EDB POSTGRES AI

PostgreSQL on Kubernetes with CloudNativePG (CNPG)

Danish Khan
Neel Patel

Danish Khan

DoKC Ambassador
SDE at EDB
CloudNativePG Contributor

github.com/danishedb
danish.khan@enterprisedb.com

EDB POSTGRES AI

Neel Patel

Principal Software Engineer at EnterpriseDB

With EnterpriseDB since ~12 years

Open source contributor for various tools around PostgreSQL for monitoring and management

Currently I am focusing on EDB Postgres AI Hybrid Manager which leverage Kubernetes for orchestration and CloudNativePG for managing the lifecycle of the PostgreSQL

https://github.com/neel5481
neel.patel@enterprisedb.com

Agenda

- Introduction to CloudNativePG (CNPG)
- Installation of the kubectl plugin for CNPG
- First deployment of a CNPG cluster
- PostgreSQL configuration, databases and roles management
- PostgreSQL Extensions
- Database import
- Monitoring of the PostgreSQL server
- Incident simulation (failover showcase)
- Log reading
- Upgrade of the Operator and the PostgreSQL version (minor, and major)
- Setup and execution of the first backup
- Restore from a backup

Useful Contents

To speed up the process and let you follow the workshop seamlessly:

- **COMMANDS.md**
  - contains the list of commands in the order of appearance in this course

    commands.md

  - http://bit.ly/4d0BCsP

CloudNativePG

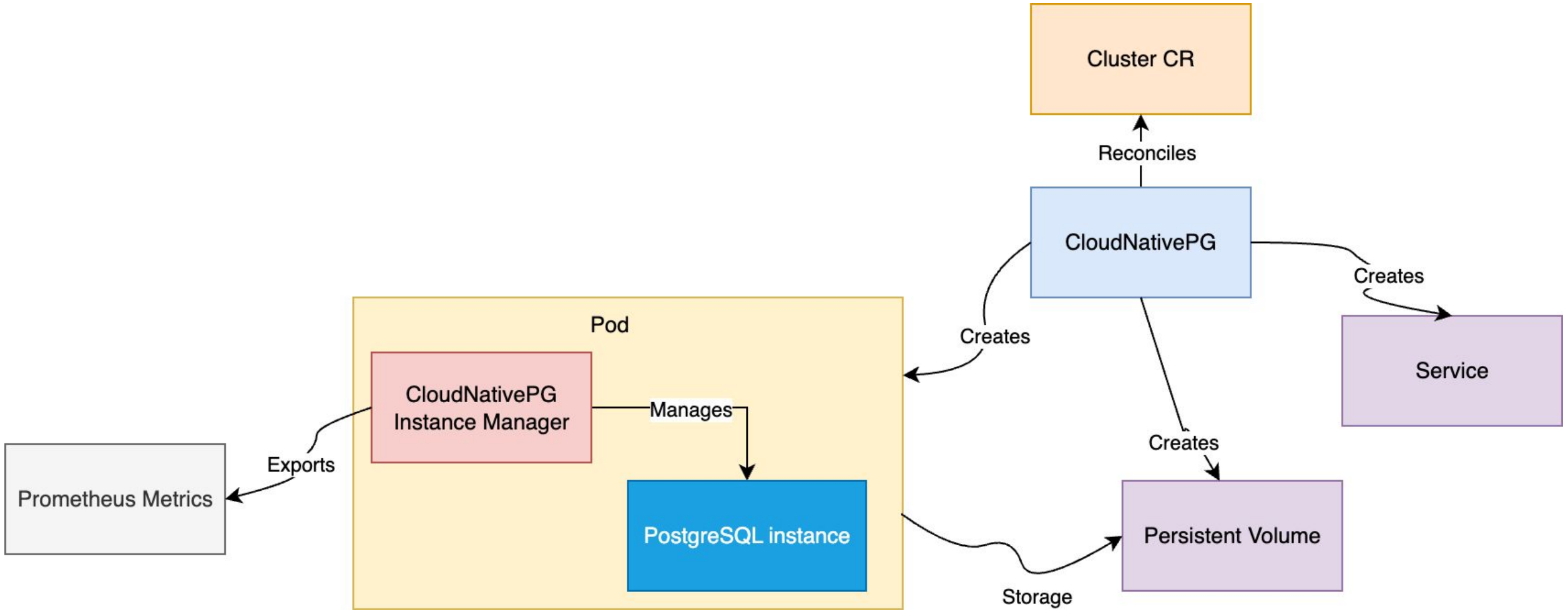
https://github.com/cloudnative-pg/cloudnative-pg/blob/main/ADOPTERS.md

