



**EDB**

Postgres® for the AI Generation



**VSHN**

# Postgres on Kubernetes Workshop

5 March 2025 | VSHN Offices Zurich

Borys Neselovskyi  
Senior Sales Engineer, EDB

Piotr Kolodziej  
Sales Engineer, EDB

# Agenda

Start	End	Session
13:45	14:00	Registration
14:00	14:30	Introduction to CloudNativePG and EDB
14:30	15:00	CNPG Operator Reference Architecture
15:00	15:30	Functionalities for CNPG
15:30	17:00	Demo
17:00	18:00	Pizza & Drinks

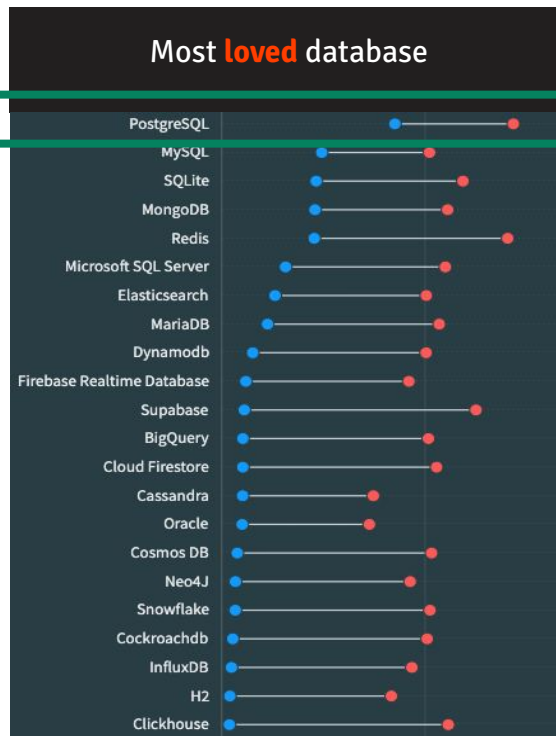
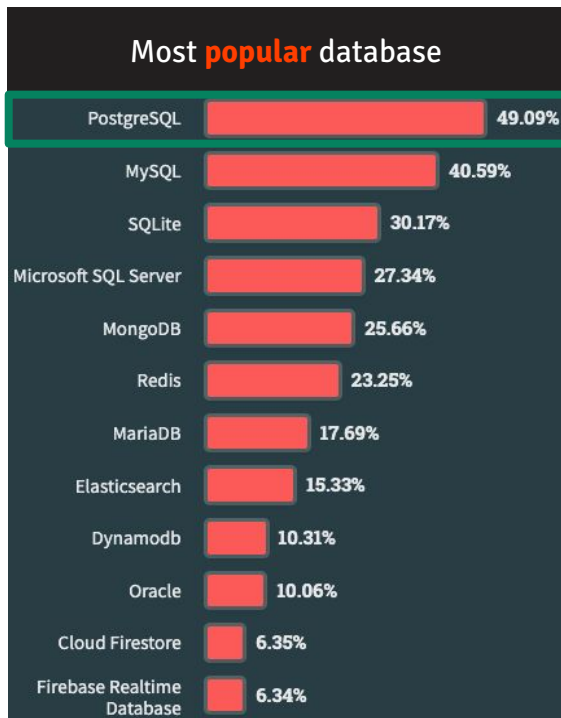


# Introduction to CloudNativePG and EDB



# PostgreSQL Is Winning

The most admired, desired & used database - Source: Stack Overflow Developer Survey, 2023



# EDB POSTGRES AI PLATFORM

## UNIFIED WORKLOAD MANAGEMENT

TRANSACTIONAL

ANALYTICAL

ARTIFICIAL INTELLIGENCE

## SINGLE PANE OF GLASS ADMINISTRATION

HYBRID  
CONTROL PLANE

DATABASE AUTOMATION  
AND MANAGEMENT

INTELLIGENT  
OBSERVABILITY

ENTERPRISE  
SECURITY

## MULTI-MODEL DATA MANGEMENT

RELATIONAL

JSON

TIME SERIES

VECTOR

COLUMNAR

## HYBRID AND MULTI-CLOUD DEPLOYMENT

SOFTWARE  
DEPLOYMENT

CLOUD  
SERVICE

HARDWARE  
INTEGRATED SOLUTION

## PLATFORM TOOLS AND SERVICES

MIGRATION  
PORTAL

CONTINUOUS HIGH  
AVAILABILITY

BACKUP AND  
RECOVERY

## EXTENSIBILITY

CSP INTEGRATIONS

DEVOPS TOOLING

KUBERNETES TOOLING

GENAI & LLM INTEGRATIONS

LAKEHOUSE INTEGRATIONS

# 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



# CloudNativePG/EDB Postgres for Kubernetes

## CloudNativePG



- Kubernetes operator for PostgreSQL
- “Level 5”, Production ready
- Day 1 & 2 operations of a Postgres database
- Open source (May 2022)
  - Originally created by EDB
  - Apache License 2.0
  - Vendor neutral openly governed
- Extends the K8s controller
  - Status of the `Cluster`
  - “no Patroni, No statefulsets”
- Immutable application containers
- Fully declarative

## EDB CloudNativePG

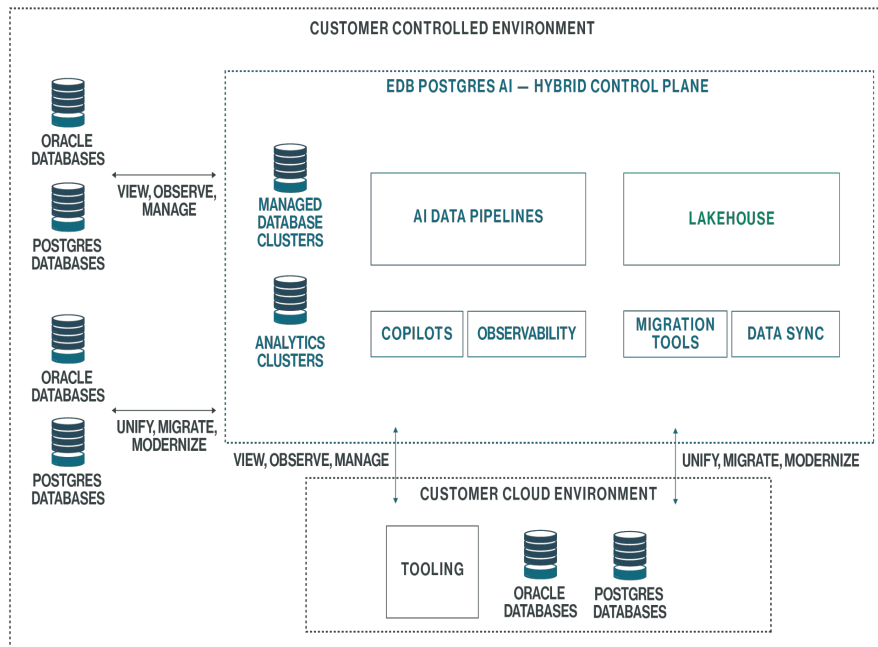


- All Features of CloudNativePG
- +
- Provides Long Term Support
- Access to EDB Postgres Advanced (TDE + Oracle Compatibility layer + Security features)
- Red Hat OpenShift compatibility
- Kubernetes level backup integration
  - Generic external backup interface
- [Hybrid Control Plane \(Preview\)](#) - centralized management and automation solution, built on Kubernetes, for EDB Postgres AI

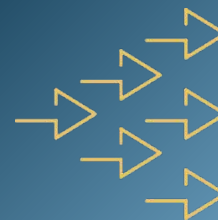


## #4: Hybrid Control Plane – Observability and automation

*Centralized management, observability, and automation means customers can do more, with less, and innovate faster*



Operate  
Efficiently

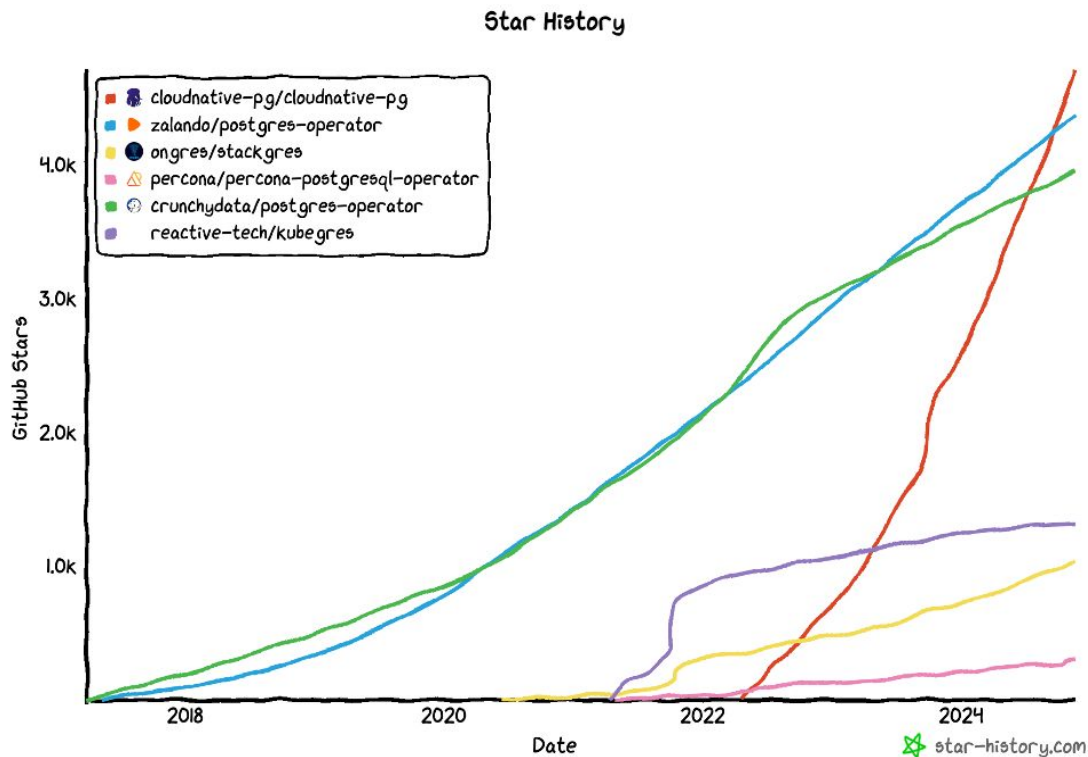


Innovate  
Faster





# The world's most popular Postgres operator for Kubernetes



CLOUD NATIVE  
COMPUTING FOUNDATION

SANDBOX PROJECTS



Run PostgreSQL.  
The Kubernetes way.

