EDB
Postgres® for the AI Generation

Mastering PostgreSQL in Kubernetes with CloudNativePG

Course introduction

Neel Patel - Principal Software Engineer at EDB
Danish Khan - Senior SDE at EDB

March 5, 2025 - PGConf India, 2025

About Danish

Senior SDE, EDB

- Certified Kubernetes application Developer
- Certified Kubernetes Admin
- DoK Ambassador
- Vmware certified VCP-DCP 2014
- Virtualization Enthusiast
- Open Source CloudNativePG contributer
- Major Contributor in patented unified test framework https://patents.justia.com/patent/11366747 .
- With EDB since ~5years

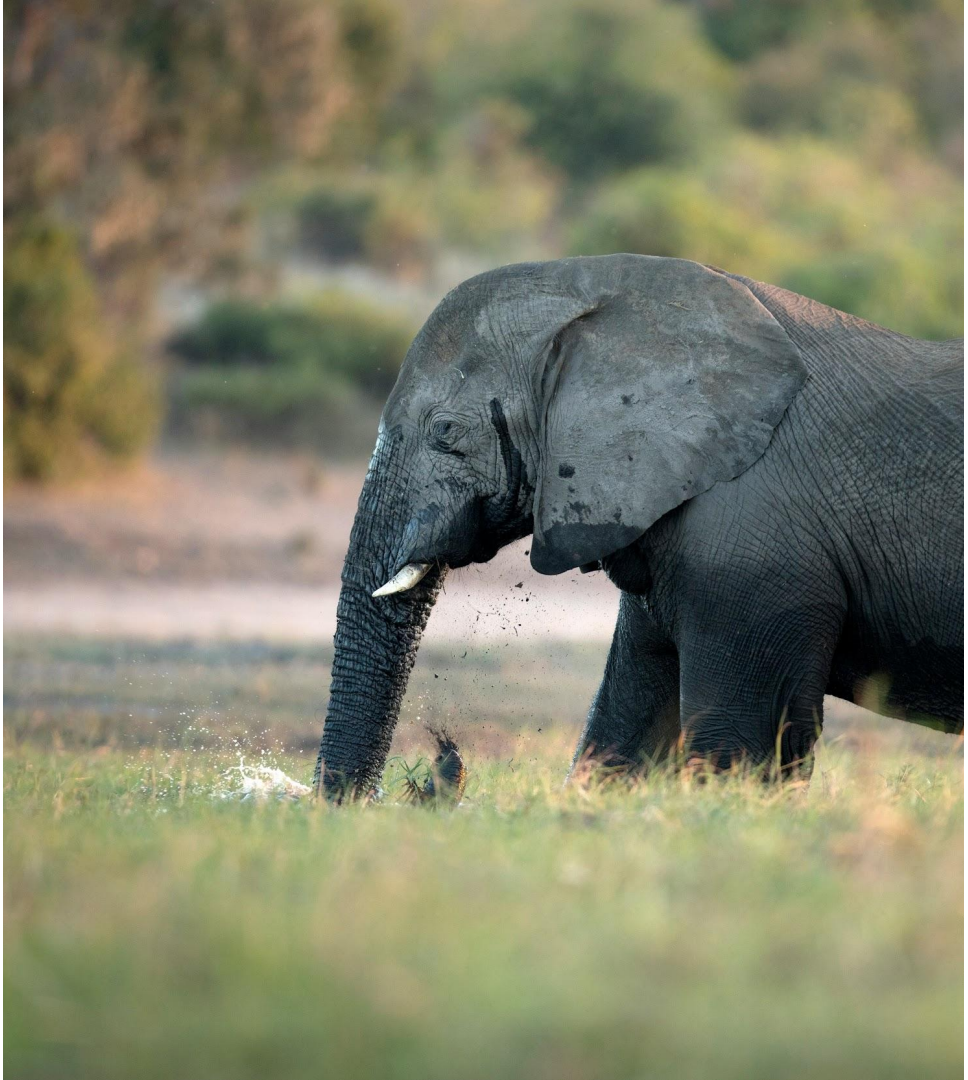
About Neel

Principal Software Engineer at EDB

- With EDB since ~10 years
- Using PostgreSQL since ~ 10 years
- Open source contributor
- Kubernetes and GoLang Developer

Agenda

- Overview of the open-source stack: PostgreSQL, Kubernetes, CloudNativePG
  Hands-on: Setting up the local environment and presenting the lab exercise (working in 2-3 person groups is encouraged). PostgreSQL 17 will be recommended for usage.
- Hands-on: Bootstrapping a new PostgreSQL cluster
- Hands-on: Configuring PostgreSQL
- Hands-on: Scaling up/down
- Hands-on: Connecting from a sample application
- Hands-on: simulating an automated failover
- Hands-on: Setting up continuous backup on object stores
- Hands-on: `cnpg` plugin to observe the status and on Demand Backup
- Hands-on: Full and Point-In-Time Recovery
- Metrics and Logging
- Hibernation & Rehydration
- Q&A



EDB
Postgres® for the AI Generation

Part 1

PostgreSQL

What is PostgreSQL?

The world's most advanced database. Also known as **Postgres**. URL: postgresql.org

- **100% Open Source**
  - Widely used, extremely robust, and feature-rich.
- **Extensible and Customizable**
  - Support for custom data types, functions, and procedural languages.
- **Advanced features**
  - Includes replication, partitioning, full-text search, and JSON support.
- **ACID-Compliant**
  - Ensures Atomicity, Consistency, Isolation, and Durability for transactions.
  - Trusted choice for mission-critical applications
- **Strong Community Support**
  - Backed by a large, active global community.

EDB
Postgres® for the AI Generation

Part 2

Kubernetes

What is Kubernetes?

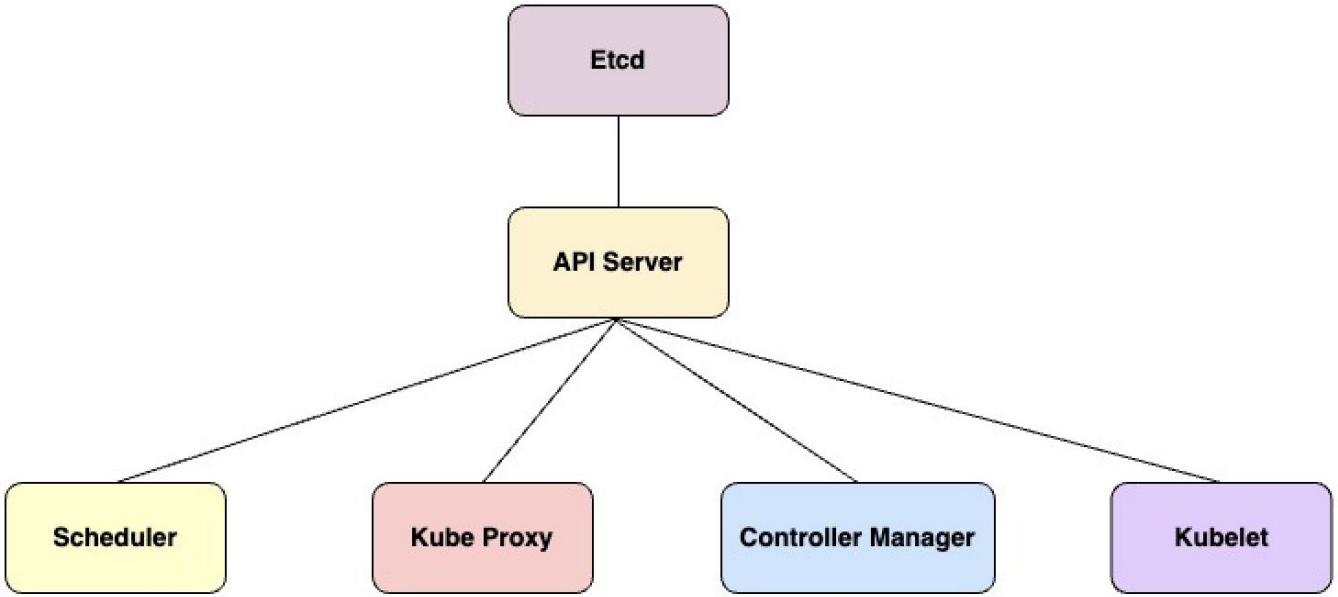
Greek for helmsman. Also known as **k8s**. URL: kubernetes.io