CloudNativePG

- A CNCF **Sandbox** Kubernetes Operator
- **Manages** the whole lifecycle of a PostgreSQL cluster
- Full-Spectrum Operations - From Day 0 to Day 2
- **Declarative** Configuration
- Specialized tool integration
- Runs on **many** Kubernetes distribution:
  - Vanilla
  - OpenShift
  - Cloud Service Providers K8S env
  - kind
  - k3d

https://cloudnative-pg.io/
https://github.com/cloudnative-pg/cloudnative-pg

CloudNativePG: under the hood



What is CNPG instance manager?

- CloudNativePG does not rely on an external tool for failover management.
- CNPG has native instance manager which takes care of the entire lifecycle of the postmaster process.

Pod Start

└→ Instance Manager (PID 1) launches

└→ [Startup Probe] pg_isready loop — wait for PostgreSQL

└→ [Liveness Probe] monitor health continuously

└→ [Readiness Probe] allow traffic + failover

└→ [Shutdown] smart → fast (2-phase)

How does it work?

- **Manages** Operands
  - PostgreSQL container images
- **Extends** Kubernetes with:
  - **Controllers**
    - Takes advantage of the Kubernetes API to reconcile the PostgreSQL cluster state
  - **Custom Resource Definitions**
    - Backups
    - Clusters
    - ImageCatalogs
    - Poolers
    - Databases
    - …