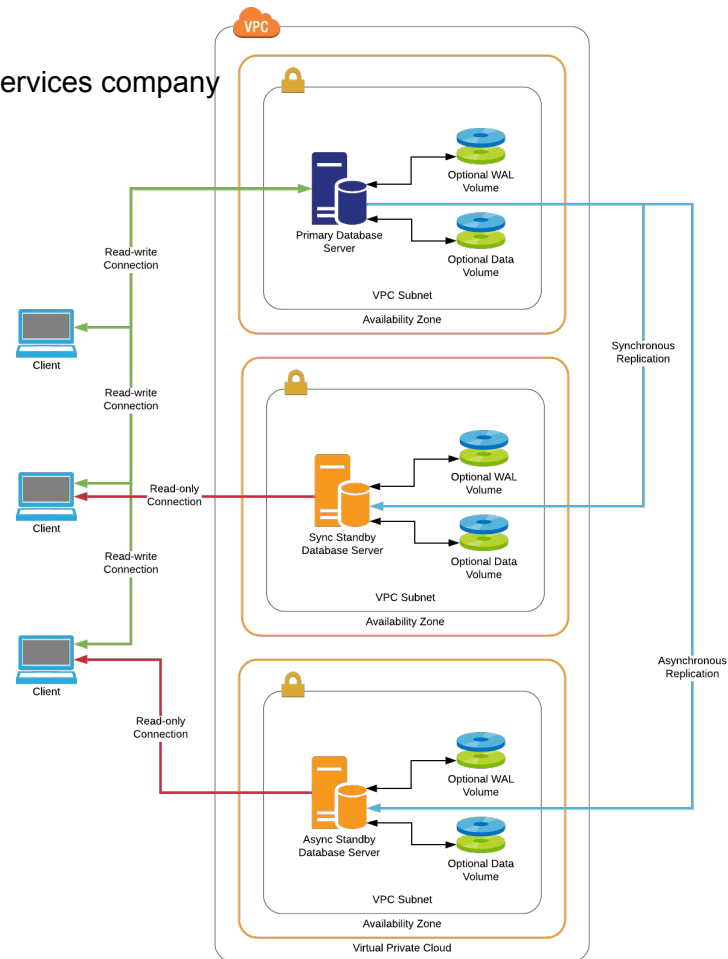
Gabriele Bartolini (He/Him) - 1st  
VP of Cloud Native at EDB | DoK Ambassador | PostgreSQL

72 Million  
downloads  
and counting

# Imperative vs Declarative

Built over 20 years, EDB is the world's largest software, support, and services company focused exclusively on PostgreSQL

- 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 16 cluster with 2 replicas:

```
apiVersion: postgresql.k8s.enterisedb.io/v1
kind: Cluster
metadata:
  name: myapp-db
spec:
  instances: 3
  imageName: quay.io/enterisedb/postgresql:16.2

  storage:
    size: 10Gi
```



# CNPG Operator Reference Architecture



“The same as running a  
database on a VM”



*I would add: "... provided **you** ..."*

Know PostgreSQL

Know Kubernetes

Have a good operator like CloudNativePG

**You** = You organization, made up of one or more multidisciplinary teams



# #2 - The right architecture for Kubernetes



# Kubernetes architectural concepts

- A Kubernetes Cluster (**k-cluster**)
- Availability zones (**AZ**)- also known as failure zones or data centers
  - Connected by redundant, low-latency, private network connectivity
  - At least 3 per k-cluster
- Kubernetes control plane to be distributed across the AZ
- Kubernetes worker nodes in each AZ running applications (workloads)
- Normally:
  - **1 k-cluster = 1 region with 3+ AZ**



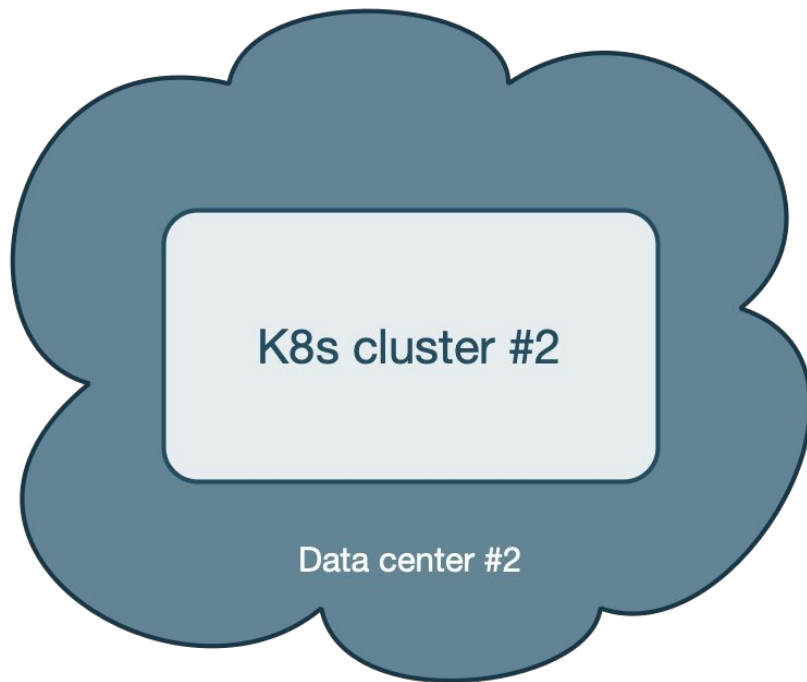
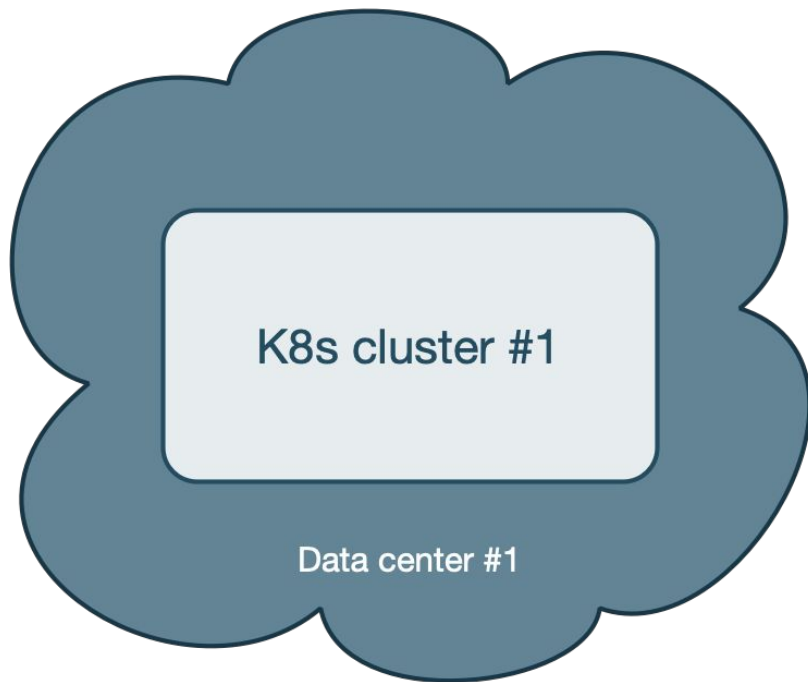


# 1 k-cluster = 1 region with 3+ AZ

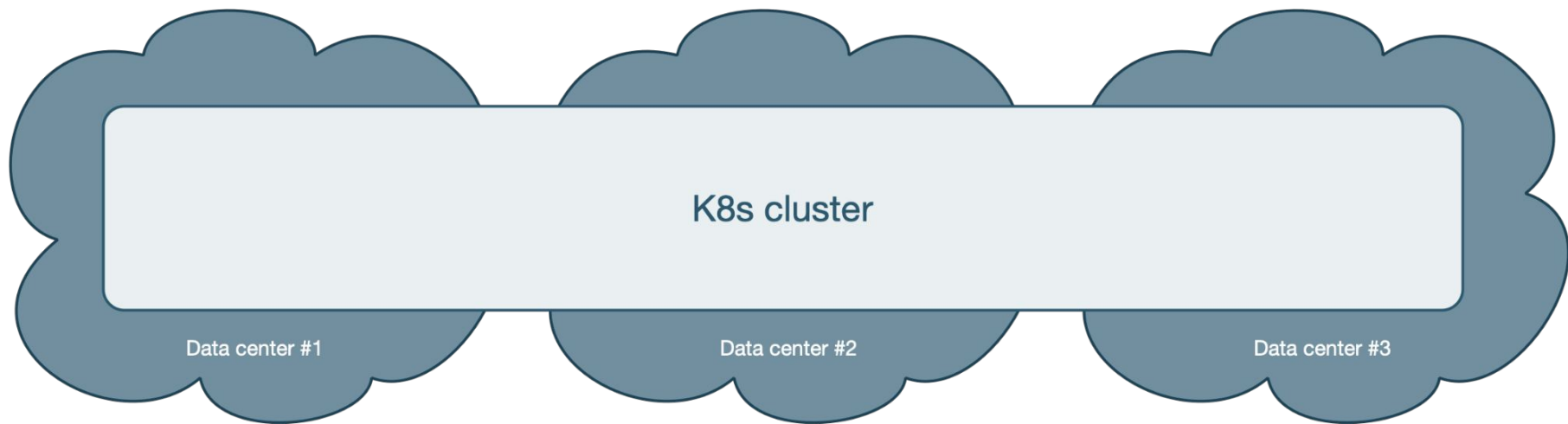
- Taken for granted if you know Kubernetes
- All major public cloud providers offering managed K8s services have 3+ AZ
- What about on-premise deployments?
  - You need to plan in advance
  - Stay away from the “2 data center in a region” setup typical of “Lift-and-Shift” exercises
    - Often results in 2 separate Kubernetes clusters
    - Severely impacts the benefits of Kubernetes, particularly self-healing
    - Shifts maintenance and procedural complexity up to the application level



No!