- Orchestration system for containers
- Automates deployment, administration and scaling of:
  - **infrastructure**
  - **cloud native applications** - also known as **workloads**
- Open Source (Apache Licence 2.0) since 2014
  - v1.0 released in 2015
- Owned by the Cloud Native Computing Foundation (CNCF, cncf.io)
- Written in Go

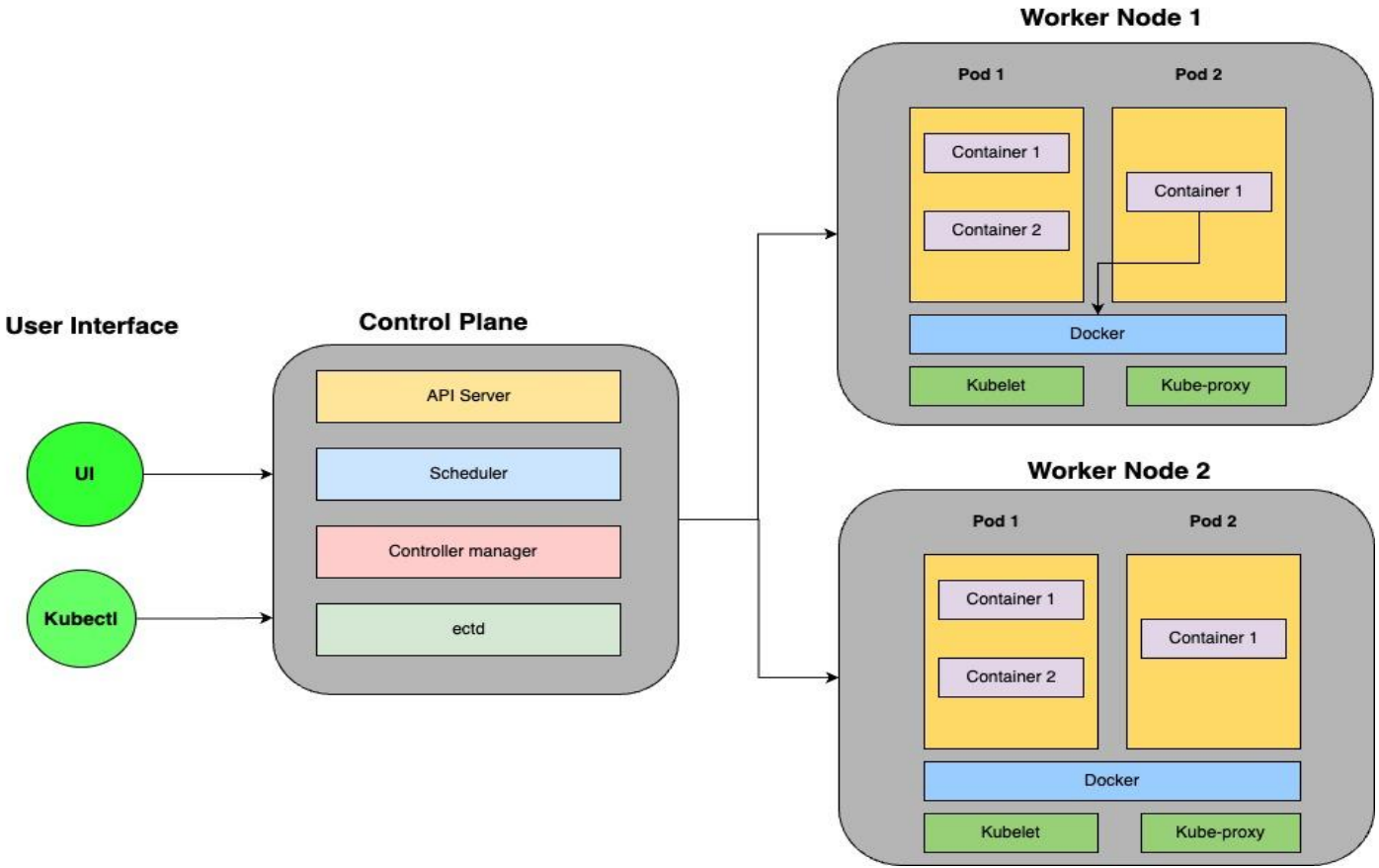
Why Kubernetes?

- **Flexibility:** Kubernetes works with many different infrastructures, including on-premise services, private and public clouds.
- **Efficiency:** Kubernetes optimizes the use of hardware resources, saving costs.
- **Self-healing mechanisms**: kubernetes constantly checks the health of nodes and containers and restart/replace the failed node/pods if it is not responding to user-defined health checks.
- **Security and resource management**: Provides security features and efficient management of resources, ensuring that the infrastructure is secure and the resources are used optimally.
- Kubernetes helps developers reduce time to market by automating many of the manual tasks associated with application development and deployment.

Main Components of Kubernetes



Kubernetes Architecture



Kubernetes vs Virtual Machines

Kubernetes

Infrastructure and containerised apps

Containers share the same Kernel

Immutable Application Containers

Portable on any cloud, standard

VMs

Operating systems

VMs have their own OS

Mutable (“dnf update”)

Indirectly portable

From infrastructure to applications

Fundamental concepts for DBAs to better understand Kubernetes

- **Region (Kubernetes Cluster)**
- **Data Centre (Availability Zone)**
- **Nodes**:
  - Linux with K8s
  - VMs or Bare Metal
- **Network**
  - The `Service` resource
    - `ClusterIP` type
    - `NodePort` type
    - `LoadBalancer` type
- **Storage**:
  - Network vs Local
  - Loca storage - Uses local disks attached to Kubernetes nodes.
  - Container Storage Interface (CSI)
  - The `StorageClass` Resource
- **Applications (workloads)**:
  - The `Deployment` resource
  - The `Statefulset` resource
  - Databases are applications

EDB
Postgres® for the AI Generation

Part 3

Kubernetes Operators

What is Kubernetes operator ?

Extends kubernetes controller and it defines how complex applications work

- A Kubernetes operator automates actions of human being in a programmatic way
- A PostgreSQL database is a complex application:
  - Deployment and configuration
  - Failure detection and Failover
  - Updates and switchovers
  - Backup and Recovery
- Relies on Kubernetes' native components and its capabilities.
  - Self healing. High availability, scalability, resource control etc.
  - Declarative and fully managed

Imperative vs Declarative with an example

Imperative (VMs)

1. Create a PostgreSQL 17 instance
2. Configure for replication
3. Clone a second one
4. Set it as a replica
5. Clone a third one
6. Set it as a replica

Declarative (Kubernetes)

There's a PostgreSQL 17 cluster with 3 instances (i.e. one primary and two replicas). **At any time.**

Extending the Kubernetes API

- Kubernetes API is extensible (similarly to PostgreSQL)
  - You can define new data types with custom resources
  - You can define a custom controller for a custom resource (or more)
    - You declare the state of the custom resource
    - The controller ensures that the **current** state matches the **declared** one
      - Foundation of self-healing
- The operator pattern is a development pattern
  - Designed to manage complex applications
  - Custom resources and custom controllers
  - Ensures a declarative API

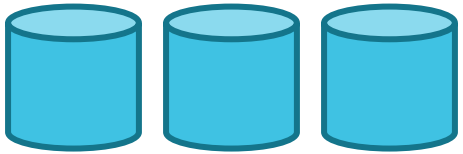
The Simplified Kubernetes Resource Model

