones, of course)
  - Automatic data management of files and databases for AI Knowledge bases

EDB Postgres® AI

LIVE DEMO



BACKUP SLIDES

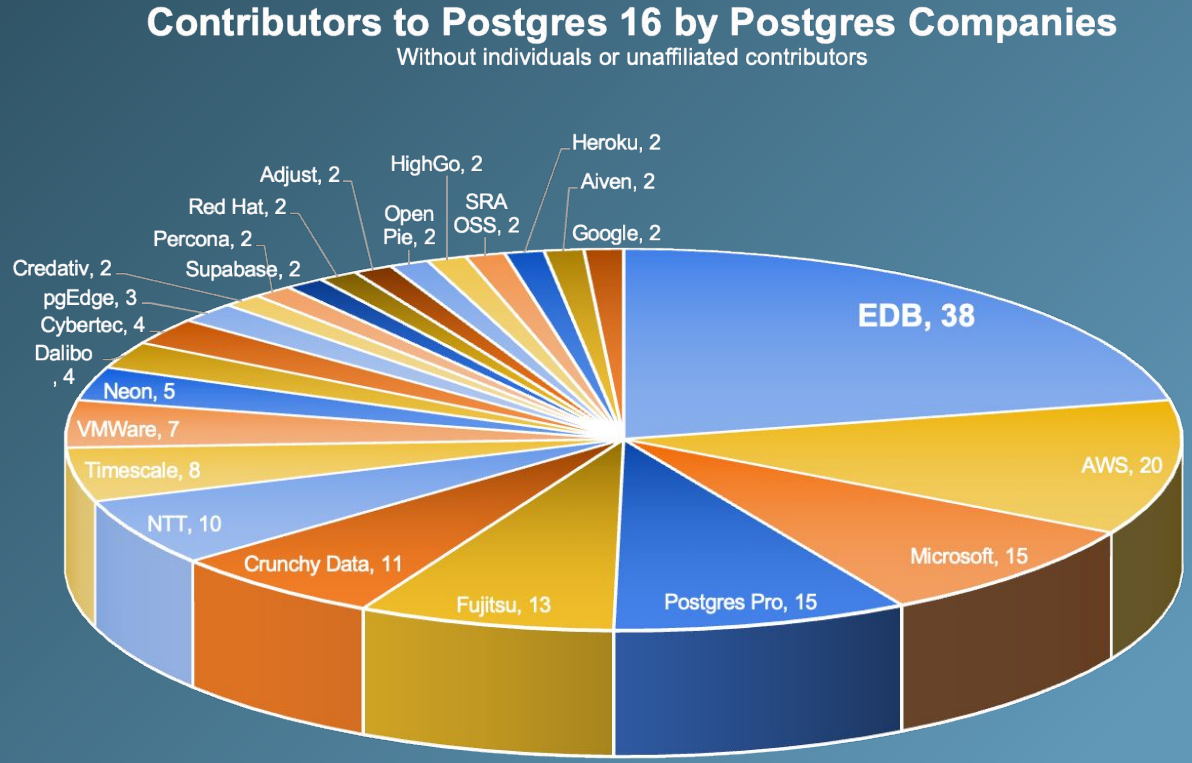
EDB
Postgres® for the AI Generation

20+ years of Postgres innovation & adoption

- Number one contributor to Postgres, fastest-growing and most loved Database in the world
  - 2 Core Team members, 7 Committers, 9 Major Contributors, 10 Contributors, #1 site for desktop downloads
- Over 700 employees in more than 30 countries
- EDB Postgres AI
  - The industry's first platform that can be deployed as cloud, software or physical appliance
  - Secure, compliant and enterprise grade performance guaranteed

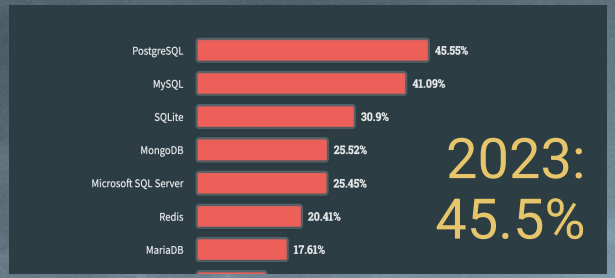
Large Developer Community

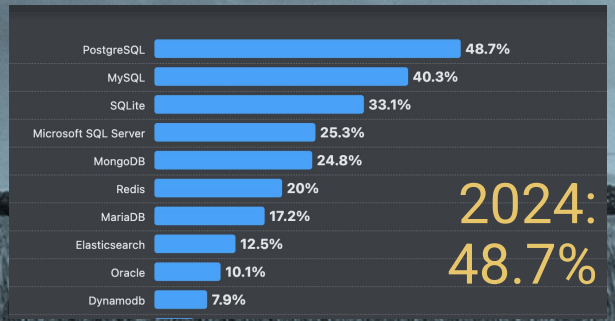
- 407 + contributors to Postgres 16
- 111 companies actively contributing to Postgres 16