Yes!



# Yes! Yes! Yes!



# Synchronizing the state



# 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



# The right storage for you



# 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





# CloudNativePG

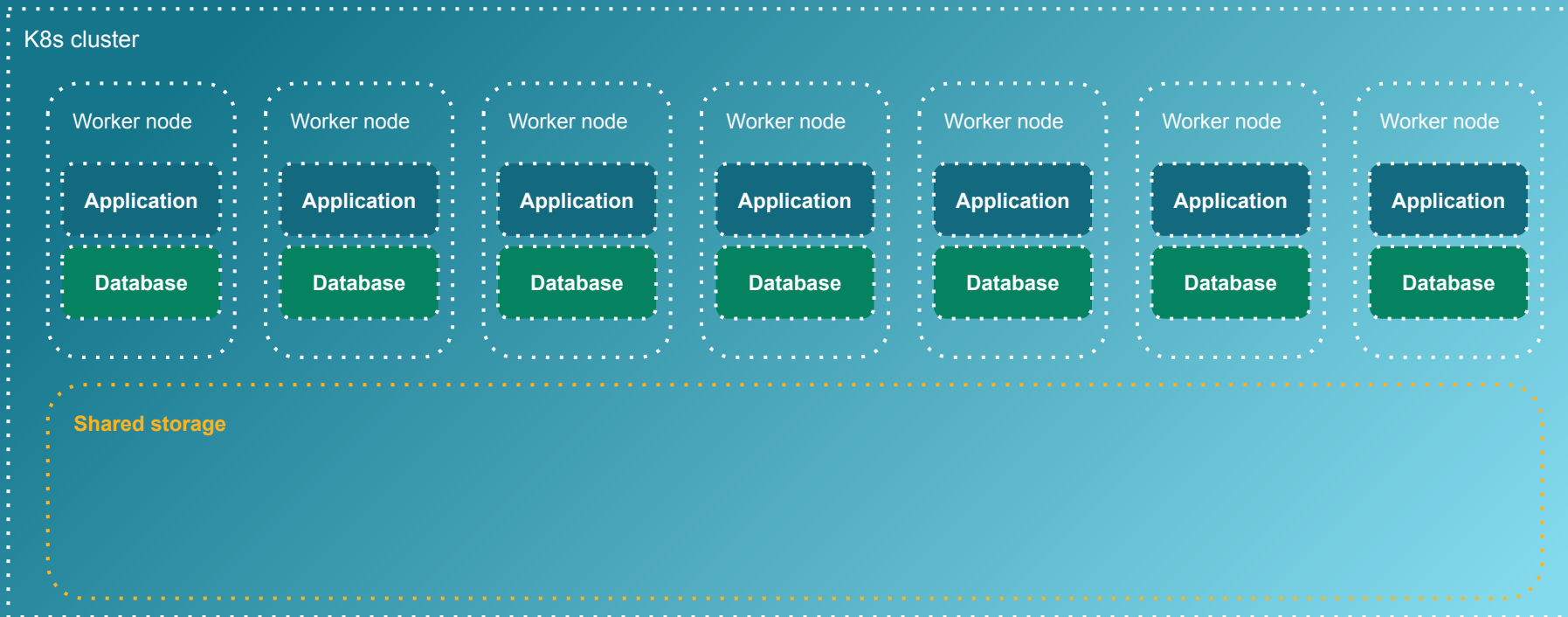
- Entirely declarative!
- Affinity section in the `Cluster` specification
  - pod affinity/anti-affinity
  - node selectors
  - tolerations against taints placed on nodes