- Resources are identified by:
  - Namespace
  - Name
  - **Kind**/Type (e.g., Pod, Service)
- Resource Structure:
  - **spec** (Desired state)
  - **status** (Observed state)
- Key API Operations (Primitives):
  - **Get**: Retrieve resource information
  - **Put**: Modify a resource
  - **Watch**: Monitor resource changes over time

A PostgreSQL 17 cluster resource with 3 instances

spec (Desired State)

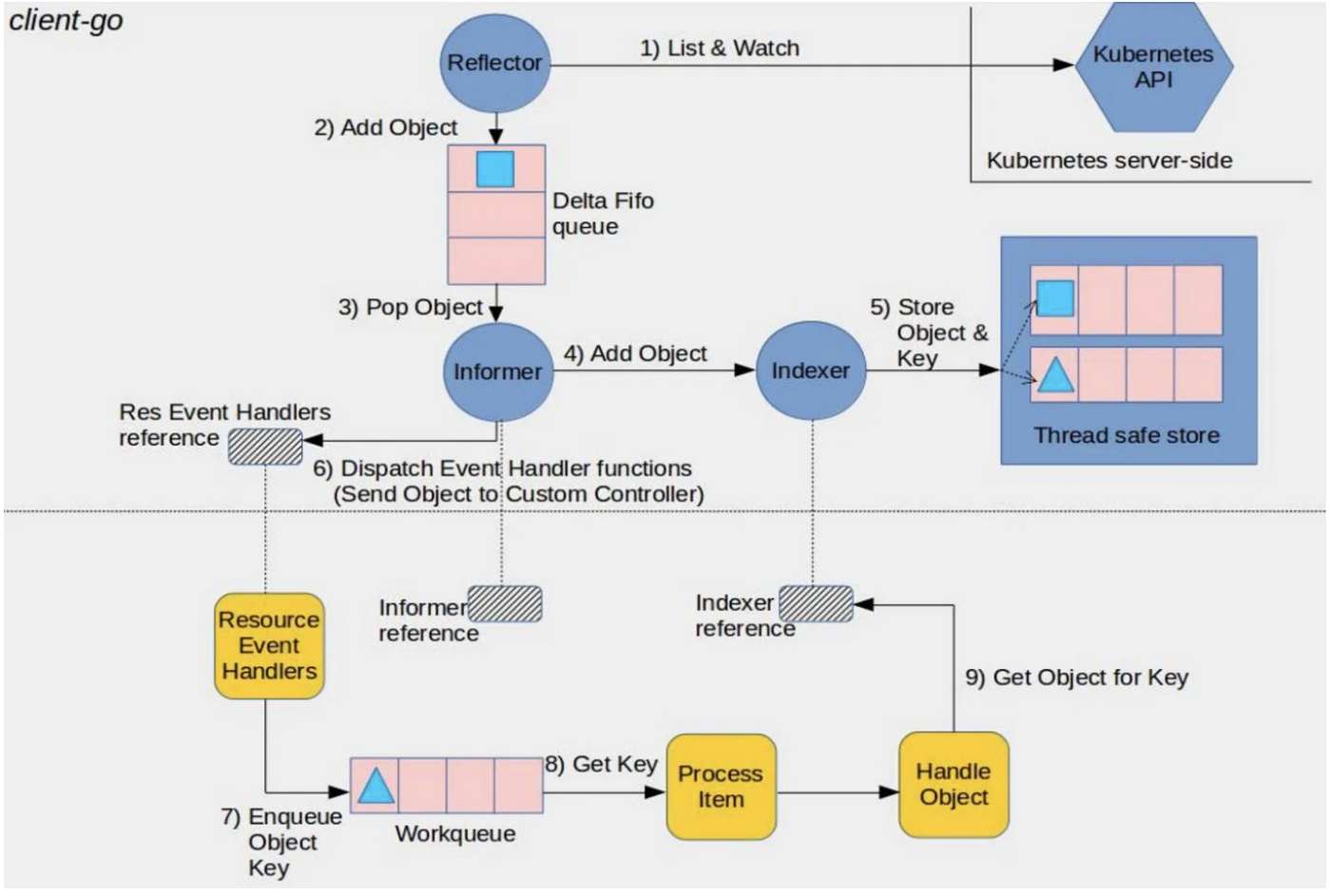
status (Observed State)

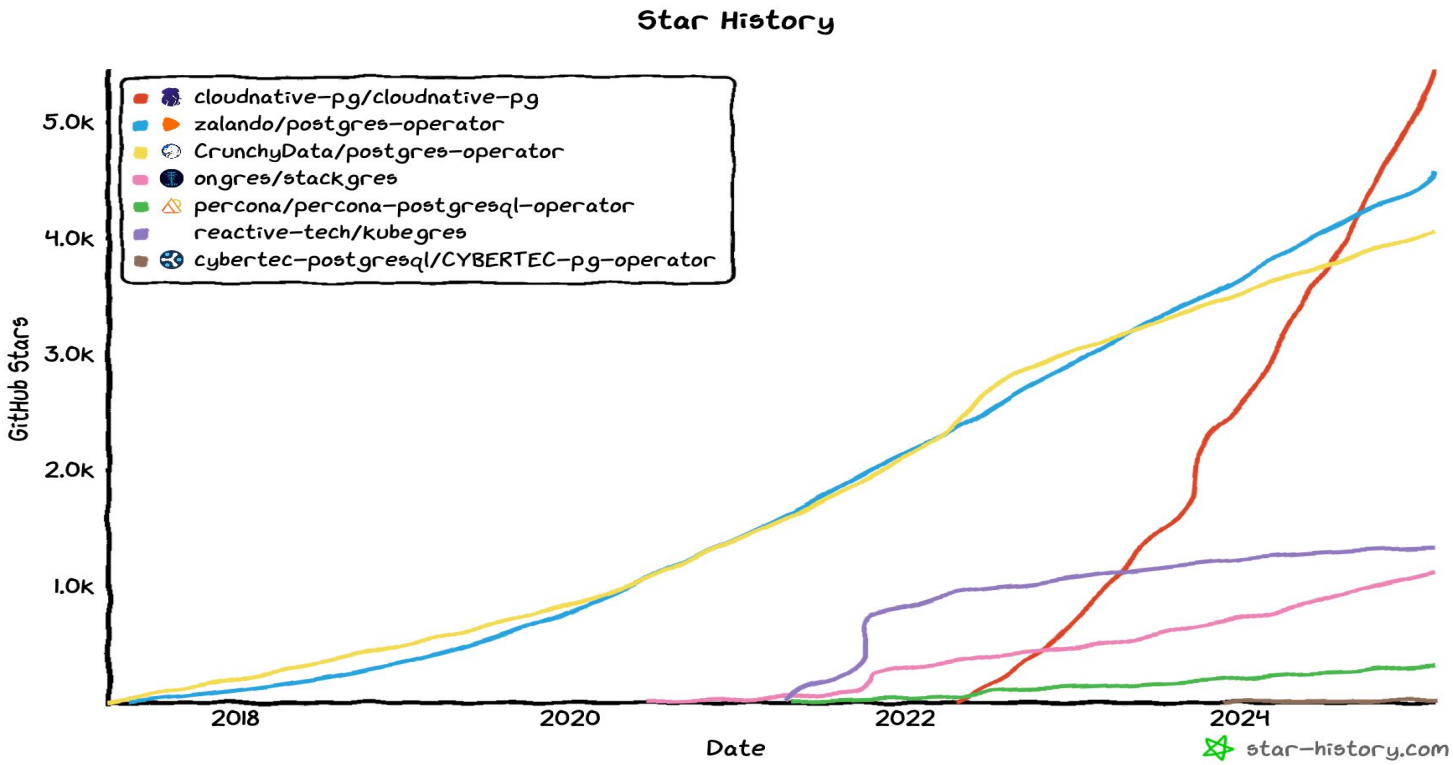


Nothing happens without a controller for the cluster resource

(simplified view)

Reconciliation loop





March 3rd, 2025

CloudNativePG has reached the ♯1 spot for the first time, surpassing Zalando's PostgreSQL operator—*a major milestone that highlights the growing impact and recognition of the project within the cloud-native ecosystem.*

EDB Postgres® for the AI Generation

Part 4

CloudNativePG

What is CloudNativePG?