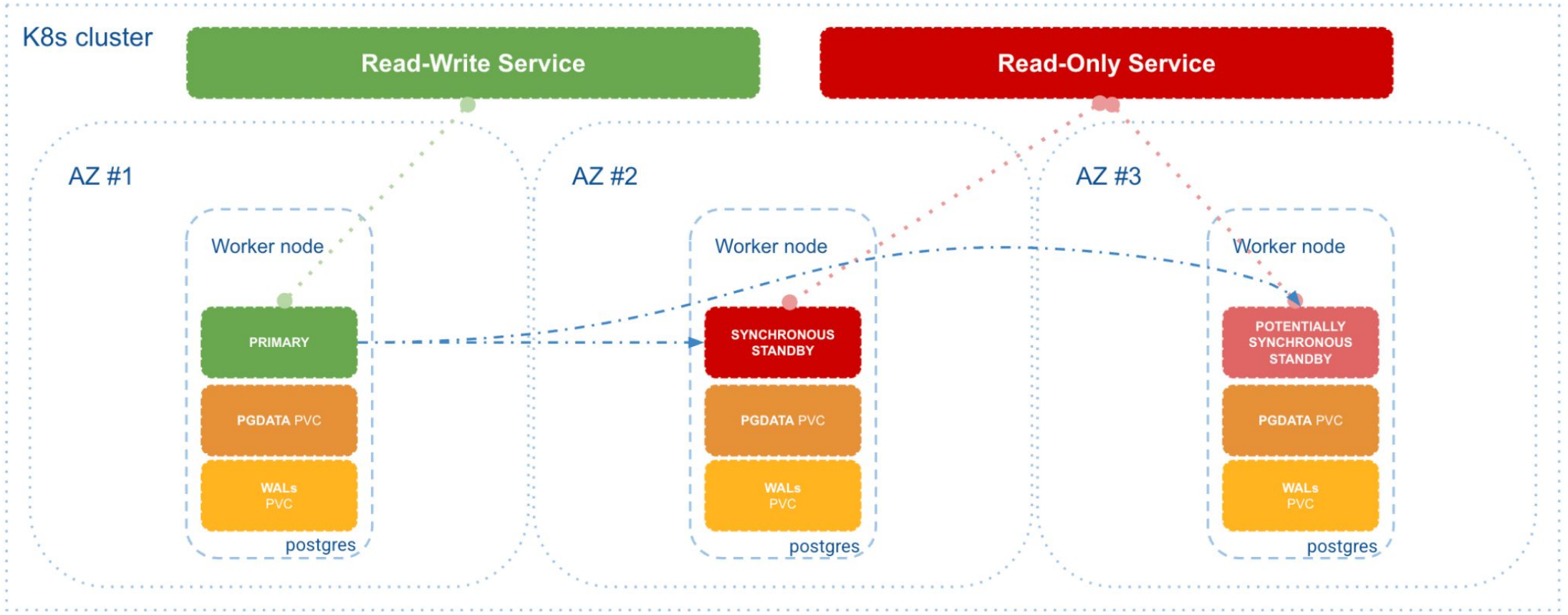
https://cloudnative-pg.io/docs/devel/

Minimum required* YAML definition:

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

  storage:
    size: 1Gi
```

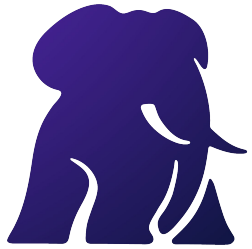
* **Convention over configuration** paradigm: all the other parameters are set by default.

Architecture



CloudNativePG capability levels

CloudNativePG - Main Features



- **High Availability** and **Self-Healing**
- Support for **local PVCs**
- Managed **services** for **rw** and **ro** workloads
- **Continuous backup** (including snapshots)
- **Point In Time Recovery** (incl. snapshots)
- **Scale up/down** of read-only replicas
- "**Security by default**", including mTLS
- Native **Prometheus exporter**
- **Logging** to stdout in **JSON** format
- **Rolling updates**, incl. **minor Postgres** releases
- **Synchronous** replication
- Online **import** of Postgres **databases**
- Separate **volume** for WALs
- Postgres tablespaces
- **Replica clusters** and **distributed topologies**
- Declarative **role management**
- Declarative **hibernation** and **fencing**
- **CNPG-I** - interface to develop CNPG **plugins**
- Connection **pooling**
- **Postgres extensions** (pgvector, PostGIS, …)

How to install it?

Setup environment

Prerequisites

- Docker
- Kind
- kubectl
- CNPG Plugin
- git

Then:

- Set limits with `sysctl` (linux)
- create a k8s kind cluster
- Test the kubectl commands

```
$ # For Linux users
$ sudo sysctl \
  fs.inotify.max_user_watches=524288 \
  fs.inotify.max_user_instances=512


$ cat <<EOF | kind create cluster --name pgconf --config=-
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
nodes:
- role: control-plane
- role: worker
featureGates:
  ImageVolume: true
EOF



$ kubectl get pods -A
$ kubectl get nodes
```

Using the manifest

1. Install the CNPG **Operator v1.29.0**
2. Check for resources
    - Analyse the resources created
        - Deployment
        - Pod

```
$ kubectl apply --server-side -f
https://raw.githubusercontent.com/cloudnative-pg/cloudn
ative-pg/release-1.29/releases/cnpg-1.29.0.yaml

$ kubectl get deployments,pods -n cnpg-system
```

https://cloudnative-pg.io/docs/devel/installation_upgrade

Using the CNPG Plugin

1. Install the **kubectl plugin** for CNPG
2. Familiarize yourself with its commands:
   - `--help`
3. Check the `install` command

```
$ curl -sSfL \
https://github.com/cloudnative-pg/cloudnative-pg/raw/ma
in/hack/install-cnpg-plugin.sh | \
  sudo sh -s -- -b /usr/local/bin