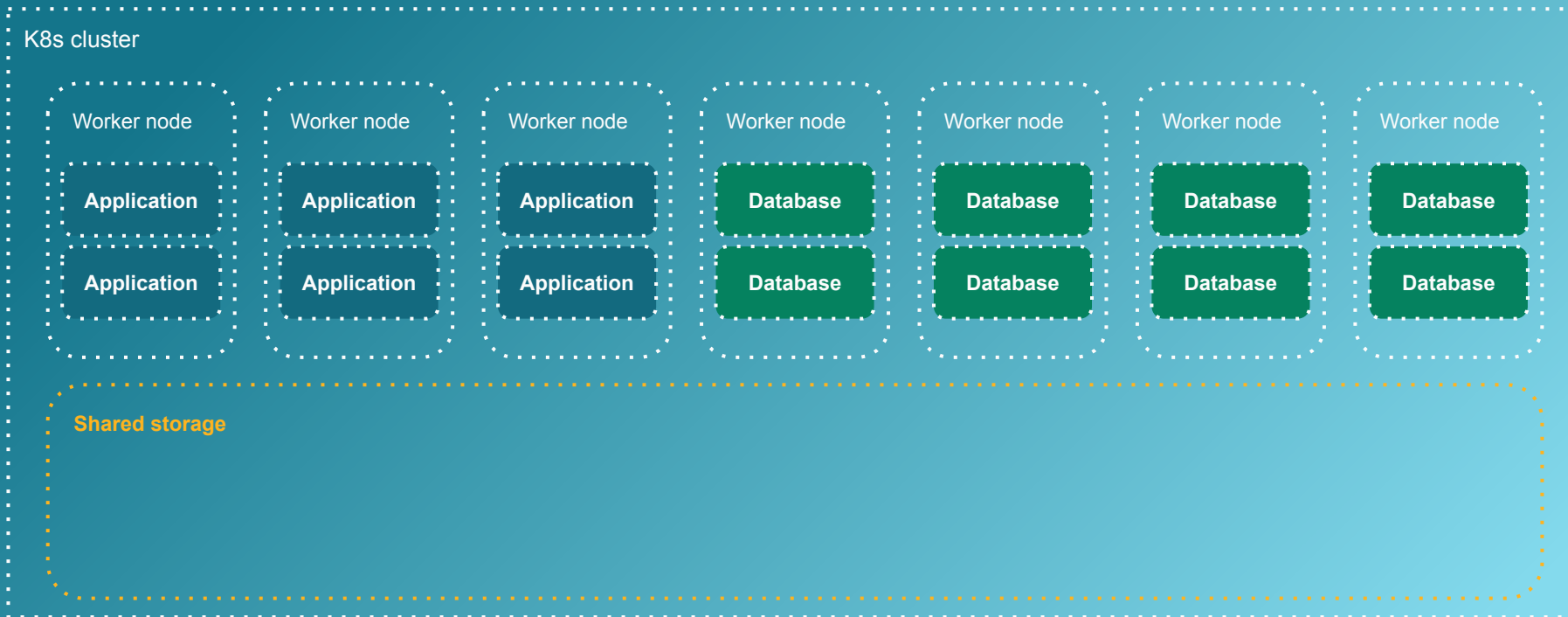
# The right architecture for Database



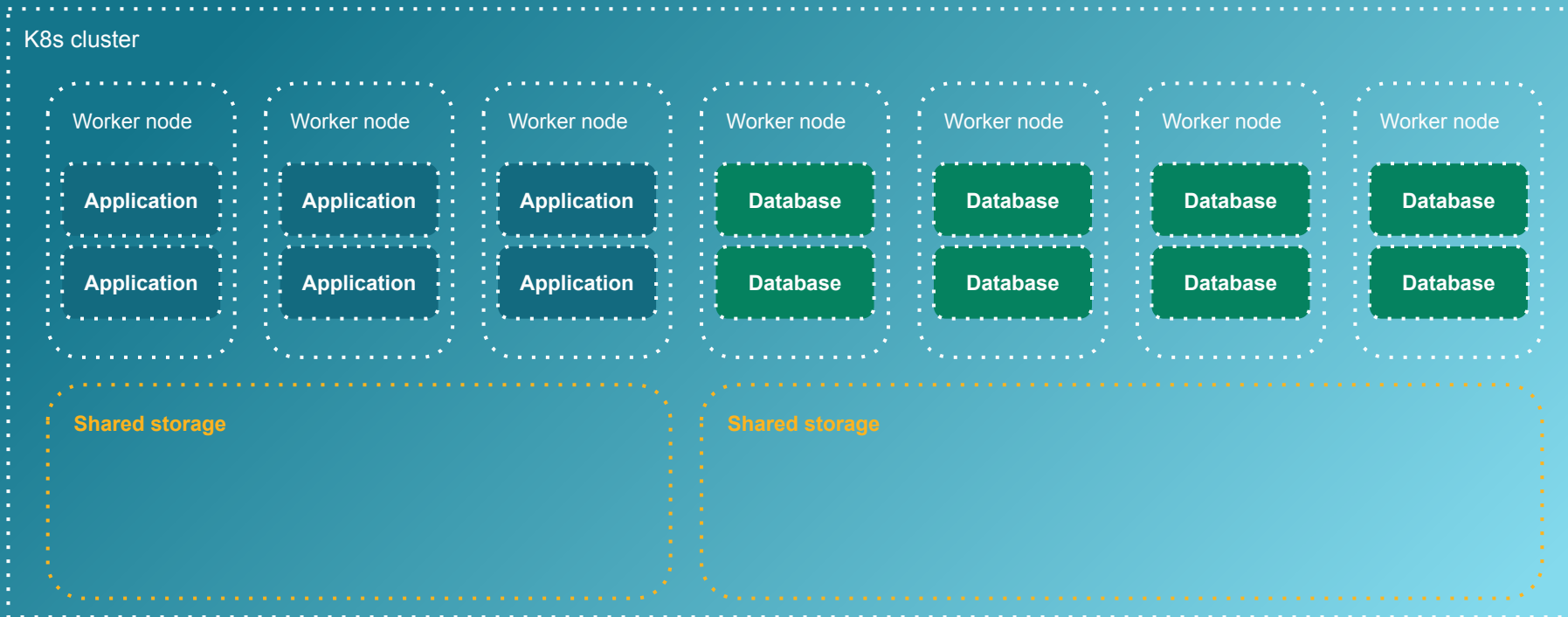
## Shared workloads, shared storage #1



## Shared workloads, shared storage #2



## Shared workloads, shared storage #3

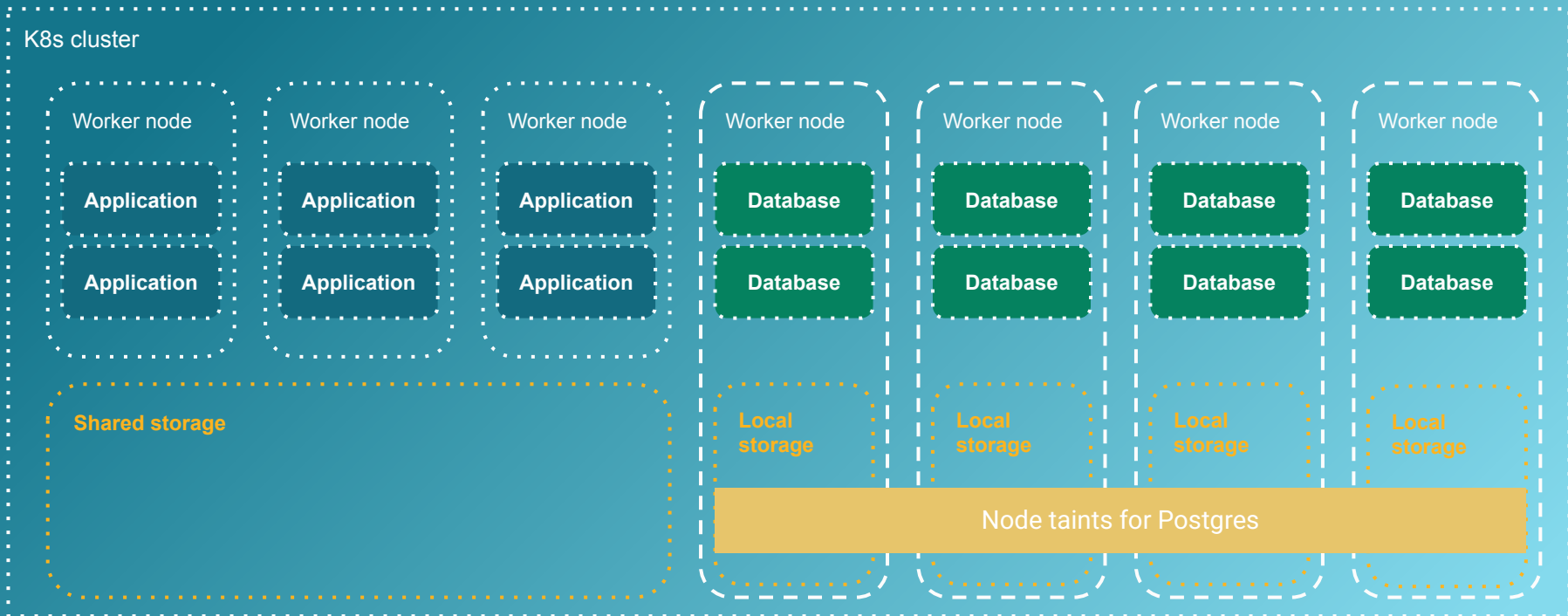


Shared workloads, local storage



Good value for money!

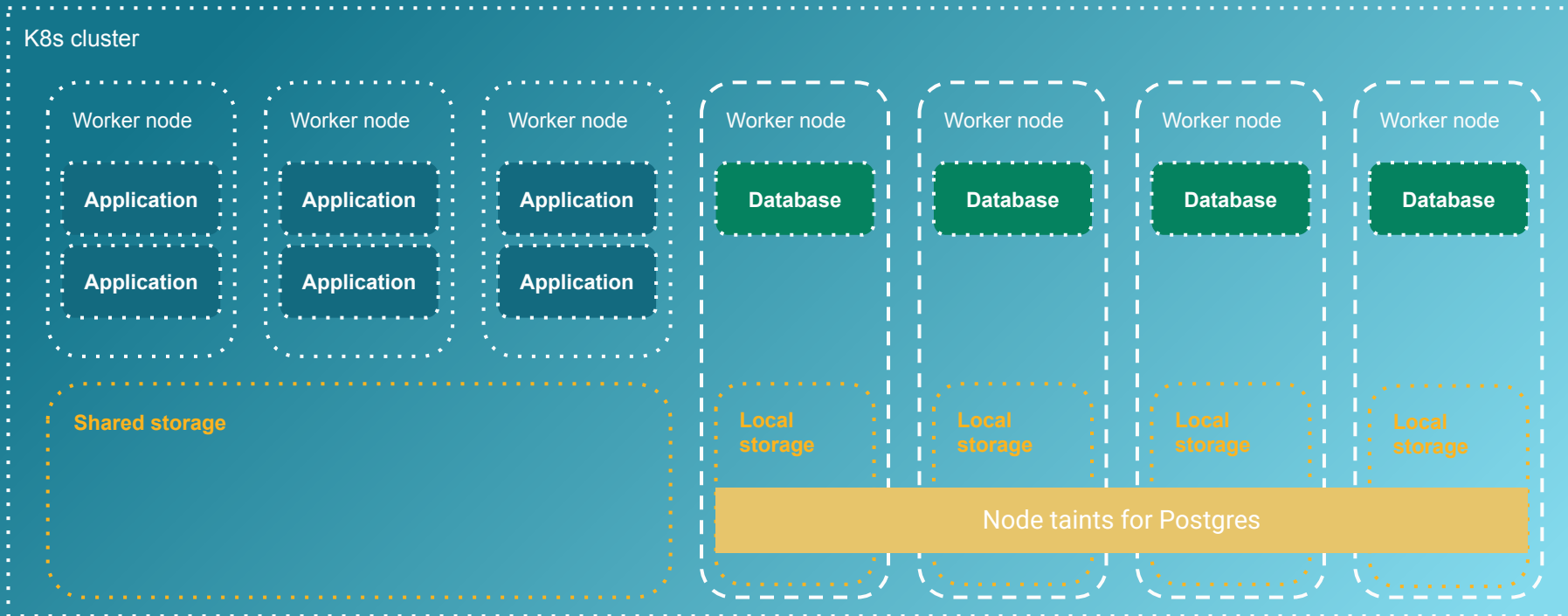
K8s cluster



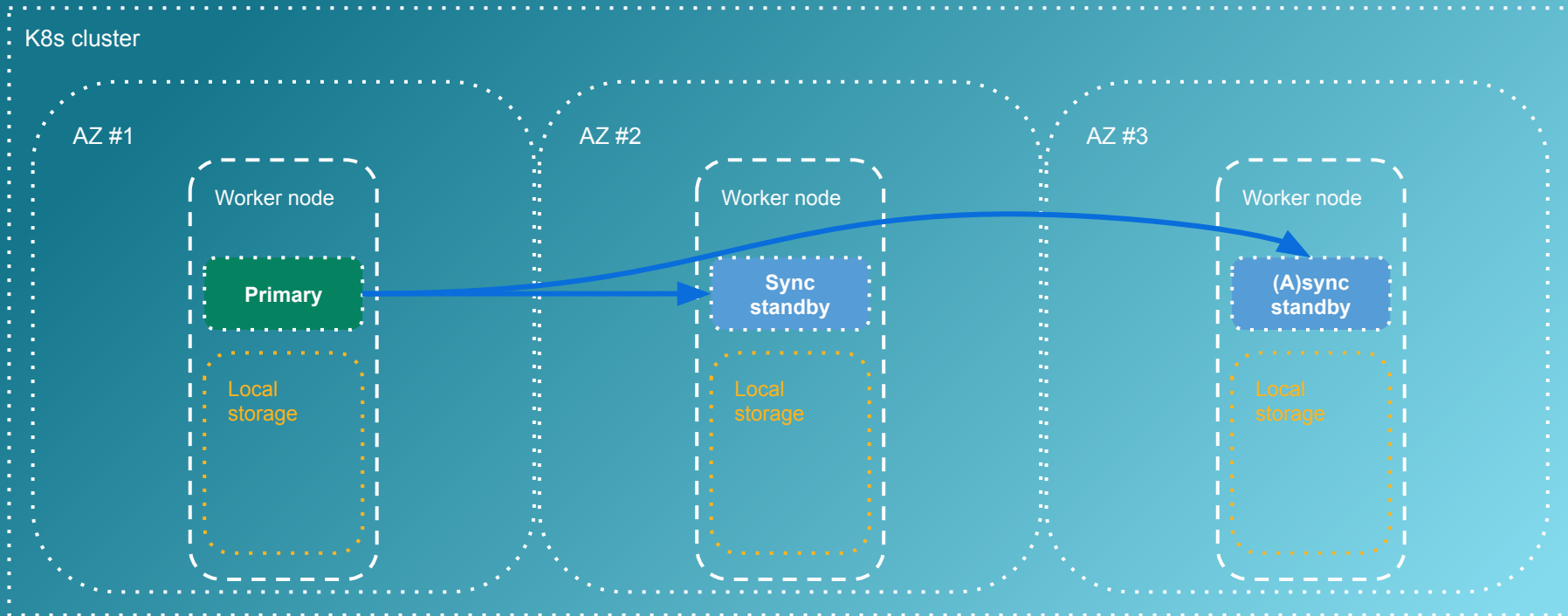
Dedicated workloads, local storage



Best Postgres results!