“CloudNativePG is an open source operator designed to manage PostgreSQL workloads on any supported Kubernetes cluster running in private, public, hybrid, or multi-cloud environments. CloudNativePG adheres to DevOps principles and concepts such as declarative configuration and immutable infrastructure.”

The history of CloudNativePG

Bringing PostgreSQL to Kubernetes (not Kubernetes to PostgreSQL)

2019-2020: The 2ndQuadrant era

- MVP (fail-fast)
- Cloud Native BDR (closed source)
- Cloud Native PostgreSQL (closed source)

2021-2022: EDB Cloud Native PostgreSQL

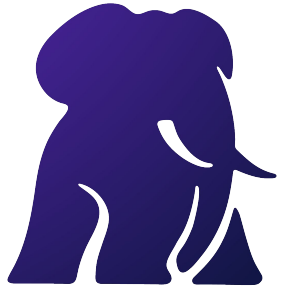
- Closed source
- Production ready (IBM Cloud Pak)
- Several features gradually introduced
  - HA + DR
  - Observability
  - Security
  - Pooling

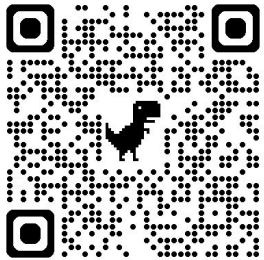
2022-: CloudNativePG Revolution

- Launched in May 2022
- Consistent commitment from EDB
  - 10 releases in ~2 years
  - 3.4k commits
  - 120+ contributors, 300 forks
- Major features:
  - Declarative Fencing (2022)
  - Volume for WALs
  - Failover of physical Replication Slots
  - Backup from a Standby (2023)
  - Declarative Hibernation
  - Declarative Role Management
  - Volumes for Tablespaces
  - Volume Snapshot backup and recovery
  - Distributed topologies (2024)
  - Managed services

About CloudNativePG

- “Level 5”, Production ready
  - EDB , IBM Cloud Pak, Google Cloud, Azure, Tembo, …
- Open source (May 2022)
  - Apache License 2.0
  - Vendor neutral, openly governed, always free
  - Originally created by EDB (2019 by 2ndQuadrant)
- Multiple installation methods:
  - K8s manifests
  - Helm chart
  - OperatorHub.io (OLM)
- >59M downloads on Github (~2 years)
- Now CloudNativePG is Officially CNCF sanbox project

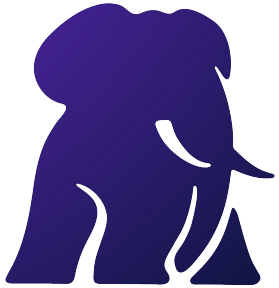




github.com/cloudnative-pg

The CloudNativePG Ecosystem

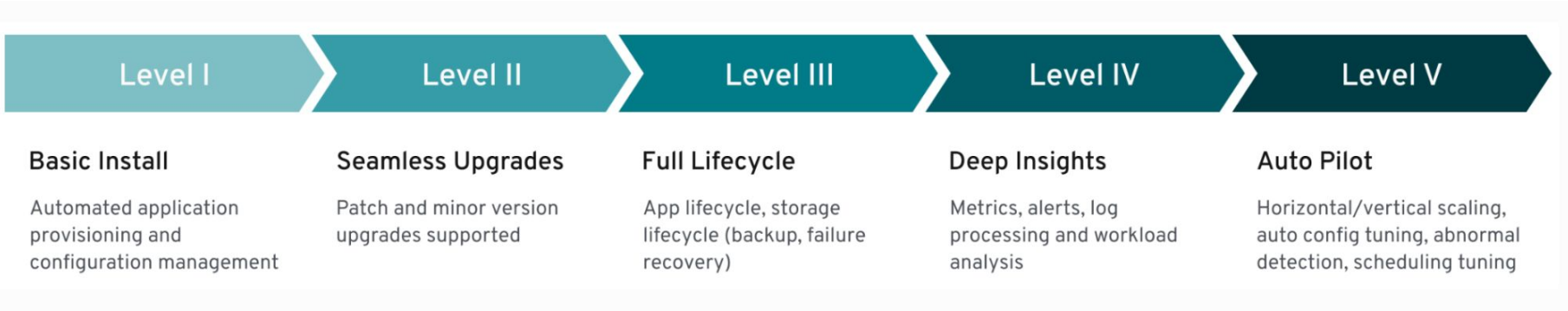
- The CloudNativePG operator is the main project (component)
  - 7 Maintainers (6 from EDB, 1 from Upbound)
  - Multiple component owners (contributors)
- Several components, with their respective owners:
  - Operand container images:
    - PostgreSQL with pgvector and pgaudit
    - PostgreSQL with PostGIS
    - PostgreSQL trunk (what will be in PostgreSQL 18 next year)
    - PgBouncer
  - Charts (operator and clusters)
  - Grafana Dashboard
  - Website
  - CNPG Playground
  - CNPG-I and official plugins (WIP)