$ kubectl cnpg --help


$ kubectl cnpg install generate \
  --control-plane \
  --version 1.29.0 \
  | kubectl apply -f - --server-side
```

https://cloudnative-pg.io/docs/devel/kubectl-plugin/

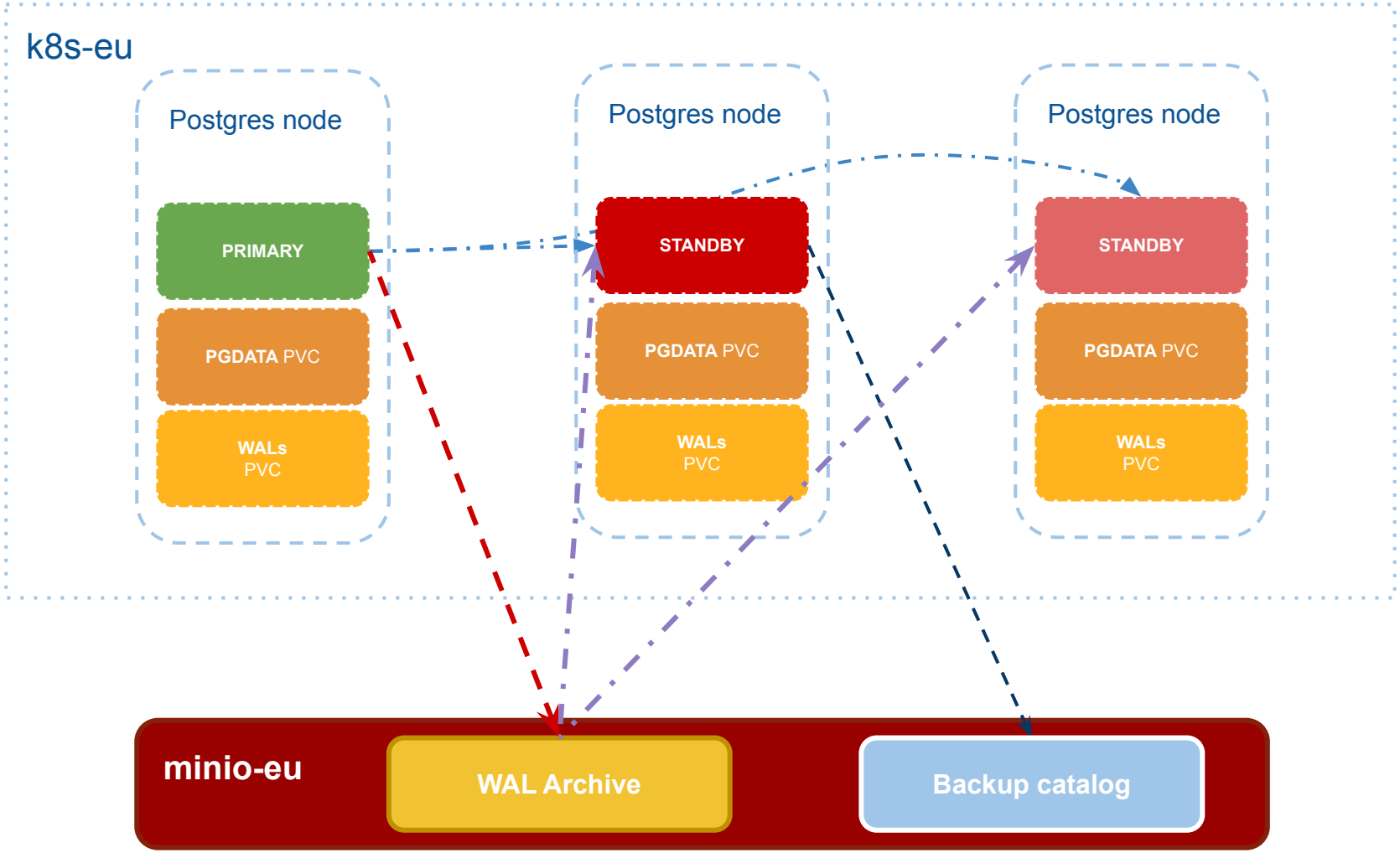
Deploy the first Cluster

1. Create a YAML file with the basic CNPG cluster definition
2. Open a new terminal window to monitor resources
3. Apply the cluster manifest
4. Check for the created resources:
   - pods
   - services
   - pvc

https://cloudnative-pg.io/docs/devel/quickstart/#part-3-deploy-a-postgresql-cluster

```
$ cat <<EOF > ./cluster-example.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example
spec:
  instances: 3
  storage:
    size: 1Gi
EOF