EDB
Postgres for the AI Generation

Postgres has won the database race

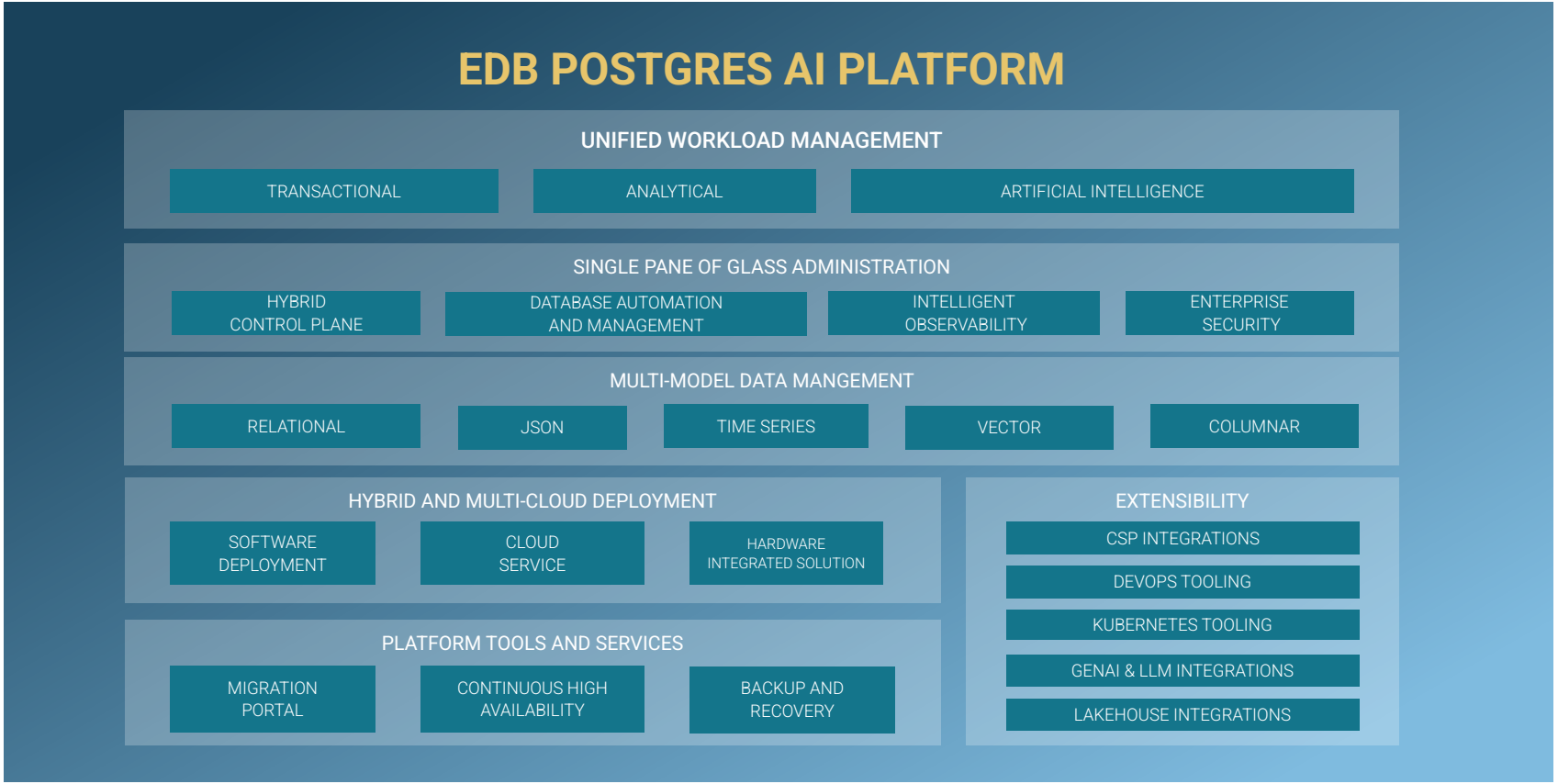




Stack Overflow Survey 2023/2024

LEADING ENTERPRISES TRUST EDB

EDB POSTGRES AI PLATFORM



CNPG Operator: Reference Architecture and functionalities

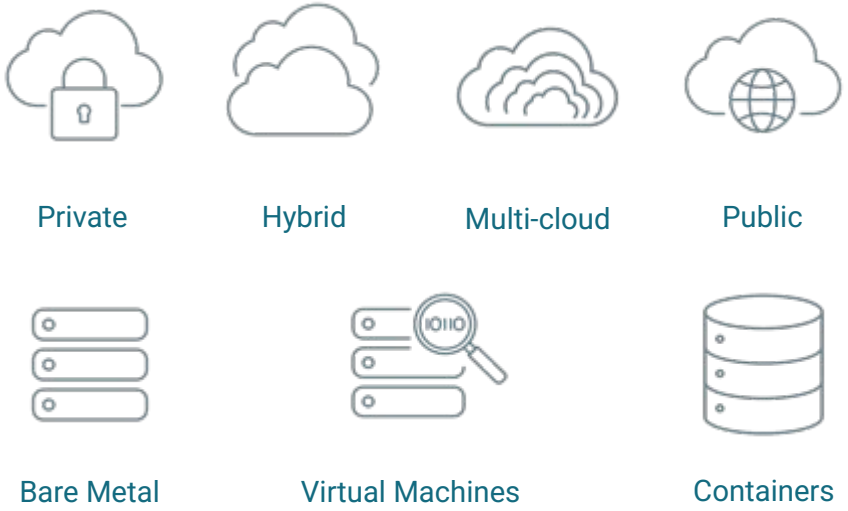
Kubernetes timeline

- 2014, June: Google open sources Kubernetes
- 2015, July: Version 1.0 is released
- 2015, July: Google and Linux Foundation start the CNCF
- 2016, November: The operator pattern is introduced in a blog post
- 2018, August: The Community takes the lead
- 2019, April: Version 1.14 introduces Local Persistent Volumes
- 2019, August: EDB team starts the Kubernetes initiative
- 2020, June: we publish this blog about benchmarking local PVs on bare metal
- 2020, June: Data on Kubernetes Community founded
- 2021, February: EDB Cloud Native Postgres (CNP) 1.0 released
- 2022, May: EDB donates CNP and open sources it under CloudNativePG
- 2025, January: CloudNativePG was recognized as an official #CNCF project

Enabling the same PostgreSQL everywhere

From self-managed to fully managed DBaaS in the Cloud

- Same applications
- Faster innovation
- Performance and scalability
- Stability, security and control
- Seamless integration
- Obsolescence



A kubernetes operator for Postgres

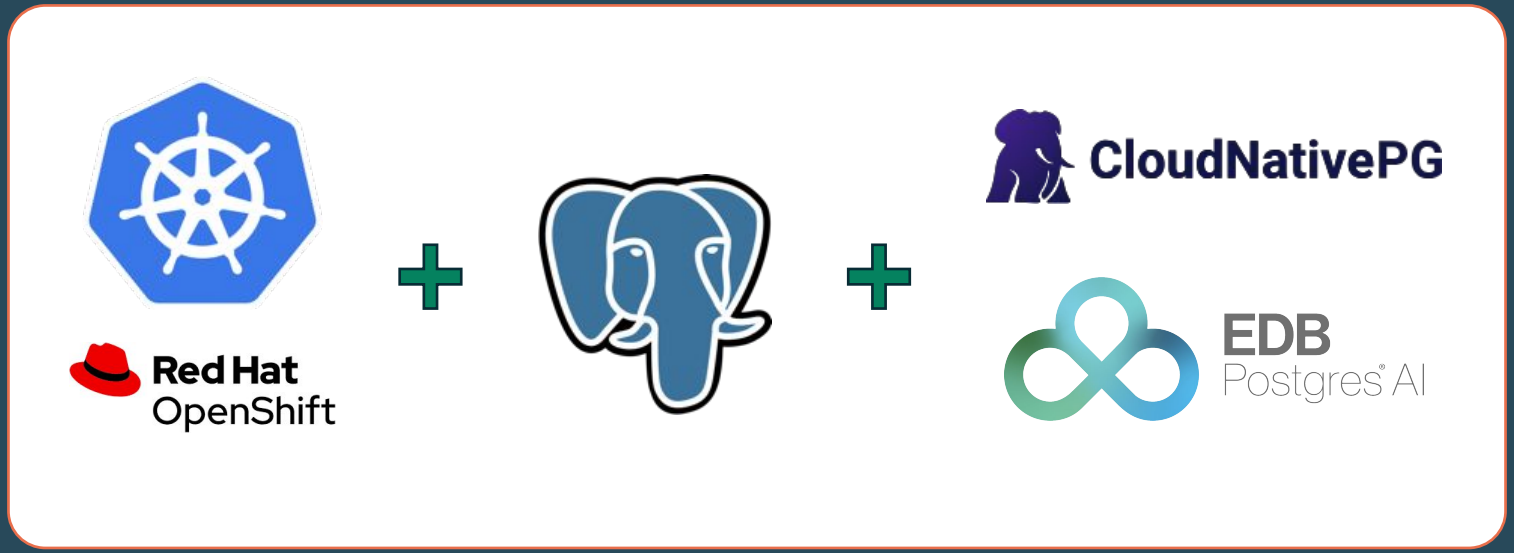
Kubernetes adoption is rising and it is already the de facto standard orchestration tool