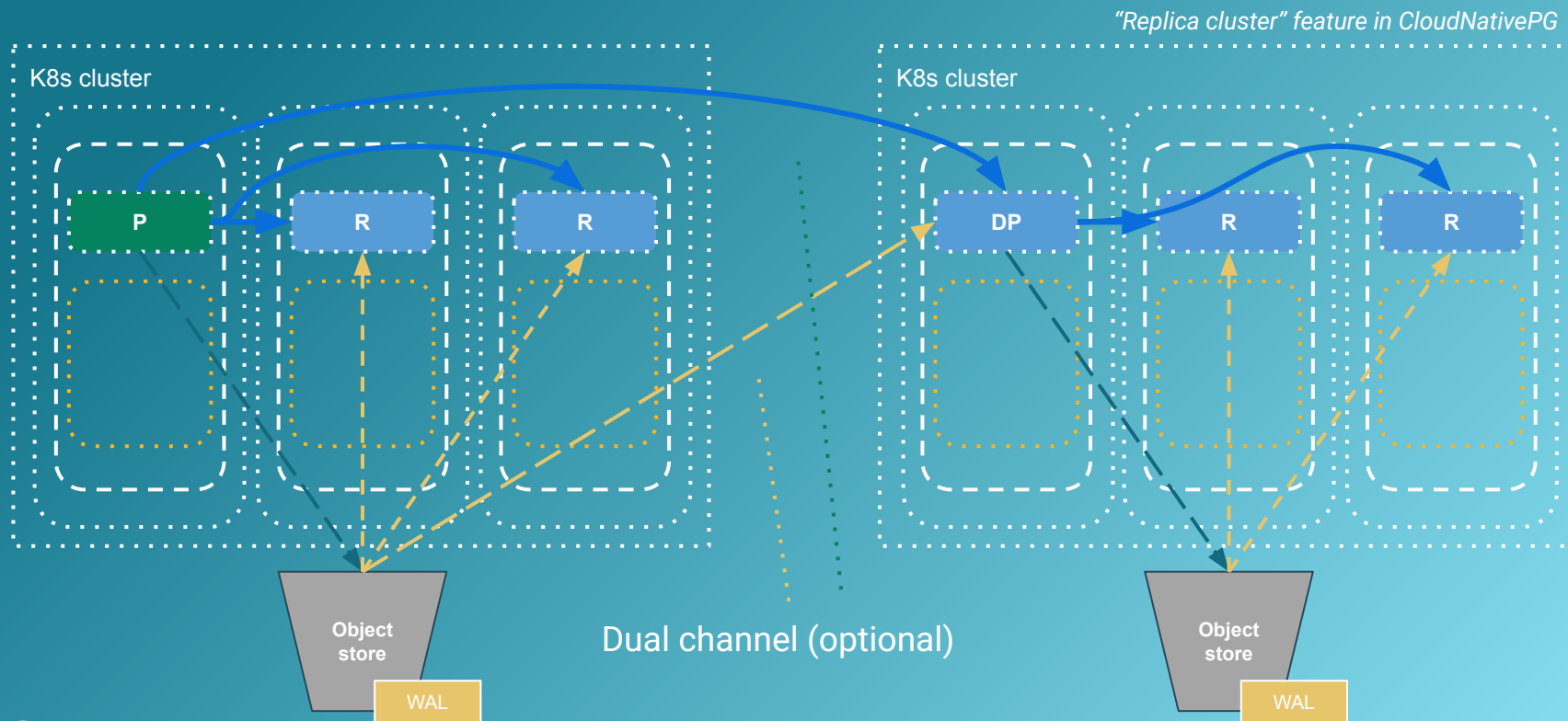


## Shared nothing architecture

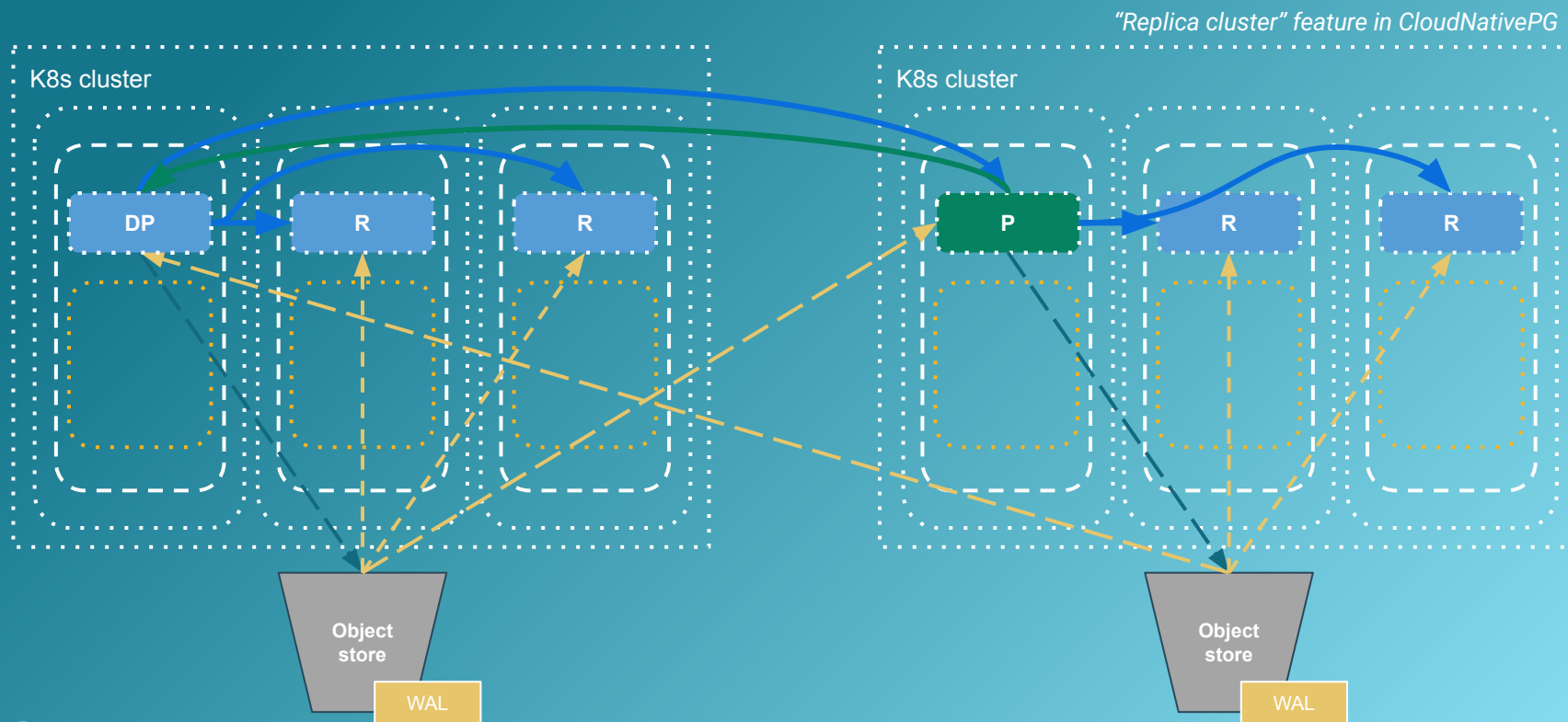




## Shared nothing architecture (hybrid/multi)



## Shared nothing architecture (hybrid/multi)



# Functionalities for CNPG



# Deploy a 3-node HA Postgres Cluster

- Install the latest PostgreSQL 15 minor
- Create a 3-node PostgreSQL 15 cluster
  - A primary, two standby (one synchronous)
  - Use mTLS authentication for the replicas
  - Use replication slots
- Resources:
  - RAM: 4 GB
  - CPU: 8 cores
  - Storage: 40GB for PGDATA, 10GB for WALs
- A user and a database for the application
- A reliable and consistent way to access the primary via network





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

```
kind: Cluster
```

```
metadata:
```

```
  name: myapp-db
```

```
spec:
```

```
  resources:
```

```
    requests:
```

```
      memory: "4Gi"
```

```
      cpu: 8
```

```
    limits:
```

```
      memory: "4Gi"
```

```
      cpu: 8
```

```
  ...
```

```
  ○
```



```
postgresql.conf
```

```
instances: 3
```

```
minSyncReplicas: 1
```

```
maxSyncReplicas: 1
```

```
replicationSlots:
```

```
  highAvailability:
```

```
    enabled: true
```

```
storage:
```

```
  size: 40Gi
```

```
walStorage:
```

```
  size: 10Gi
```

```
postgresql:
```

```
  parameters:
```

```
    shared_buffers: "1GB"
```



```
replication
```



```
storage
```