$ kubectl get pods -w

$ kubectl apply -f cluster-example.yaml

$ kubectl get clusters,pods,pvc,svc,ep

$ kubectl cnpg status cluster-example
```

CNPG Cluster Architecture



How to manage PostgreSQL **roles**?

Problem - Managing Postgres roles without CNPG

Source of truth - Lives only inside the DB

Disaster recovery - Roles not in pg_dump as after a restore, roles are gone, app can't connect

Drift - Dev runs ALTER ROLE temporarily and never reverted and production incident

Password rotation - Someone manually runs ALTER ROLE after secret changes or forgets to revert

Multi-cluster - dev / staging / prod / DR all have different roles and nobody knows which is right.

Solution - Declarative role management with CNPG

Roles as code, Continuously enforced, Passwords in Secrets.

Drift - Operator reverts manual changes every reconcile cycle

Password rotation - Secret updated and ALTER ROLE runs automatically

Post-DR recovery Roles re-applied from spec, zero manual steps

Offboarding - ensure: absent , dropped from all clusters

Cross-cluster - Same YAML - identical roles on dev / staging / prod / DR

Declarative Roles

- Defines roles in a **declarative manner**
- Manages **full lifecycle** of Roles
- Uses PostgreSQL functions:
  - `CREATE ROLE`
  - `ALTER ROLE`
- Requires **human intervention** in case of errors
- Passwords:
  - Uses **secrets reference** for passwords
  - Empty password = no password
  - **Certificates** are preferable

https://cloudnative-pg.io/docs/devel/declarative_role_management

```
status:
  […snipped…]
  managedRolesStatus:
    byStatus:
      not-managed:
      - app
      pending-reconciliation:
      - pguser1
      - pguser2
      reconciled:
      - pguser3
      reserved:
      - postgres
      - streaming_replica
    cannotReconcile:
      pguser1:
      - 'could not perform DELETE on role pguser1:
owner of database inferno'
      pguser2:
      - 'could not perform UPDATE_MEMBERSHIPS on role
pguser2: role "poets" does not exist'
```

Declarative Roles

1. Check the current roles in Postgres
2. Edit the cluster manifest
3. Add the roles you want
4. Apply the `cluster-example.yaml` manifest
5. Check the presence of wanted roles in Postgres

```
$ kubectl cnpg psql cluster-example -- app -c “\duS+”

$ # Edit cluster-example.yaml and add the following:
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
spec:
[...]
  managed:
    roles:
    - name: pguser1
      ensure: present
      comment: Postgres User
      login: true
      superuser: false
      inRoles:
        - pg_monitor
        - pg_signal_backend
[...]

$ kubectl apply -f cluster-example.yaml

$ kubectl cnpg psql cluster-example -- app -c “\duS+”
```

Declarative Roles

1. Manually change the role in Postgres
2. Check for the role changes
3. Trigger a reconciliation loop
4. Check the role

```
$ kubectl get cluster cluster-example \
  -o jsonpath='{.status.managedRolesStatus}' | jq .

$ kubectl cnpg psql cluster-example -- app \
  -c “ALTER ROLE pguser1 NOLOGIN”

$ kubectl cnpg psql cluster-example -- app -c “\duS+
pguser1”

$ kubectl cnpg reload cluster-example

$ kubectl cnpg psql cluster-example -- app -c “\duS+
pguser1”
```

How to manage PostgreSQL **databases**?

EDB POSTGRES AI

Declarative Databases

- Defines databases in a **declarative manner**
- Separate **Custom Resource Definition (CRD)**
- **Manages** a database with optionals:
  - extensions
  - schema
  - FDW
- Two different database **deletion methods**:
  - Deleting the Database CRD with `databaseReclaimPolicy`:
    - `retain`
    - `delete`
  - Declaratively with `ensure: absent`

https://cloudnative-pg.io/docs/devel/declarative_database_management

```
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  generation: 1
  name: cluster-example-one
spec:
spec:
  databaseReclaimPolicy: delete
  cluster:
    name: cluster-example
  name: one
  owner: app
status:
  observedGeneration: 1
  applied: true
```

Declarative Databases

1. Check for the current databases in Postgres
2. Create a Database resource in a YAML file
3. Apply the `database_one.yaml` manifest
4. Check for the changes in Postgres