PostgreSQL clusters “management the kubernetes way” enables many cloud native usage patterns, e.g. spinning up, disposable clusters during tests, one cluster per microservice and one database per cluster

CNPG tries to encode years of experience managing PostgreSQL clusters into an Operator which should automate all the known tasks a user could be willing to do

Our PostgreSQL operator must simulate the work of a DBA

Win Technology



Autopilot

It automates the steps that a human operator would do to deploy and to manage a Postgres database inside Kubernetes, including automated failover.

Security

CloudNativePG is secured by default.

It doesn’t rely on statefulsets and uses its own way to manage persistent volume claims where the PGDATA is stored.

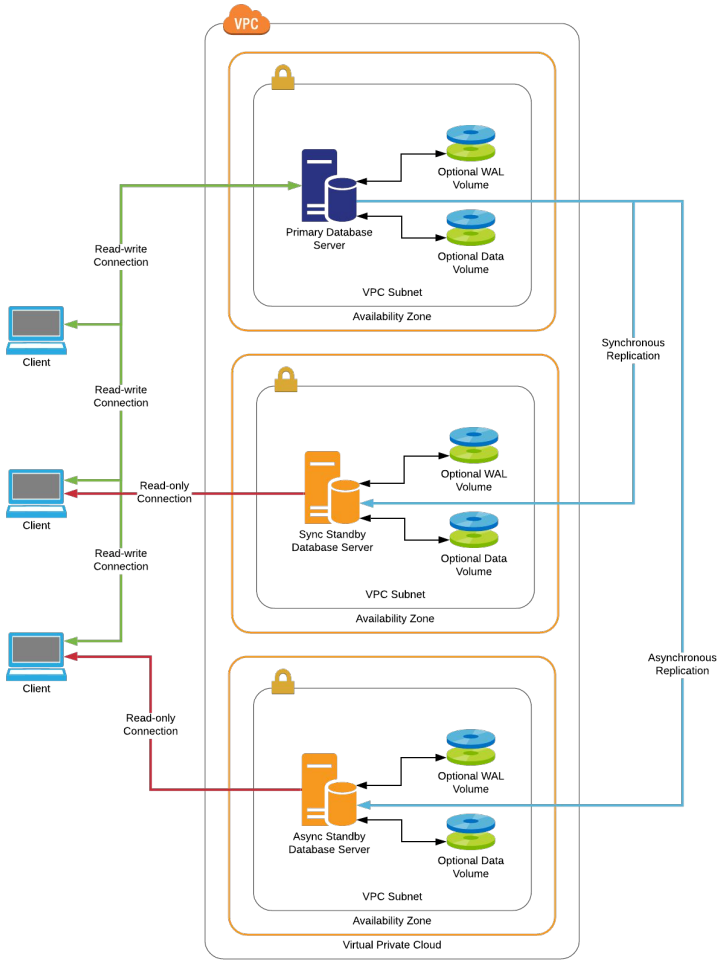
Data persistence

Designed for Kubernetes

It’s entirely declarative, and directly integrates with the Kubernetes API server to update the state of the cluster — for this reason, it does not require an external failover management tool.

Imperative vs Declarative

- Create and configure VMs
- Create a PostgreSQL 13 instance
- Configure for replication
- Clone a second one
- Set it as a replica
- Clone a third one
- Set it as a replica
- Configure networking
- Configure security
- etc.



Convention over configuration

Declarative - simple to install, simple to maintain

There's a PostgreSQL 17 cluster with 2 replicas:

```
apiVersion: postgresql.k8s.enterprisedb.io/v1
kind: Cluster
metadata:
  name: myapp-db
spec:
  instances: 3
  imageName: quay.io/enterprisedb/postgresql:17

  storage:
    size: 10Gi
```

Features

Deployment	Administration	Backup & Recovery	Monitoring	Security	High Availability
Kubernetes operator	Single node	Backup	Prometheus	TDE	Switchover
Kubernetes plugin	Cluster (Multi node)	Recovery	Grafana dashboards	Certificates	Failover
EDB Postgres (EPAS)	PostgreSQL configuration	PITR	Postgres Enterprise Manager	Data redaction	Scale out / scale down
PostGIS	Pooling	Volume Snapshots	Logging	Password management	Minor / Major updates