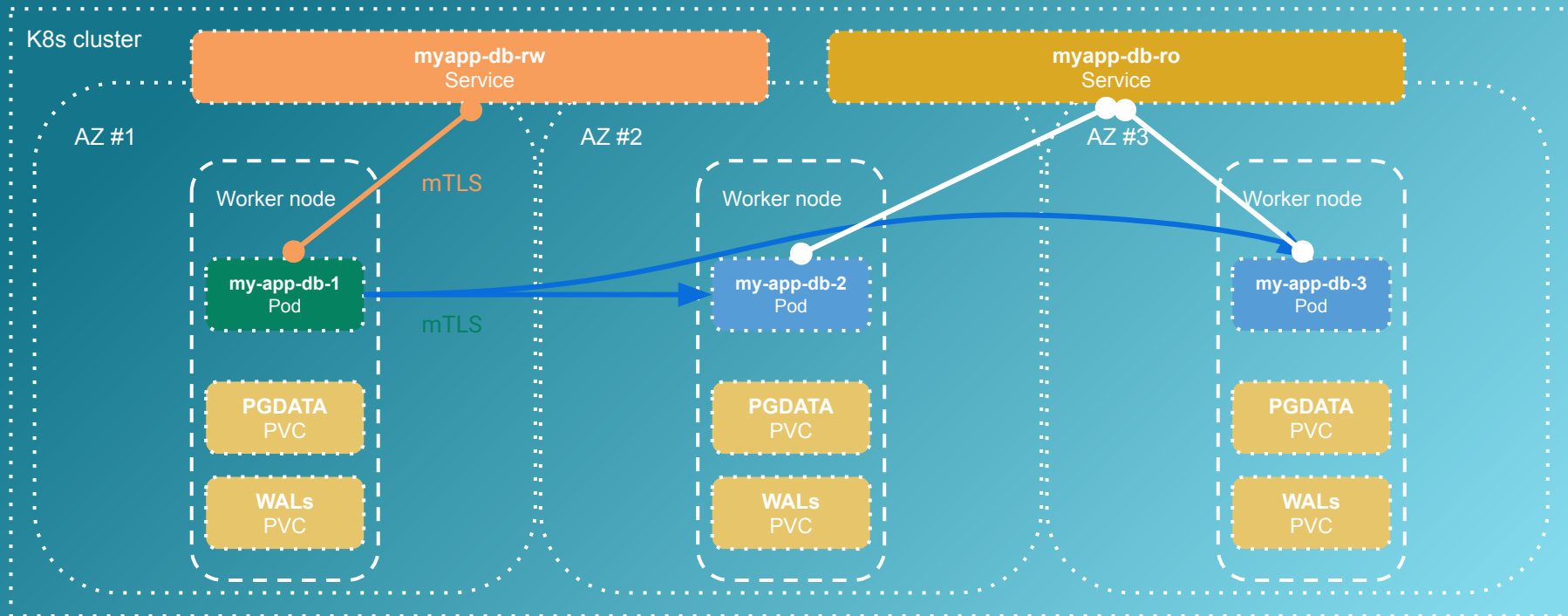


# How to deploy the PostgreSQL Cluster

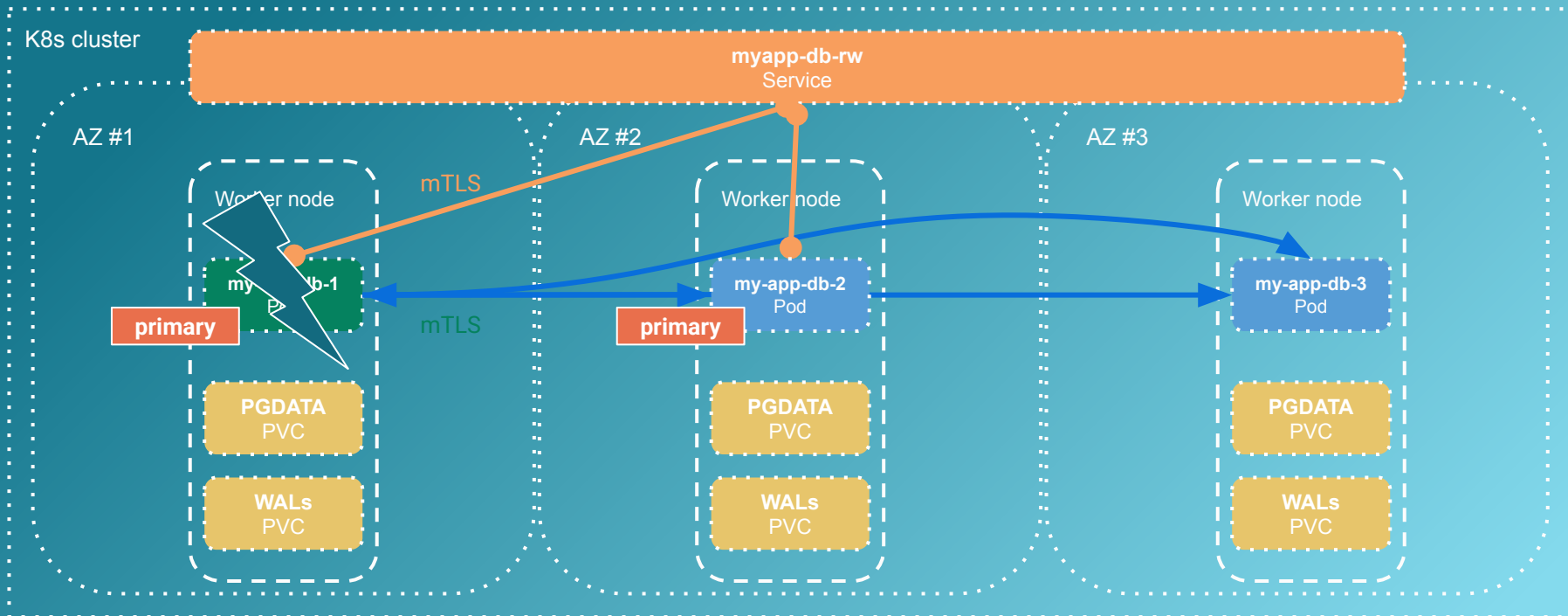
```
kubectl apply -f myapp-db.yaml
```



This is what happens under the hood



## Automated failover





# Advanced security



# 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 (TDE)

- 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 Key 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



# Main capabilities



# Bootstrap

- Create a new cluster from scratch
  - “initdb”: named after the standard “initdb” process in PostgreSQL that initializes an instance
  - This method can be used to migrate another database (“import”) or upgrade it
    - Uses pg\_dump and pg\_restore with some intelligence we’ve added
- Create a new cluster from an existing one:
  - Directly (“pg\_basebackup”), using physical streaming replication
  - 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



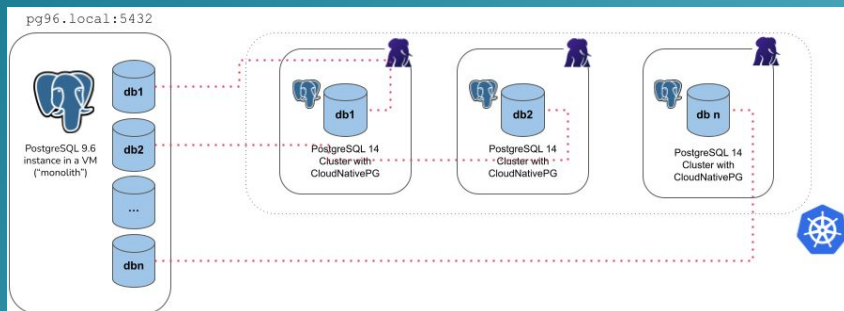
# Migrate from external instance

- Create a new cluster from scratch
  - “initdb”: named after the standard “initdb” process in PostgreSQL that initializes an instance
  - This method can be used to migrate another database (“import”) or upgrade it
    - Uses pg\_dump and pg\_restore with some intelligence we’ve added
- Create a new cluster from an existing one:
  - Directly (“pg\_basebackup”), using physical streaming replication
  - 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

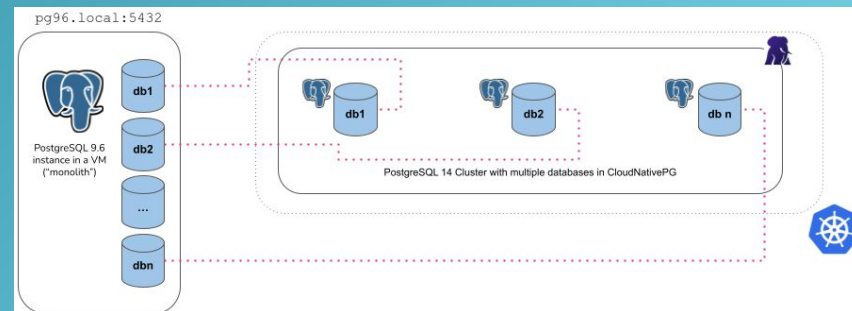


## Migrate from external instance

### Microservices



### Monolith



# 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





# Backup and Recovery - Part 1

- Continuous physical backup on “backup object stores”
  - Scheduled and on-demand base backups
  - Continuous WAL archiving (including parallel)
  - From primary or a standby
  - Support for recovery window retention policies (e.g. 30 days)
- 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 - Part 2

- WAL management
  - Object store
- Physical Base backups
  - Object store
  - Kubernetes level backup integration (Velero/OADP, Veem Kasten K10, generic interface)
  - Kubernetes Volume Snapshots



# Kubernetes Volume Snapshot: major advantages

- 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
<i>tiny</i>	4.5 GB	300	8 GB	1 GB	67s
<i>small</i>	44 GB	3,000	80 GB	10 GB	10m 50s
<i>medium</i>	438 GB	3,0000	800 GB	100 GB	3h 15m 34s
<i>large</i>	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
<i>tiny</i>	2m 43s	4m 16s	31s
<i>small</i>	20m 38s	16m 45s	27s
<i>medium</i>	2h 42m	2h 34m	48s
<i>large</i>	3h 54m 6s	2h 3s	2m 2s

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



# Native Prometheus exporter for monitoring

- Built-in metrics at the operator level
- Built-in metrics at the Postgres instance level
  - Customizable metrics at the Postgres instance level
    - Via ConfigMap(s) and/or Secret(s)
    - Syntax compatible with the PostgreSQL Prometheus Exporter
    - Auto-discovery of databases
  - Queries are:
    - transactionally atomic and read-only
    - executed with the `pg_monitor` role
    - executed with `application_name` set to `cnp_metrics_exporter`
- Support for `pg_stat_statements` and `auto_explain`



# Prometheus



localhost



## Graph


Help

☐ Use local time☐ Enable query history☒ Enable autocomplete☒ Enable highlighting☒ Enable linting