CloudNativePG

github.com/cloudnative-pg

CloudNativePG capability levels



The PostgreSQL `Cluster` resource

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: clapton
spec:
  instances: 3
  imageName: ghcr.io/cloudnative-pg/postgresql:17.2
  affinity:
    nodeSelector:
      workload: postgres
  storage:
    size: 40Gi
  walStorage:
    size: 10Gi
```

(Physical) Continuous backup

Guaranteed RPO from the first available backup to the latest archived WAL (not a snapshot)

WAL archive

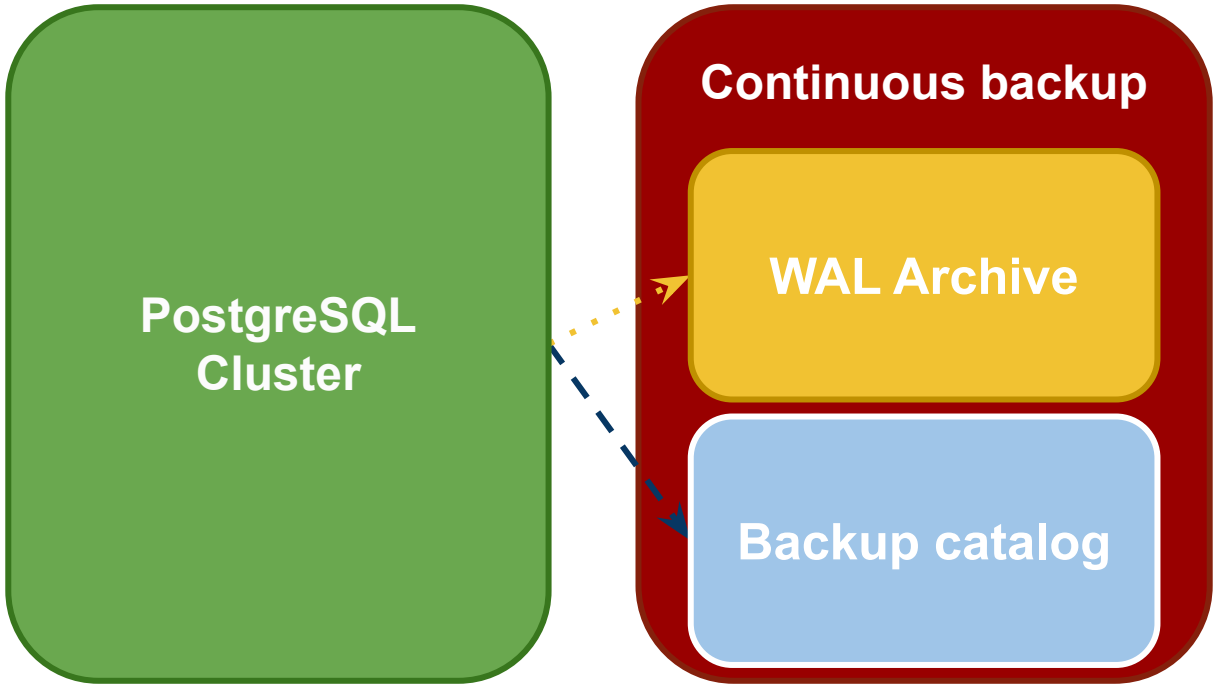
- Currently only via object store (Barman Cloud)
- WAL files are archived max every 5 minutes (RPO <= 5 min)

Physical base backups

- From the primary or a standby
- Scheduled or on demand
- On object stores (Barman Cloud)
  - Hot (online)
- As Kubernetes Volume Snapshots
  - Hot (online) and cold (offline)
  - Transparent incremental/differential backup

Future: generic plug-in interface (CNPG-I)

Continuous backup



Recovery

Full or to a specific point

Bootstrap method

- Copy from an available physical base backup
- Apply WAL files (REDO logs) until you reach the target

Full recovery

- Without a target (until the end of the WAL)
- Can be used to setup a physical replica
  - Replica cluster, even delayed
  - “Continuous recovery”

Point-In-Time Recovery

- Up to a given time or transaction
- When reaching the target, the cluster promotes itself

RTO vs RPO

- RPO is the Recovery Point Objective:

  the maximum amount of data (measured in terms of time) that can be lost in case of a disaster.

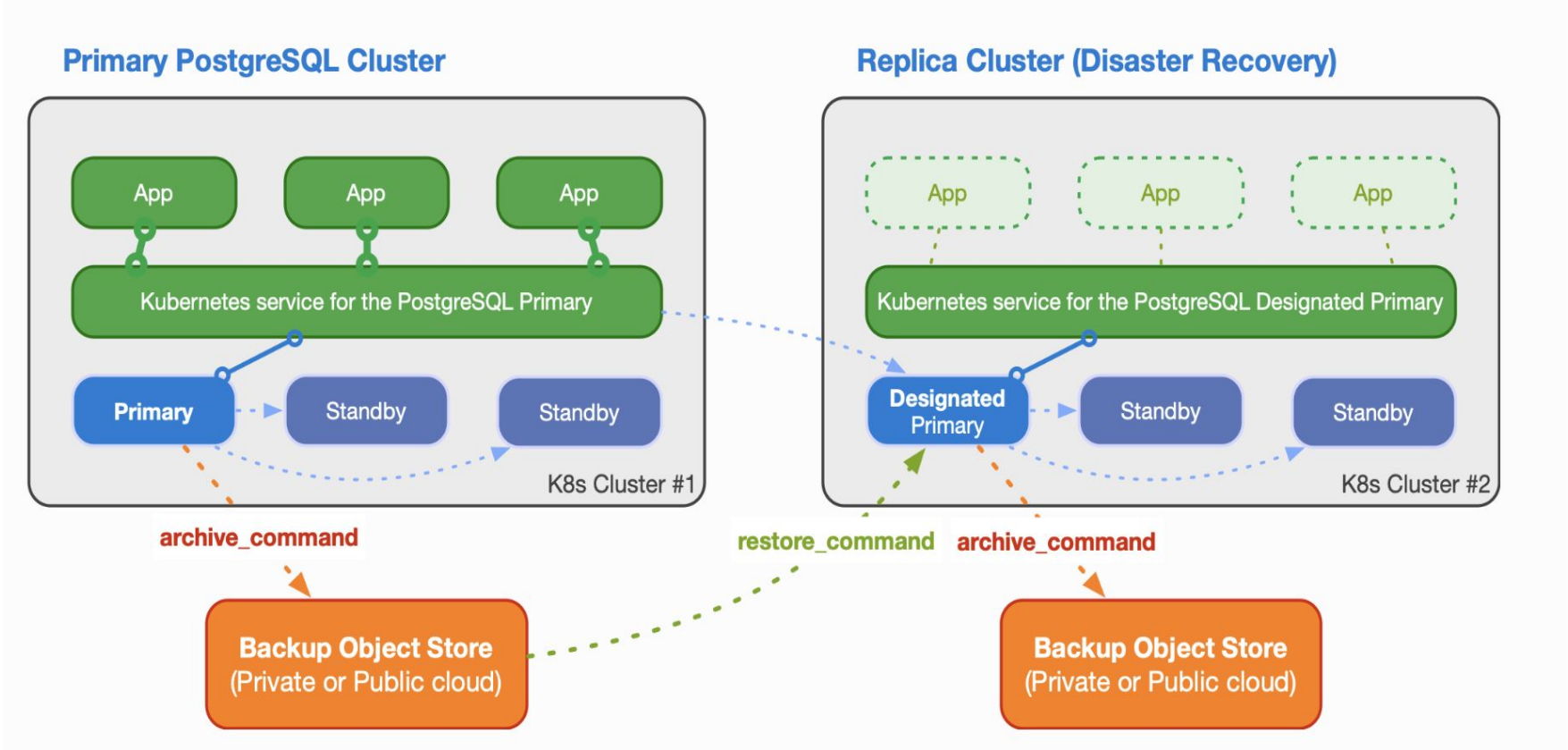
  In PostgreSQL clusters with transaction log archiving and point-in-time recovery, this is a function of how frequently WALs are shipped to the backup archive.

  By default, Cloud Native PostgreSQL is configured to close a WAL file after a maximum of 5 minutes (“archive_timeout” options).

- RTO is the Recovery Time Objective:

  the maximum amount of time acceptable for a restoration to take. RTO can be extremely complicated, depending on the nature / scope of a disaster, hardware availability, snapshot frequency, etc.

Replica cluster



Storage

For PostgreSQL data files

- Requires a volume with a file system on top of it, mounted inside a pod
- PVC with Filesystem volume mode
- PVC with ReadWriteOnce access mode

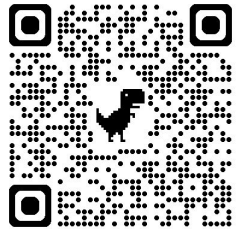
For Backup data files

- WAL archive requires object stores
  - Future: PVCs (CNPG-I plugin)
- Backup catalog
  - Object stores
  - VolumeSnapshots
  - Hybrid (both object stores and VolumeSnapshots)
  - Future: PVCs (CNPG-I plugin)

Direct management of Persistent Volume Claims (PVCs)

CloudNativePG is one of the few database operators that doesn’t use Statefulsets

- Custom controller that directly manages PVCs instead of using Statefulsets
- Why?
  - `PGDATA` (database files) is the heart of a PostgreSQL database
  - Volumes for PGDATA, WAL files and tablespaces
    - They need to be coherent (PVC Group concept)
  - Online Resizing of volumes
  - Understanding of the primary/standby architecture in PostgreSQL
    - Rolling updates
    - Kubernetes upgrades
    - Management of standby sensitive parameters (e.g. max_connections)
  - Advanced features made available by CSI for PVCs:
    - Online Resizing
    - PVC Cloning
    - Volume Snapshot backup and recovery
    - Volume Group Snapshot backup and recovery (alpha in K8s, with our active contribution)



Monitoring in CloudNativePG

- Metrics
  - ➢ Native prometheus exporter for the operator
  - ➢ Native prometheus exported for each postgres instance

- Logs
  - ➢ Native support for the kubernetes events
  - ➢ Direct stdout logging of operator and PostgreSQL logs in JSON format

Offline major version upgrades

- Offline major version upgrade
  - ➢ Leveraging logical backup methods like pg_dump and pg_restore

Limitations:

- ➢ Does not capture changes made to the origin database once the process begins
- ➢ Halt write operations on the origin's primary database until the migration is complete

Readiness:

- ➢ Verify that the import or upgrade processes function correctly on a new cluster, ensuring compatibility with applications on the updated Postgres version.

Tolerance:

- ➢ Accurately estimate the cutover time, primarily based on import duration

Online major version upgrade using CloudNativePG

- Online major version upgrade
  - ➢ Leveraging PostgreSQL's native logical replication to minimize cutover downtime to nearly zero.

Limitations of Logical replication:

- ➢ Replication of Data Definition Language (DDL), such as *CREATE TABLE* and *ALTER TABLE*, does not occur.
- ➢ Sequences are not replicated.
- ➢ Large objects (LOBs) are excluded from replication.

With CloudnativePG:

- ➢ Managing database schema changes, as DDL is not replicated.
- ➢ Avoid schema modifications during upgrades or apply changes manually if needed.
- ➢ Updating sequences as part of the cutover process.

How to achieve online upgrade using CloudNativePG

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg13
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:13
  enableSuperuserAccess: false
  instances: 1
  storage:
    size: 1Gi
  bootstrap:
    initdb:
      dataChecksums: true
      walSegmentSize: 32
      postInitApplicationSQL:
      - CREATE TABLE songs (id SERIAL PRIMARY KEY, title TEXT)
      - ALTER TABLE songs OWNER TO app
      - INSERT INTO songs(title) VALUES ('Back in black')
      - CREATE PUBLICATION migrate FOR TABLE songs
  managed:
    roles:
    - name: user1
      ensure: present
      comment: User for logical replication connections
      login: true
      replication: true
      inRoles:
      - app
      passwordSecret:
        name: user1
  postgresql:
    pg_hba:
    - hostssl replication user1 10.0.0.0/8 md5
```