Decision-making for choosing the deployment platform

Advantage of deploying Postgres Databases in Kubernetes

01 | Automation & Orchestration

- Self-healing
- Automated scaling
- Rolling updates

02 | Self-healing

- Best resource utilization
- Dynamic Resource allocation

03 | Rolling updates

- Cloud-agnostic
- Consistent deployment

04 | Service discovery & networking

- Built-it service discovery
- Load Balancing

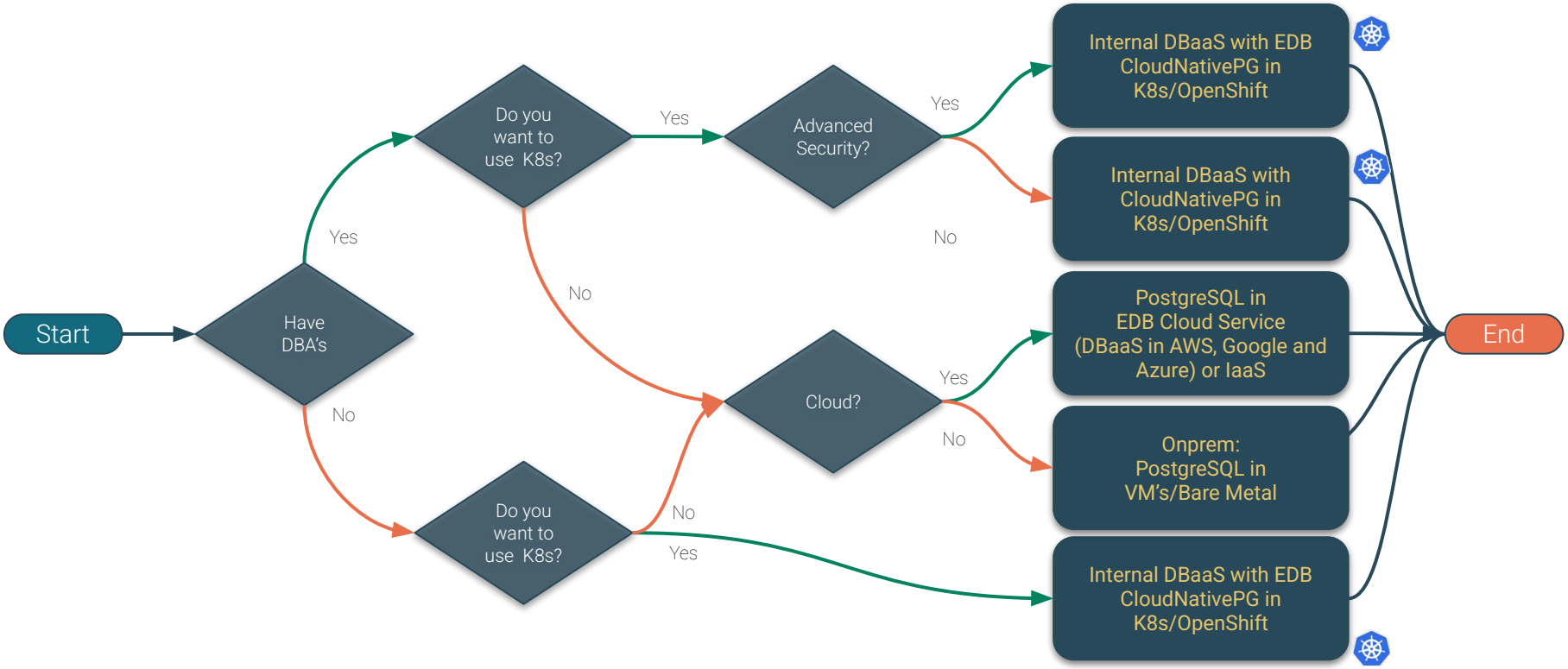
05 | Automated backups and disaster recovery

- Automated backups
- Multi-region failover

06 | Security & access control

- RBAC
- Secret management

Decision-making for choosing the deployment platform



Architectures

When to choose Kubernetes over VMs?

01 | Cloud Native Applications that already run in Kubernetes

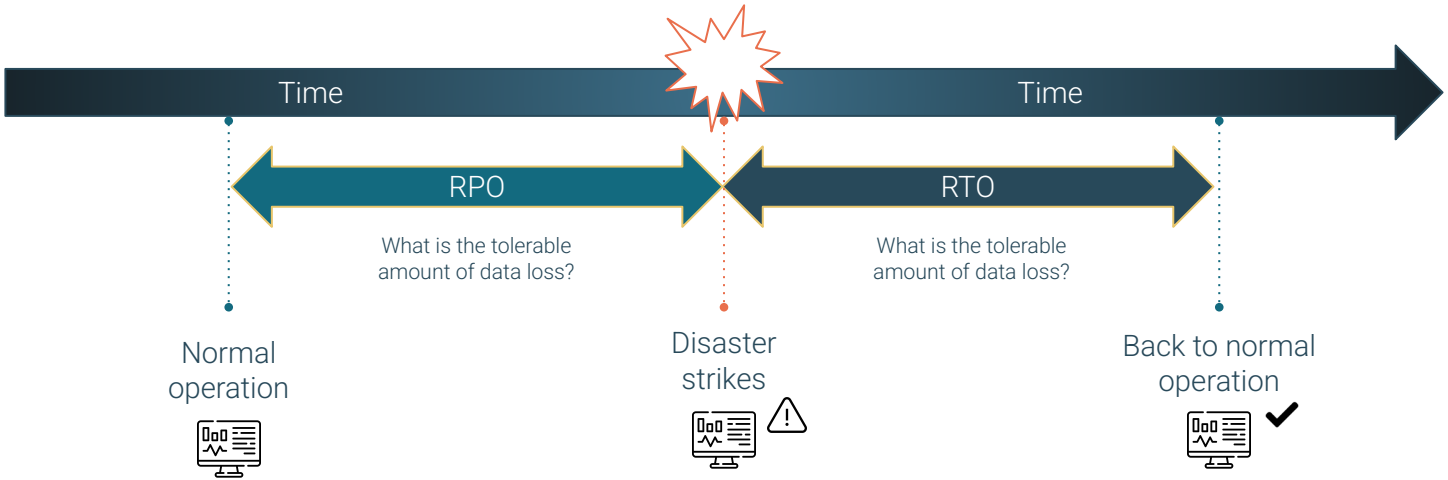
02 | Scalable, replicated databases

03 | Applications requiring automated failover and self-healing

04 | Teams skilled in Kubernetes who want a unified infrastructure

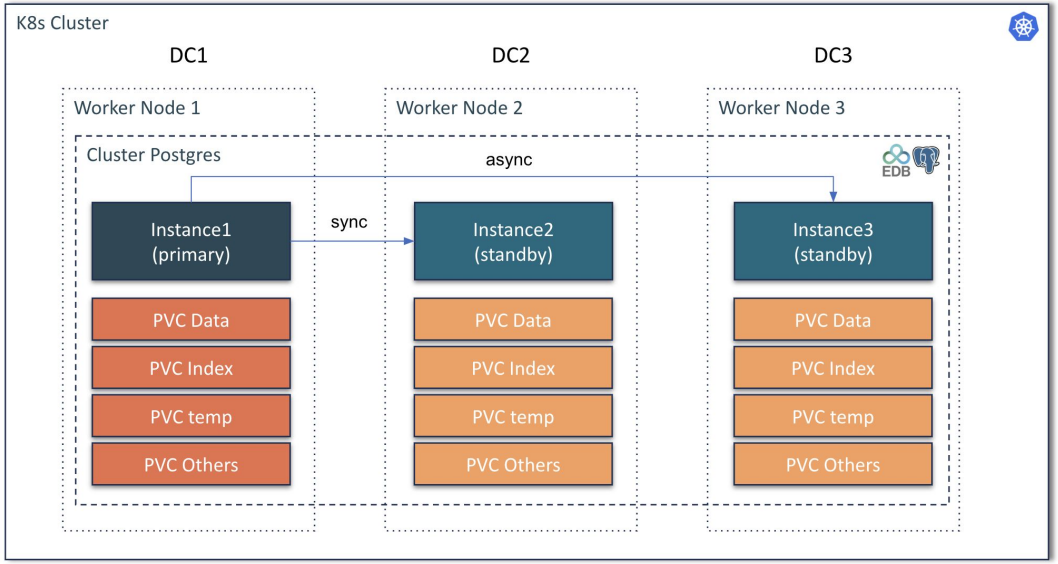
Concepts

- Recovery Point Objective (**RPO**) and Recovery Time Objective (**RTO**) are key concepts in disaster recovery and business continuity planning, particularly related to data loss and system downtime.