 **cnpg\_backends\_max\_tx\_duration\_seconds**

*gauge*

## Table

 **cnpg\_backends\_total**

*gauge*

 **cnpg\_backends\_waiting\_total**

*gauge*



*gauge*



*counter*

 `cnpg_collector_first_recoverability_point`

*gauge*

No d

 **cnpg\_collector\_last\_collection\_error**

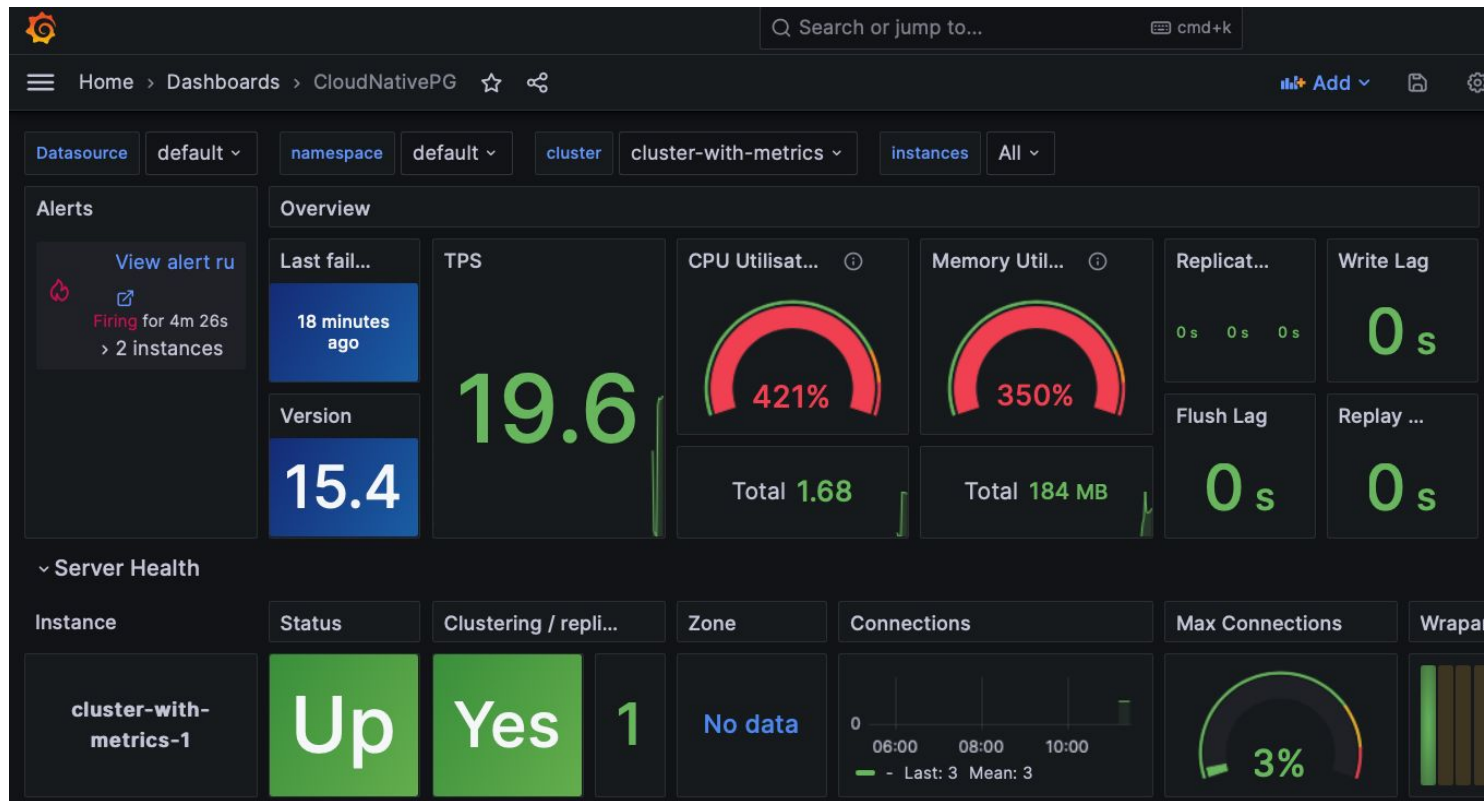
gauge

 **cnpg\_collector\_lo\_pages**

*gauge*



# Grafana Dashboard



# Logging

- All components directly log to standard output in JSON format
- Each entry has the following structure:
  - level: log level (info, notice, ...)
  - ts: the timestamp (epoch with microseconds)
  - logger: the type of the record (e.g. postgres or pg\_controldata)
  - msg: the type of the record (e.g. postgres or pg\_controldata)
  - record: the actual record (with structure that varies depending on the msg type)
- Seamless integration with many log management stacks in Kubernetes
- Support for PGAudit
  - EDB Audit as well for EDB Postgres for Kubernetes





# 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
  - Start pgbench



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: 00000004000000000000000C)

#### 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: 00000004000000000000000B @ 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



# Connection pooling with PgBouncer

- Managed by the “Pooler” Custom Resource Definition
- Directly mapped to a service of a given Postgres cluster
- Deploys multiple instances of PgBouncer for High Availability
- Supports Pod templates, very important for Pod scheduling rules
- Transparent support for password authentication
- Connects to PostgreSQL via a standard user through a TLS certificate
- Supports configuration for most of PgBouncer options
- Automated integration with Prometheus
- JSON log in standard output



# Hands-on



# Features shown during the demo

- Kubernetes plugin install
- CloudNativePG operator install
- Postgres cluster install
- Insert data in the cluster
- Failover
- Backup
- Recovery
- Scale out/down
- Fencing
- Hibernation
- Monitoring
- Rolling updates (minor and major)

Deployment

Administration

Backup and  
Recovery

High Availability

Monitoring

Last CloudNativePG tested version is 1.25



This demo is in



<https://github.com/sergioenterprisedb/kubecon2022-demo>



# VMs IP Addresses

vm	vm ip	public ip
vm1		
vm2		
vm3		
vm4		
vm5		
vm6		
vm7		
vm8		
vm9		

vm	vm ip	public ip
vm10		
vm11		
vm12		
vm13		
vm14		
vm15		
vm16		
vm17		
vm18		



# Setup the demo env





# Step 1: Connect to your demo environment

- Connect to the terminal 1:
  - Run in the browser <https://<your vm ip address>>
  - Connect to the terminal as user **workshop** with the password **workshop**
- Connect to the terminal 2:
  - Open the new browser tab.
  - Run <https://<your vm ip address>>
  - Connect to the terminal as user **workshop** with the password **workshop**



## Step 2: Create k3d cluster

- In the terminal 1:
  - Go to the directory `/home/workshop/workshop/cnp_demo`:  
`cd /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`



## Step 3: Run Minio server

- In the terminal 1:
  - Go to the directory `/home/workshop/workshop/cnp_demo`:  
`cd /home/workshop/workshop/cnp_demo`
  - Start MinIO:  
`./start_minio_docker_server.sh &`



# Install the operator



# Step 4: Install CNPG Operator and CNPG plugin

- In the terminal 1:
  - Install CNPG plugin:  
`./01_install_plugin.sh`
  - Install CNPG operator:  
`./02_install_operator.sh`
  - Check the installation of the operator (try several times):  
`./03_check_operator_installed.sh`



# Create the postgres cluster



# Step 5: Install Postgres cluster

- In the terminal 1:
  - Create the yaml file:  
`./04_get_cluster_config_file.sh`
  - Create the postgres cluster:  
`./05_install_cluster.sh`
- In the terminal **2**:
  - Go to the directory `/home/workshop/workshop/cnp_demo`:  
`cd /home/workshop/workshop/cnp_demo`
  - Check the Postgres cluster status:  
`./06_show_status.sh`



## Step 6: Create table test with 1000 rows

- In the terminal 1:
  - Run the script:  
`./07_insert_data.sh`
  - Check the data in the table test:  
`./check_table_test.sh`



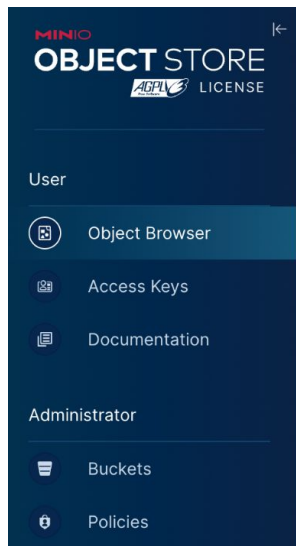


# Upgrade the postgres cluster



# Step 7: 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:



## Object Browser



### Buckets

MinIO uses buckets to organize objects. A bucket is similar to a folder or directory in a filesystem, where each bucket can hold an arbitrary number of objects.

To get started, [Create a Bucket](#).



## Step 8: Check the cluster status

- In the terminal 1:
  - Run the command  
`kubectl -cnpg status cluster-example`
  - In the output
    - check Postgres version: "PostgreSQL Image: ghcr.io/cloudnative-pg/postgresql:**16.1**"
    - check "Continuous Backup status": "**Not configured**"



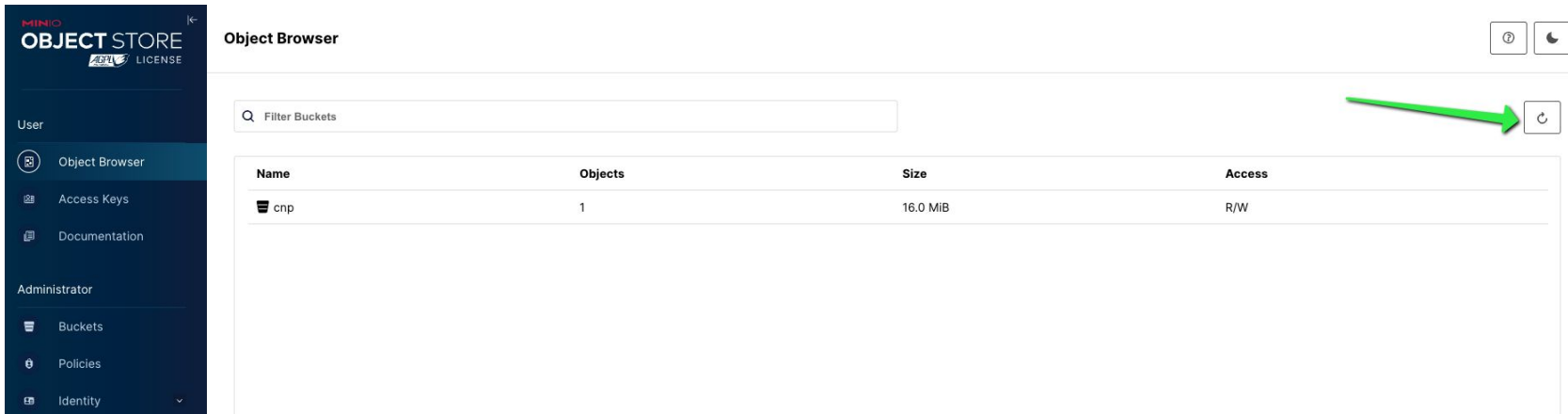
## Step 9: Run the script 09\_upgrade.sh

- With this step we will:
  - Run the postgres minor update from the version 16.1 to 16.4
  - We will configure the WAL files backup to the MinIO storage
- In the terminal 1:
  - Run the script:  
`./09_upgrade.sh`
- In the terminal **2**:
  - Check the upgrade status:  
`./06_show_status.sh`



# Step 10: Check the WAL backup on the MinIO server:

- In the browser tab - MinIO server
  - Press “refresh” button:



The screenshot displays the MinIO Object Browser interface. On the left is a dark sidebar with the 'MINIO OBJECT STORE' logo and a navigation menu. The main area is titled 'Object Browser' and contains a search bar labeled 'Filter Buckets'. Below the search bar is a table with the following data:

Name	Objects	Size	Access
cnp	1	16.0 MiB	R/W

A green arrow points to a circular refresh icon (a square with a circular arrow) located in the top right corner of the main content area, next to the search bar.