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: pg16
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:16
  bootstrap:
    initdb:
      import:
        schemaOnly: true
        type: microservice
        databases:
        - app
        source:
          externalCluster: pg13
        postImportApplicationSQL:
        - CREATE SUBSCRIPTION migrate CONNECTION 'host=pg13-rw
user=user1 dbname=app sslmode=require password=user1pwd'
PUBLICATION migrate
  externalClusters:
  - name: pg13
    connectionParameters:
      # Use the correct IP or host name for the source database
      host: pg13-rw
      user: user1
      dbname: postgres
    password:
      name: user1
      key: password
```

Upgrade: Fully declarative way using CloudNativePG

Draft PR for community to validate:

https://github.com/cloudnative-pg/cloudnative-pg/pull/6664

What user have to do:

- Users can specify a new image with a higher major version

Process of upgrade:

- Hibernating the cluster which will delete cluster pod to ensure no write to data directory
- Create upgrade job on primary instance
- Job will use "***pg_upgrade --link***" to efficiently migrate the data
- Once the migration is complete, all non-primary instances will be removed and new standbys are cloned as per the desired instance.

Thank you

All the Maintainers, contributors, users and community members of CloudNativePG

Want to contribute?
https://github.com/cloudnative-pg/cloudnative-pg/blob/main/CONTRIBUTING.md

EDB
For generously donating the intellectual property for this project and for their decades of invaluable contributions to the broader PostgreSQL ecosystem.

Time for Practice

What is the requirements for the setup?

Laptop: Sufficient RAM, CPU, and disk to run Kubernetes in Docker (Kind) and any operating system that supports it.
Software: Students must come to the training session with the following software already installed, running the latest available version at the time of training:

- Kind (please follow instructions at kind.sigs.k8s.io/docs/user/quick-start)
- Kubectl
- The cnpg plugin for Kubectl (https://cloudnative-pg.io/documentation/1.25/kubectl-plugin/)

Additional requirements

- curl
- jq
- stern
- kubectx
- Kubectl plugins:
  - view-secret
  - view-cert

Exercise #1 **CloudNativePG Installation**

- Create kind cluster using (kind should be installed in local machine)

  “kind create cluster --name cnpg”

  // get kind clusters

  kind get clusters

- Deploy the operator and ensure it is installed on the control plane node.

  kubectl apply --server-side -f

https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/release-1.25/releases/cnpg-1.25.0.yaml

- Make sure CNP operator gets installed using

  kubectl get deployment -n cnpg-system cnpg-controller-manager

- Install CNPG kubectl plugin

  *curl -sSfL *

  *https://github.com/cloudnative-pg/cloudnative-pg/raw/main/hack/install-cnpg-plugin.sh | *

  sudo sh -s -- -b /usr/local/bin

- Validate version of CNPG kubectl plugin installed

  kubectl cnpg version

Exercise #2 **Bootstrap new PostgreSQL cluster**

Open the “Quickstart” page from the documentation and download the `cluster-example.yaml` file from the website and save it locally. For example, you can use `curl`:

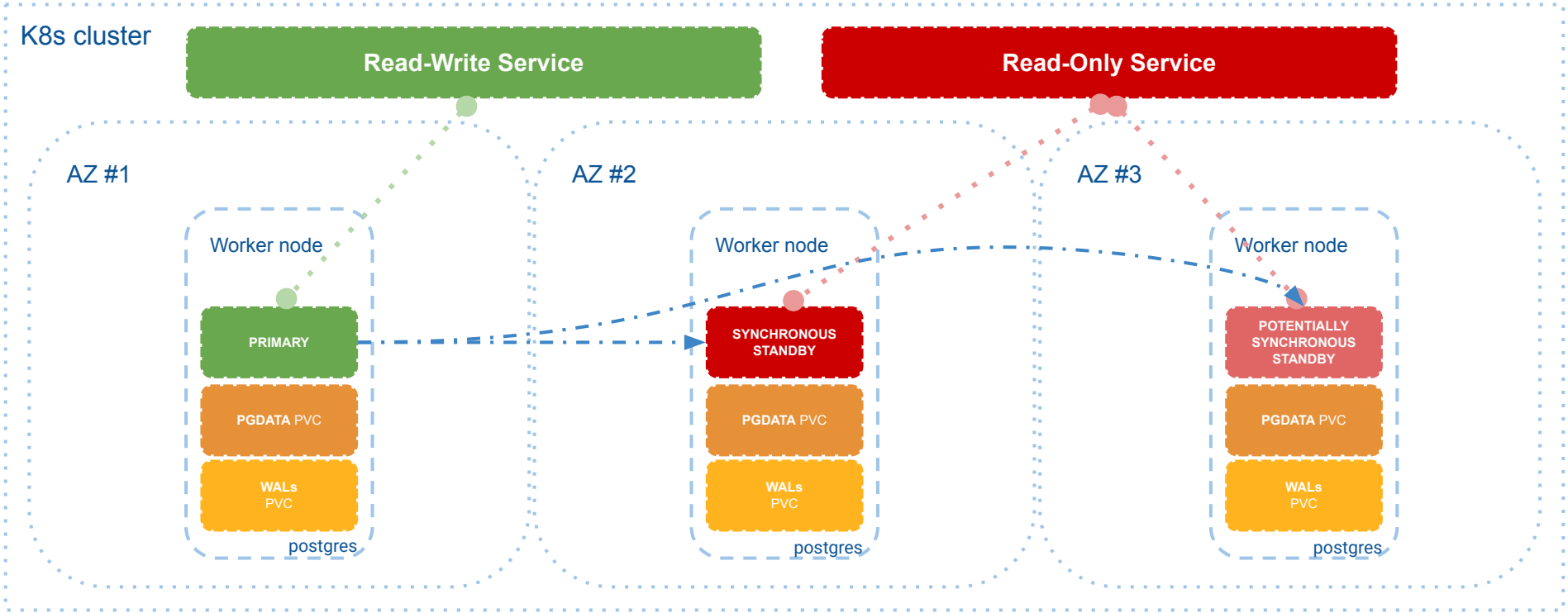
$ curl -LO https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/refs/heads/main/docs/src/samples/cluster-example.yaml

Open the `cluster-example.yaml` file and inspect it.
$ kubectl apply -f cluster-example.yaml

Get the YAML definition from K8s.
$ kubectl get cluster -o cluster-example

It will be different from the one we applied. Why?

Highly Available PostgreSQL Cluster



Exercise #3 **Explore generated kubernetes resources**