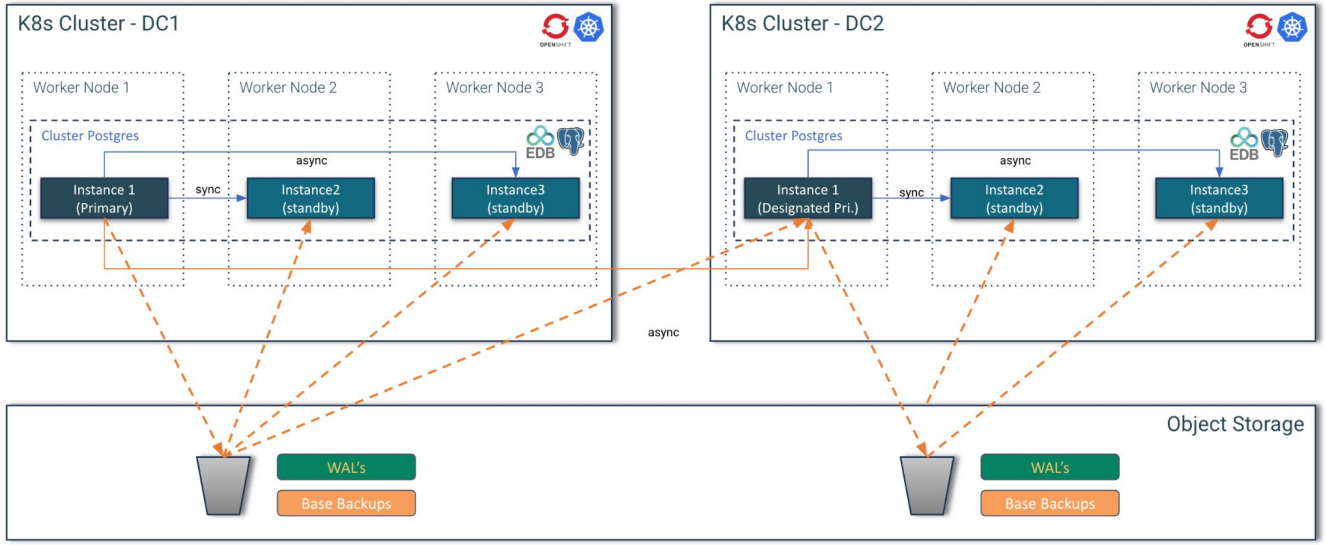
Red Hat Recommendation

Red Hat recommend stretched clusters ONLY when latencies don't exceed 5 milliseconds (ms) round-trip time (RTT) between the nodes in different locations, with a maximum RTT of 10 ms.