# Backup & Restore



# Step 11: 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 backup in the MinIO GUI



# Step 12: 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 script:  
`./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:  
`./check_restore_table_test.sh`





# Failover



# Step 13: Run failover test

- With this step we will:
  - Delete the primary database of the cluster cluster-example
  - Check the cluster status in the another terminal window
- In the terminal 1:
  - Run the script:  
`./13_failover.sh`
- In the terminal **2**:
  - Check the failover cluster status:  
`./06_show_status.sh`



# Scale-out and scale-down



# Step 14: Scale-out the postgres cluster

- With this step we will:
  - Add the 1 standby to the cluster
- In the terminal 1:
  - Run the script:  
`./14_scale_out.sh`
- In the terminal **2**:
  - Check the cluster status:  
`./06_show_status.sh`



# Step 15: Scale-down the postgres cluster

- With this step we will:
  - Remove 2 standby pods from the cluster
- In the terminal 1:
  - Run the script:  
`./15_scale_down.sh`
- In the terminal **2**:
  - Check the cluster status:  
`./06_show_status.sh`



# Fencing



# Step 16: Stop postgres process on the pod

- In the terminal 1:
  - Run the script:  
`./30_fencing_on.sh`
- In the terminal **2**:
  - Check the cluster status:  
`./06_show_status.sh`



# Step 17: Start the postgres process on the pod

- In the terminal 1:
  - Run the script:  
`./31_fencing_off.sh`
- In the terminal **2**:
  - Check the cluster status:  
`./06_show_status.sh`





# Hibernation



# Step 18: Stop the postgres cluster

- In the terminal 1:
  - Run the script:  
`./32_hibernation_on.sh`
- In the terminal **2**:
  - Check the cluster status:  
`./06_show_status.sh`



# Step 19: Start the postgres cluster

- In the terminal 1:
  - Run the script:  
`./33_hibernation_off.sh`
- In the terminal **2**:
  - Check the cluster status:  
`./06_show_status.sh`



# Major Upgrade



# Step 20: Create the Postgres 16 Cluster

- In the terminal 1:
  - Change directory to /home/workshop/workshop/cnp-demo/major\_upgrade\_demo  
`cd /home/workshop/workshop/cnp-demo/major_upgrade_demo`
  - Create the cluster v16::  
`./04_create_cluster_v16.sh`
  - Check the cluster status:  
`./05_show_status_v16.sh`
  - Insert the data:  
`./06_insert_data_cluster_v16.sh`
  - Verify the inserted data::  
`./07_verify_data_inserted.sh`



## Step 21: Create the Postgres 17 Cluster and import data from PG 16

- Create the cluster postgres v17 and import the data from the postgres v16:

`./08_upgrade_v16_to_v17.sh`

- Check the cluster status:

`./09_show_status_v17.sh`

- Verify the data in the postgres v17:

`./10_verify_data_migrated_16_17.sh`



# Monitoring



# Setup monitoring

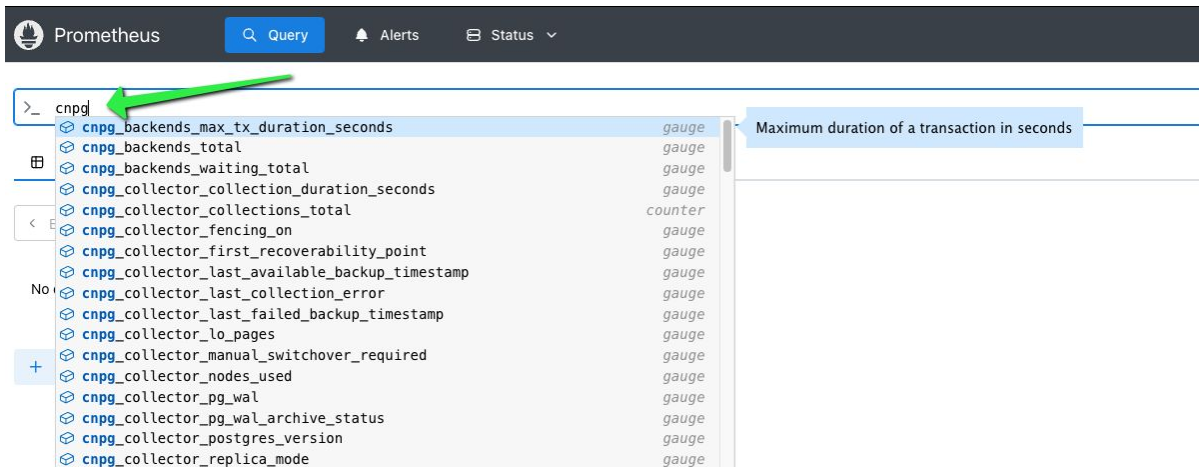
- In the terminal 1:
  - Run the command:  
`kubectl get pods`
  - Change the directory::  
`cd /home/workshop/workshop/cnp-demo/monitoring`
  - Install the prometheus rules:  
`./01_prometheus_rules.sh`
  - Start port forwarding for prometheus and grafana:  
`./02_port_forwarding_prometheus_grafana.sh`
- Download the Grafana Dashboard to your laptop:
  - <https://github.com/cloudnative-pg/grafana-dashboards/blob/main/charts/cluster/grafana-dashboard.json>





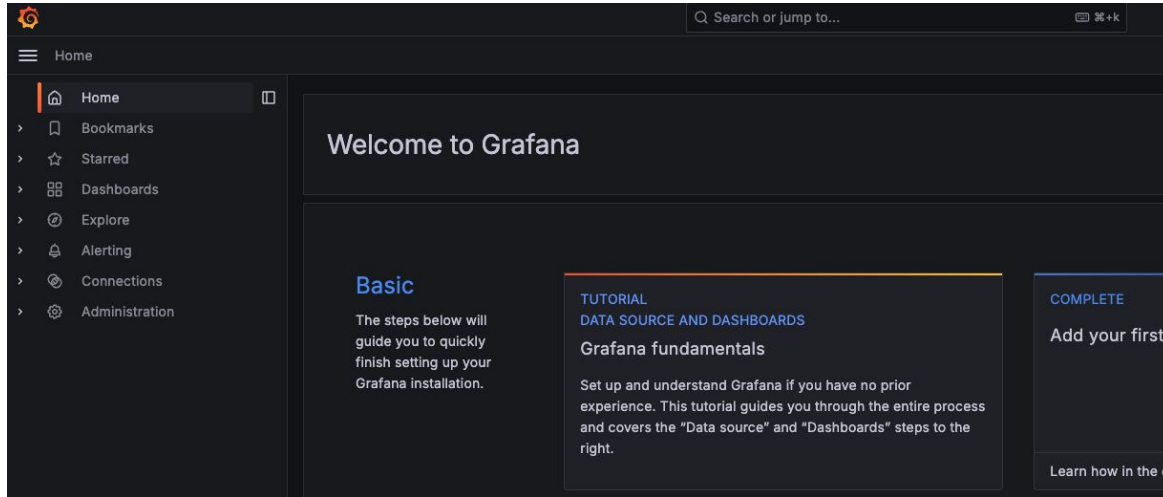
# Explore Prometheus

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



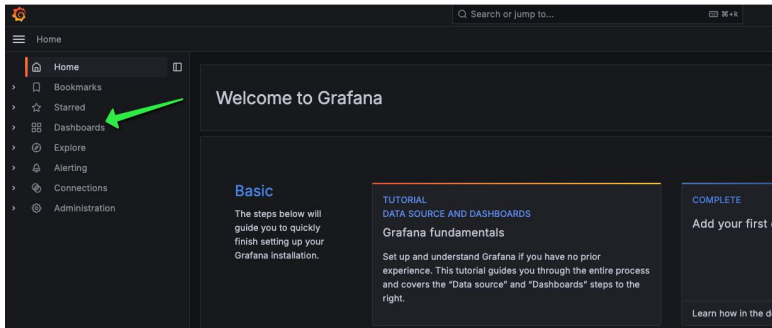
# Access the Grafana page

- 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

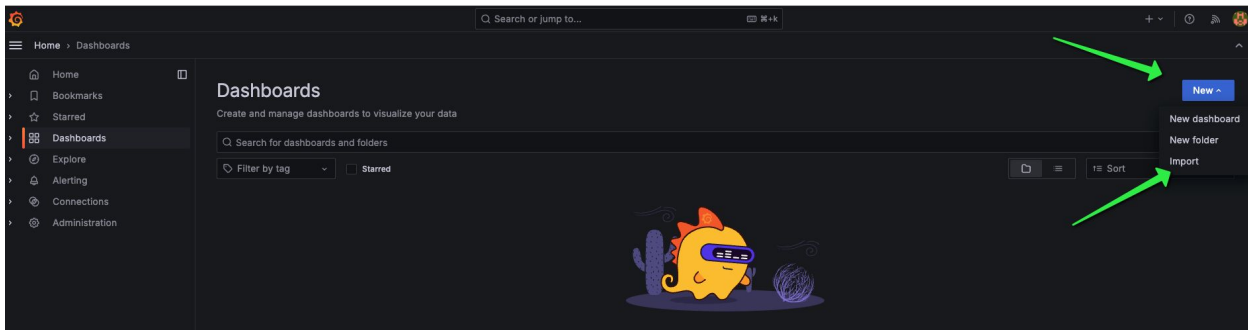


# Configure Grafana

- Go to Dashboards

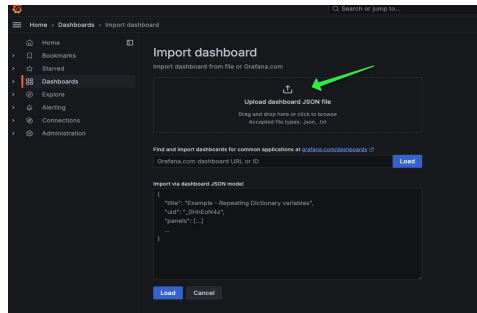


- Press “New”, then “Import”:

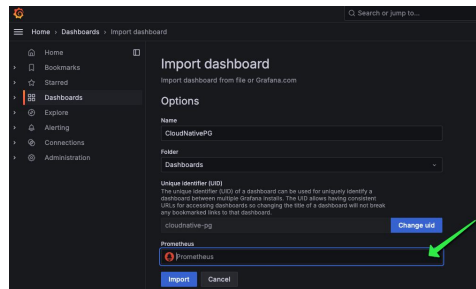


# Configure Grafana - continued

- Upload the Dashboard json file:



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



# Explore CNPG Dashboard- continued

- Explore the CNPG Dashboard:





**EDB**

Postgres® for the AI Generation



**VSHN**

Thank you for participating in the  
Postgres on Kubernetes Workshop