```
$ kubectl get cluster
$ kubectl get pod
$ kubectl get service
$ kubectl get pvc
$ kubectl get pdb

# Optional
$ kubectl get role
$ kubectl get rolebinding
$ kubectl get sa
```

Contd….

// inspect the users

```
kubectl exec -ti -c postgres cluster-example-1 -- psql -c '\du'
```

// inspect the databases

```
kubectl exec -ti -c postgres cluster-example-1 -- psql -c '\l'
```

Contd….

$ kubectl get secrets -A

NAME	TYPE
cluster-example-app	kubernetes.io/basic-auth
cluster-example-ca	Opaque
cluster-example-replication	kubernetes.io/tls
cluster-example-server	kubernetes.io/tls

cluster-example-app: secret containing information about the app user in the PostgreSQL cluster, including credentials
cluster-example-ca: secret containing the Certification Authority (CA) used by the operator for emitting TLS certificates
cluster-example-replication: secret with the TLS client certificate for streaming replication in the HA cluster
cluster-example-server: secret with the TLS certificate of the PostgreSQL server

Contd....

Inspect CA certificate

$ kubectl get secret cluster-example-ca -o jsonpath="{.data['ca\.crt']}" | base64 -d | openssl x509 -text -noout

Server certificate signed with above CA

$ kubectl get secret cluster-example-server -o jsonpath="{.data['tls\.crt']}" | base64 -d | openssl x509 -text -noout

Certificate for the streaming_replica user

$ kubectl get secret cluster-example-replication -o jsonpath="{.data['tls\.crt']}" | base64 -d | openssl x509 -text -noout

Check, how standby is connecting to the primary cluster

$ kubectl exec -ti -c postgres cluster-example-2 -- psql -qAt -c 'SHOW primary_conninfo'

host=cluster-example-rw
user=streaming_replica
port=5432
sslkey=/controller/certificates/streaming_replica.key
sslcert=/controller/certificates/streaming_replica.crt
sslrootcert=/controller/certificates/server-ca.crt
application_name=cluster-example-2
sslmode=verify-ca

Exercise #4 **CNPG CRD**

- Try changing the image from ghcr.io/cloudnative-pg/postgresql:16.1 to: (ghcr.io/cloudnative-pg/postgresql:15).

  Is that possible? Why?

- Try changing the ***shared_buffers*** parameters to “thistest”.

  Is that possible? Why?

Exercise #5 **Manual PostgreSQL administration activities**

$ kubectl log cluster-example-1 | jq

$ kubectl cnpg psql cluster-example

```
psql (16.1 (Debian 16.1-1.pgdg110+1), server 16.2 (Debian 16.2-1.pgdg110+2))
Type "help" for help.

postgres=# \! hostname
cluster-example-1
```

$ kubectl cnpg psql --replica cluster-example

Exercise #6 **Testing the controller**

Try deleting the -rw service with:

$ kubectl delete service cluster-example-rw

And get the list of services from Kubernetes:
$ kubectl get services

Why cluster-example-rw is still there? Is that the same service?

Try deleting cluster-example-2. What happened?

Try deleting cluster-example-1. What happened? Who is the primary now?

Promote manually cluster-example-2 with:
kubectl cnpg promote cluster-example 2

Exercise #7 **Configuring PostgreSQL**

```
$ kubectl get cluster cluster-example -o yaml |grep -A25 parameters:
    parameters:
      archive_mode: "on"
      archive_timeout: 5min
      dynamic_shared_memory_type: posix
      wal_keep_size: 512MB
      wal_receiver_timeout: 5s
      wal_sender_timeout: 5s
    syncReplicaElectionConstraint:
      enabled: false
  primaryUpdateMethod: restart
  primaryUpdateStrategy: unsupervised
```

1. Try setting ***log_directory*** with kubectl edit cluster cluster-example. Is that possible? Why?
2. What is the current value of ***shared_buffers***?
3. In a separate tab run kubectl get pod -w. Increase **shared_buffers**. What happened?
4. Decrease **shared_buffers**. What happened?

Exercise #8 **Scaling**

Cluster-example has three instances (one primary and two replicas). Scale it to 4 with:

$ kubectl scale cluster cluster-example –replicas=4

Scale it back to 3

Links

- https://cloudnative-pg.io/documentation/current/operator_capability_levels/#scale-up-and-down-of-replicas

Exercise #9 **Wreck it all**

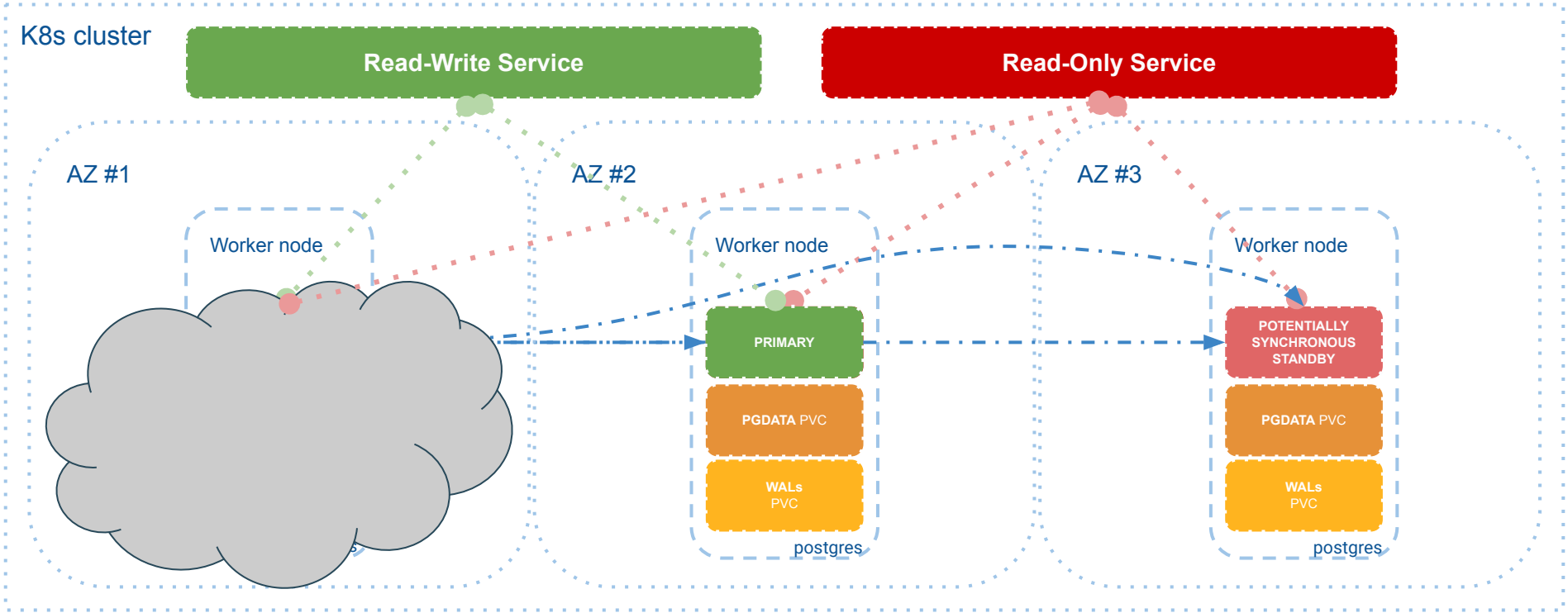
Remove the directory of the `postgres` database in the current primary:
kubectl exec cluster-example-<X> -- rm -rf /var/lib/postgresql/data/pgdata/base/5

What happened? Which Pod is the new primary?

$ kubectl exec -ti cluster-example-<Y> -- psql
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
psql (16.4 (Debian 16.4-1.pgdg110+2))
Type "help" for help.
postgres=# create table testtest (i integer);
CREATE TABLE
postgres=# insert into testtest (select generate_series(1,1000000));
INSERT 0 1000000
postgres=#

Destroy the Pod and its data with:
kubectl cnpg destroy cluster-example <X>

Automated failover (HA with very low RTO)



Exercise #10 **WAL Archiving**

While connected to the EU region, apply the example in examples/eu/cluster-dc-eu.yaml:

$ kubectl apply -f examples/eu/cluster-dc-eu.yaml