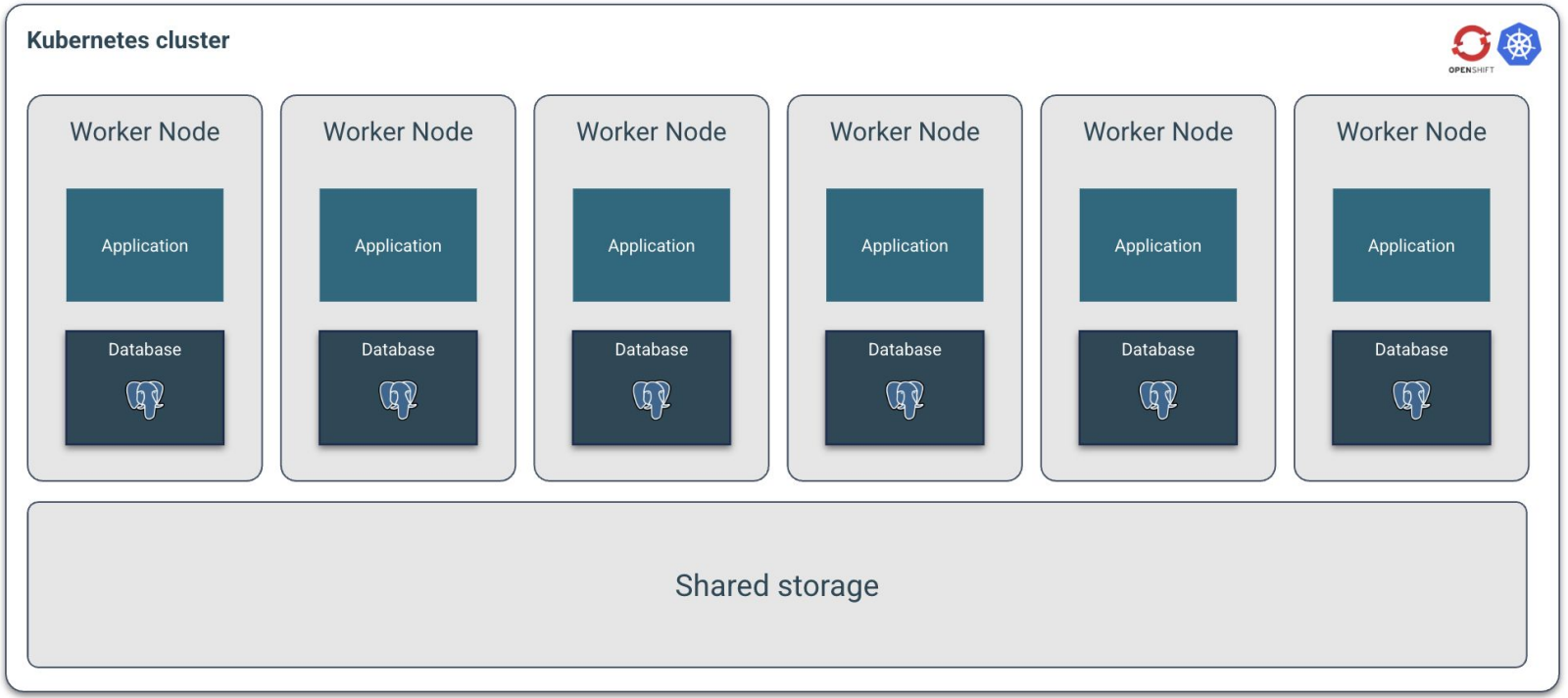


Two separate single data center Kubernetes clusters

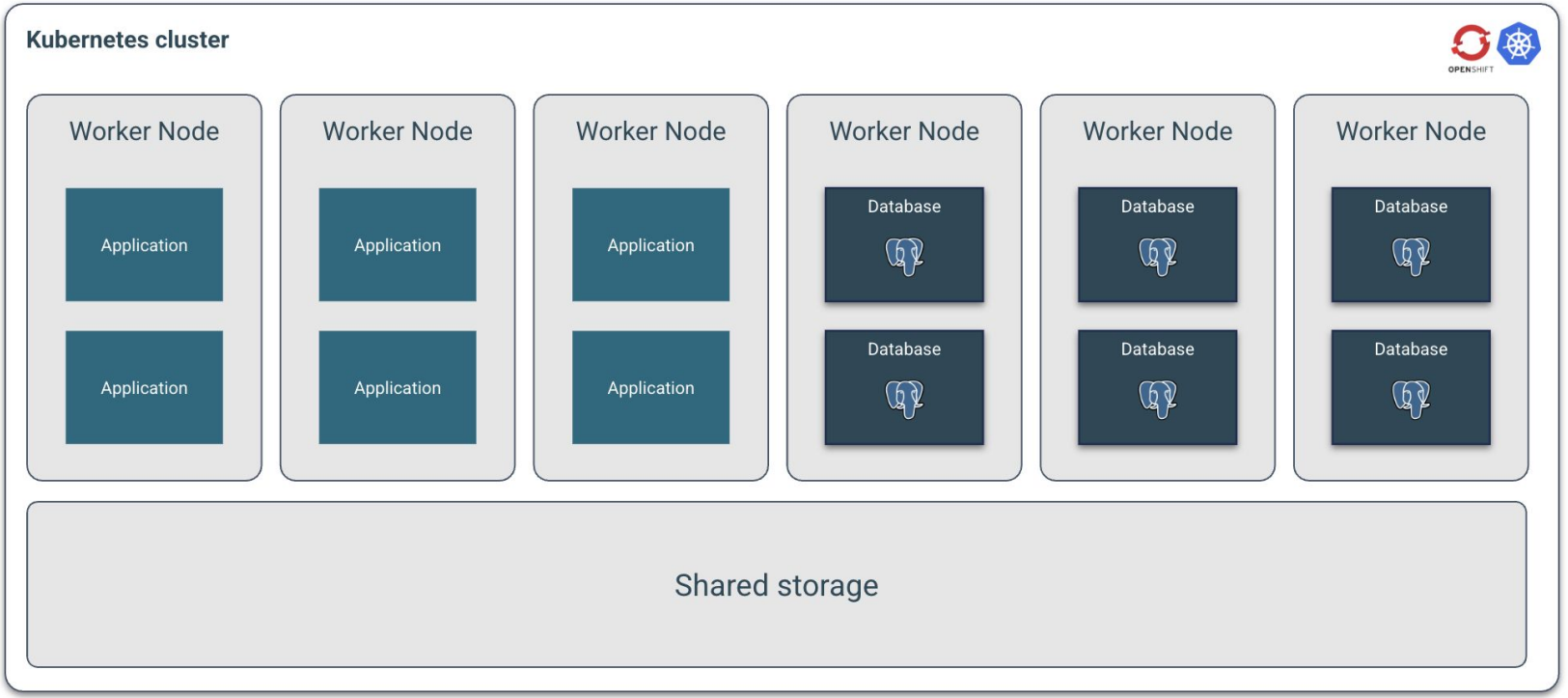
In case you cannot go beyond two data centers and you end up with two separate Kubernetes clusters, don't despair.