```
$ k get cluster cluster-dc-eu  -o yaml|grep -A20 backup:
  backup:
    barmanObjectStore:
      destinationPath: s3://backups/
      endpointURL: http://minio-eu:9000
      s3Credentials:
        accessKeyId:
          key: ACCESS_KEY_ID
          name: minio-eu
        secretAccessKey:
          key: ACCESS_SECRET_KEY
          name: minio-eu
      wal:
        compression: gzip
    target: prefer-standby
  bootstrap:
    initdb:
      database: app
      owner: app
```

Is it archiving WALs?

$ docker exec minio-eu ls /data/backups/cluster-dc-eu/wals/0000000100000000

000000010000000000000001.gz

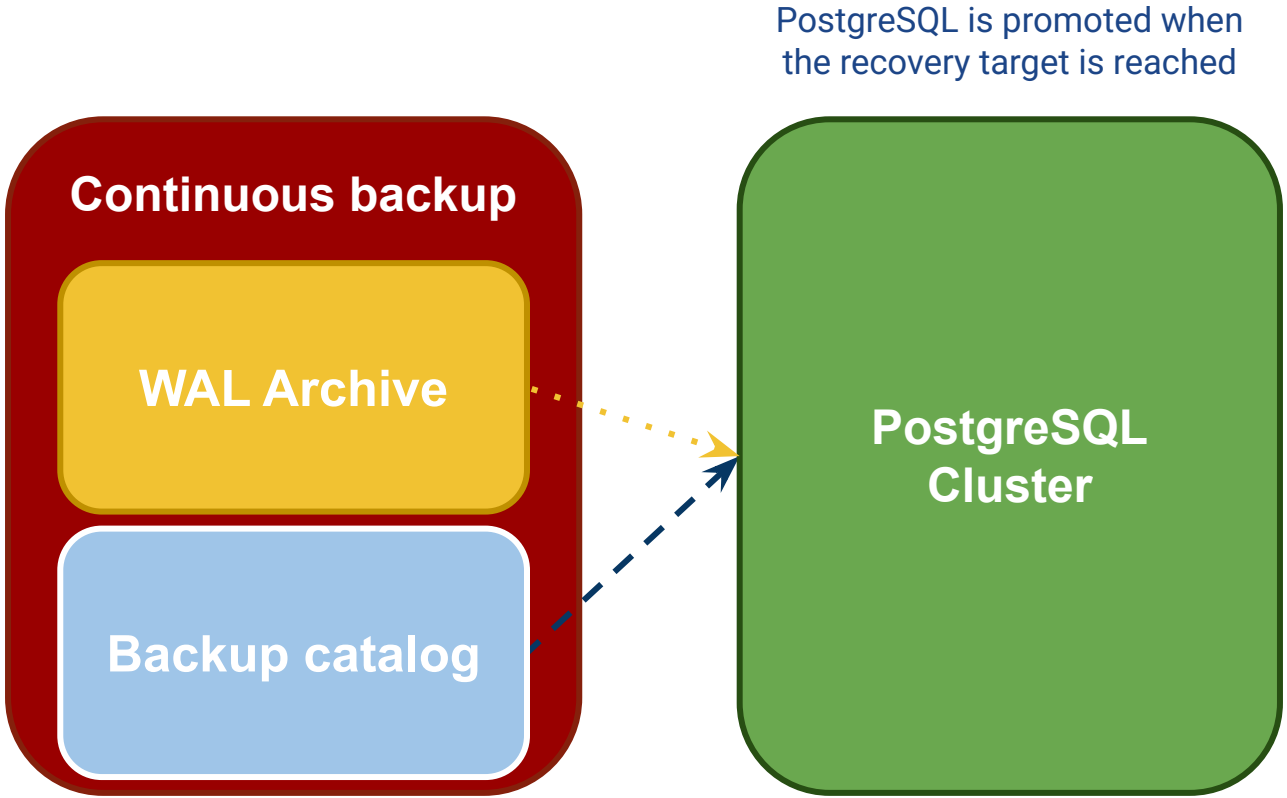
Exercise #11 **On Demand backups/PITR**

```
$ kubectl cnpg backup cluster-dc-eu
backup/cluster-dc-eu-20241007164438 created

$ kubectl get backup cluster-dc-eu-20241007164438
NAME                          AGE   CLUSTER       METHOD             PHASE      ERROR
cluster-dc-eu-20241007164438   15s   cluster-dc-eu   barmanObjectStore   completed

$ kubectl get backup cluster-dc-eu-20241007164438 -o yaml
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  creationTimestamp: "2024-10-07T14:44:38Z"
  generation: 1
  labels:
    cnpg.io/cluster: cluster-dc-eu
  name: cluster-dc-eu-20241007164438
  namespace: default
  resourceVersion: "4487"
  uid: 61eb23dd-737f-4e19-879f-15f0487bebc1
spec:
  cluster:
    name: cluster-dc-eu
  method: barmanObjectStore
```

Exercise #12 **Creating new cluster from backup**

```
$ kubectl apply -f - <<EOF
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: restore
spec:
  instances: 3
  storage:
    size: 1Gi
  bootstrap:
    recovery:
      source: cluster-dc-eu
  externalClusters:
  - name: cluster-dc-eu
    barmanObjectStore:
      serverName: cluster-dc-eu
      destinationPath: s3://backups/
      endpointURL: http://minio-eu:9000
      s3Credentials:
        accessKeyId:
          name: minio-eu
          key: ACCESS_KEY_ID
        secretAccessKey:
          name: minio-eu
          key: ACCESS_SECRET_KEY
      wal:
        compression: gzip
EOF
```

Full recovery or PITR



Exercise #13 **Metrics**

```
helm repo add prometheus-community \
  https://prometheus-community.github.io/helm-charts

helm upgrade --install \
  -f https://raw.githubusercontent.com/cloudnative-pg/cloudnative-pg/main/docs/src/samples/monitoring/kube-stack-config.yaml \
  prometheus-community \
  prometheus-community/kube-prometheus-stack
```

```
kubectl apply -f - <<EOF
---
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
m
etadata:
  name: cluster-with-metrics
spec:
  instances: 3

  storage:
    size: 1Gi

  monitoring:
    enablePodMonitor: true
EOF
```

kubectl port-forward svc/prometheus-community-kube-prometheus 9090
kubectl port-forward svc/prometheus-community-grafana 3000:80

Exercise #14 Hibernation and rehydration

Create a basic cluster

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
 name: cluster-example
spec:
 imagePullPolicy: Always
 instances: 3
 storage:
   storageClass: standard
   size: 1Gi
```

```
kubectl annotate cluster cluster-example --overwrite cnpg.io/hibernation=on
kubectl get cluster cluster-example -o "jsonpath={.status.conditions[?(.type==\"cnpg.io/hibernation\")]}"
kubectl annotate cluster cluster-example --overwrite cnpg.io/hibernation=off
```

Thank you