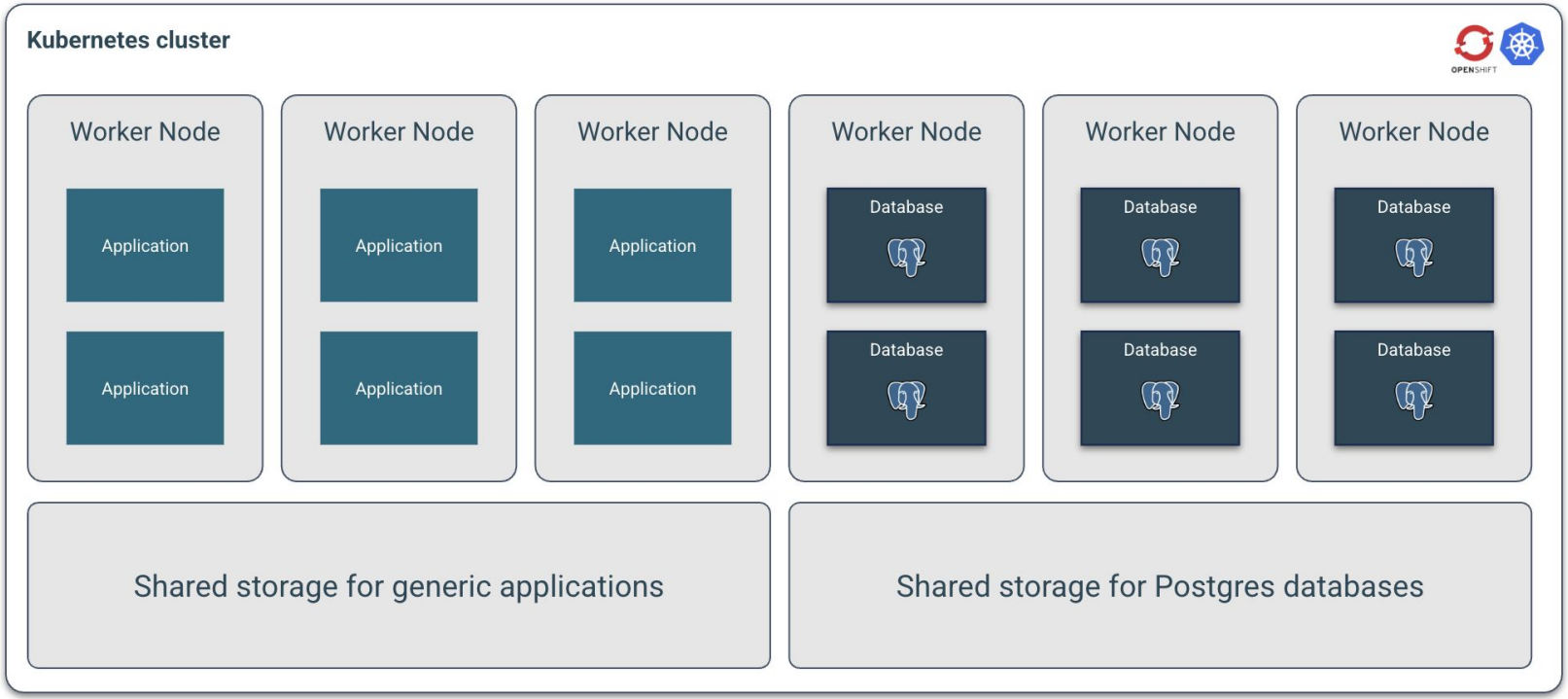
Shared workload, shared storage



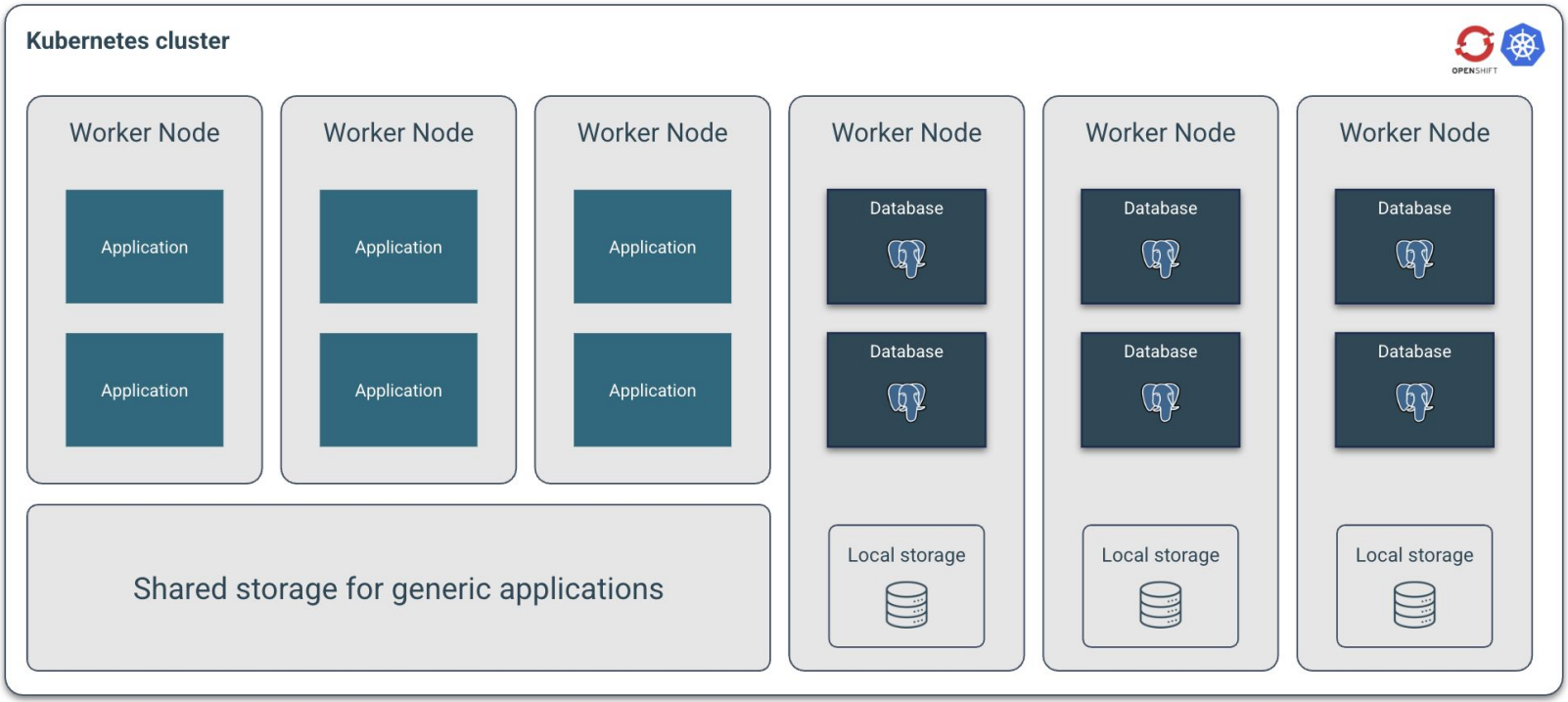
Shared workload, shared storage



Shared workload, shared storage

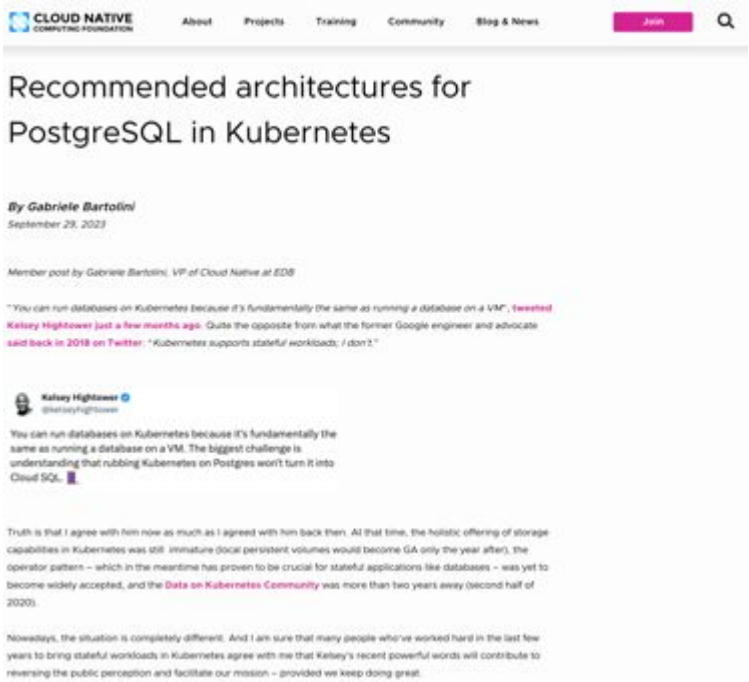


Shared workloads, local storage



Recommended architectures

https://www.cncf.io/blog/2023/09/29/recommended-architectures-for-postgresql-in-kubernetes/



CLOUD NATIVE COMPUTING FOUNDATION

About Projects Training Community Blog & News Join

Recommended architectures for PostgreSQL in Kubernetes

By Gabriele Bartolini
September 29, 2023

Member post by Gabriele Bartolini, VP of Cloud Native at EDB

"*You can run databases on Kubernetes because it's fundamentally the same as running a database on a VM*", tweeted Kelsey Hightower just a few months ago. Quite the opposite from what the former Google engineer and advocate said back in 2018 on Twitter: "*Kubernetes supports stateful workloads; I don't.*"

Kelsey Hightower
@kelseyhightower

You can run databases on Kubernetes because it's fundamentally the same as running a database on a VM. The biggest challenge is understanding that rubbing Kubernetes on Postgres won't turn it into Cloud SQL.

Truth is that I agree with him now as much as I agreed with him back then. At that time, the holistic offering of storage capabilities in Kubernetes was still immature (local persistent volumes would become GA only the year after), the operator pattern – which in the meantime has proven to be crucial for stateful applications like databases – was yet to become widely accepted, and the Data on Kubernetes Community was more than two years away (second half of 2020).

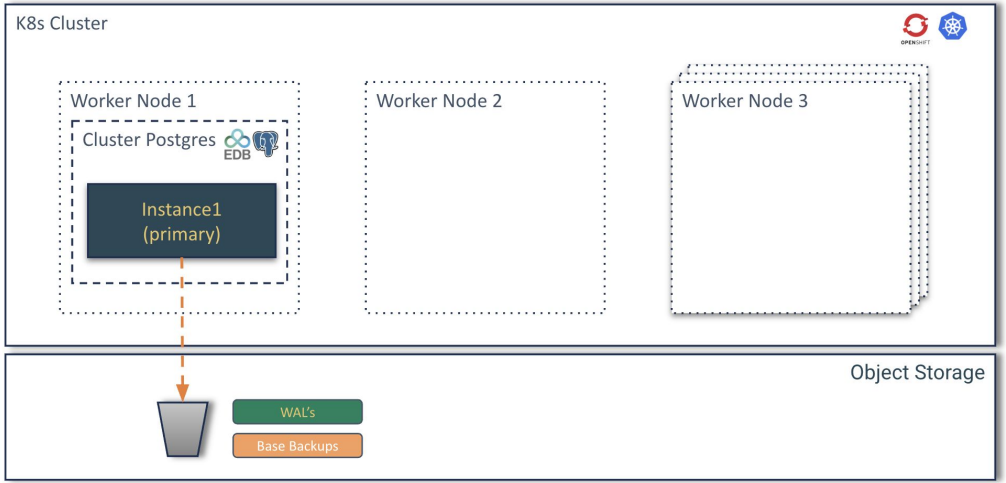
Nowadays, the situation is completely different. And I am sure that many people who've worked hard in the last few years to bring stateful workloads in Kubernetes agree with me that Kelsey's recent powerful words will contribute to reversing the public perception and facilitate our mission – provided we keep doing great.

Use Cases

Use case 1 architecture

A single database is the simplest setup, involving one instance of a database server.

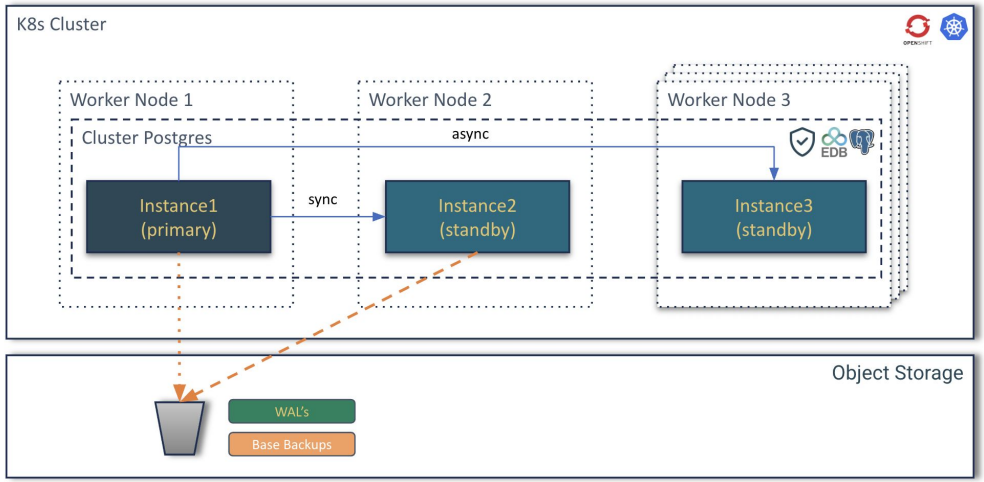
- Development and testing environments
- Small applications with low traffic
- Non-critical data analysis
- Applications with high tolerance for downtime
- Cost-sensitive projects



Use case 2 architecture

An HA database setup aims to minimize downtime by having redundant components. If one component fails, another takes over automatically or with minimal intervention. This usually involves techniques like clustering, replication, or mirroring within the same data center or availability zone.

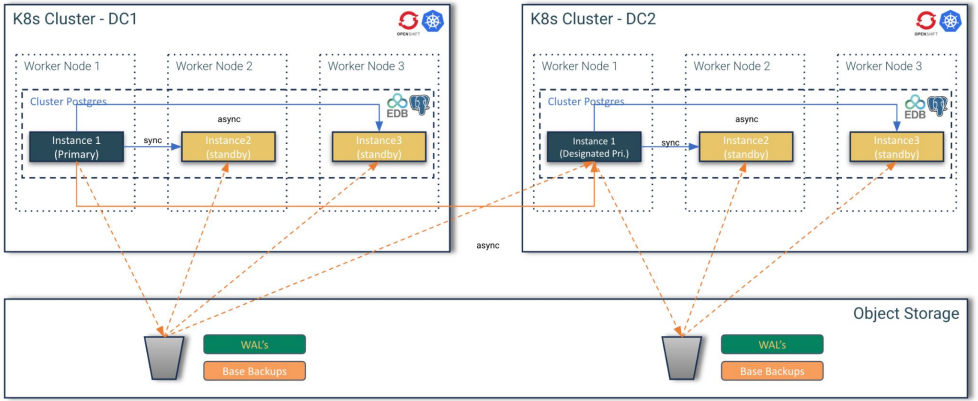
- Business critical Applications
- Applications with stringent SLAs
- Real-time systems
- Improving user experience
- Minimizing planned downtime



Use case 3 architecture

A DR database setup focuses on protecting data and ensuring business continuity in the event of a large-scale disaster affecting an entire data center or region (e.g., natural disasters, power outages, cyberattacks). This typically involves replicating data to a geographically separate location.

- Regulatory compliance
- Protecting against catastrophic data loss
- Ensuring business continuity for mission-critical systems



Use case 3 architecture