```
$ kubectl cnpg psql cluster-example -- -c “\l”

$ # Create a new Database manifest:

$ cat <<EOF > database_one.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: cluster-example-one
spec:
spec:
  databaseReclaimPolicy: retain
  name: one
  owner: app
  cluster:
    name: cluster-example
EOF

$ kubectl apply -f database_one.yaml

$ kubectl cnpg psql cluster-example -- -c “\l”
```

Declarative Databases

1. Delete the database resource
2. Check for the changes in Postgres
3. Edit the database manifest
4. Apply the `database.yaml` manifest
5. Check for the changes in Postgres again

```
$ kubectl delete -f database_one.yaml

$ kubectl cnpg psql cluster-example -- -c “\l”

$ cat <<EOF > database_one.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: cluster-example-one
spec:
spec:
  databaseReclaimPolicy: retain
  name: one
  owner: app
  ensure: absent
  cluster:
    name: cluster-example
EOF

$ kubectl apply -f database_one.yaml

$ kubectl cnpg psql cluster-example -- -c “\l”
```

How to manage PostgreSQL **extensions**?

EDB POSTGRES AI

Problem - Extensions were baked into the postgres image

- Every extension (pgvector, PostGIS, etc.) had to be compiled into the PostgreSQL container image at build time.
- Forced to maintain large, custom images, hard to update, hard to secure
- Updating one extension meant rebuilding and redeploying the entire image
- CVE in one extension? Rebuild everything.
- Kubernetes pods are immutable so you can't install packages into a running container

Solution: Image Volume Extension

Root cause: PostgreSQL only looked in one place for extensions - inside the container. No way to load from external paths.

Features	Introduced in
extension_control_path - PostgreSQL can look in multiple directories for extensions	PostgreSQL v18
ImageVolume - mount an OCI image as a read-only volume directly into a running pod	Kubernetes 1.33+
Declarative extensions: API in CloudNativePG	CloudNativePG 1.27+

How it works now

- **Before:** Base image + pgvector + PostGIS + .. = large custom image (~600MB+)
- **After**: Small base image (~260MB) + tiny extension images mounted at runtime
- CloudNativePG automatically mounts the image as an ImageVolume
- Sets `extension_control_path`, and runs the SQL

```
postgresql:
extensions:
- name: pgvector
  image:
    reference:
      ghcr.io/cloudnative-pg/pgvector:0.8.1-18-trixie
---
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: ext-example-db
spec:
  name: app
  owner: app
  cluster:
    name: ext-example
  extensions:
  - name: vector
    version: '0.8.1'
    ensure: present
```

Summary

	Before Image Volume	After Image Volume
Image Size	Hundreds of MBs (custom)	~250 MB base + ~KBs per extension
Add Extension	Rebuild entire image	Allows dynamically adding extension without rebuilding core postgres image
Update Extension	Rebuild + Redeploy all	Update core postgres and extension image independently
CVE in extension	Rebuild everything	Replace that extension image only

Declarative Extensions

- Manages extensions in a **declarative manner**
- List of extensions in the Database manifest
- Uses PostgreSQL functions:
  - `CREATE EXTENSION`
  - `DROP EXTENSION`
  - `ALTER EXTENSION` (limited)
- **Requires** extensions to be available in the PostgreSQL image
  - Single **heavy PG image** which contains required extensions files

https://cloudnative-pg.io/docs/devel/declarative_database_management#managing-extensions-in-a-database

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: ext-example
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:18-minimal-trixie
  instances: 1

  storage:
    size: 1Gi

  postgresql:
    extensions:
    - name: pgvector
      image:
        reference: ghcr.io/cloudnative-pg/pgvector:0.8.1-18-trixie
---
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: ext-example-app
spec:
  name: app
  owner: app
  cluster:
    name: ext-example
  extensions:
  - name: vector
    version: '0.8.1'
    ensure: present
```

Declarative Extensions

1. Create a Cluster YAML manifest
2. Apply the `ext-example.yaml` manifest
3. Check for the current extensions in Postgres

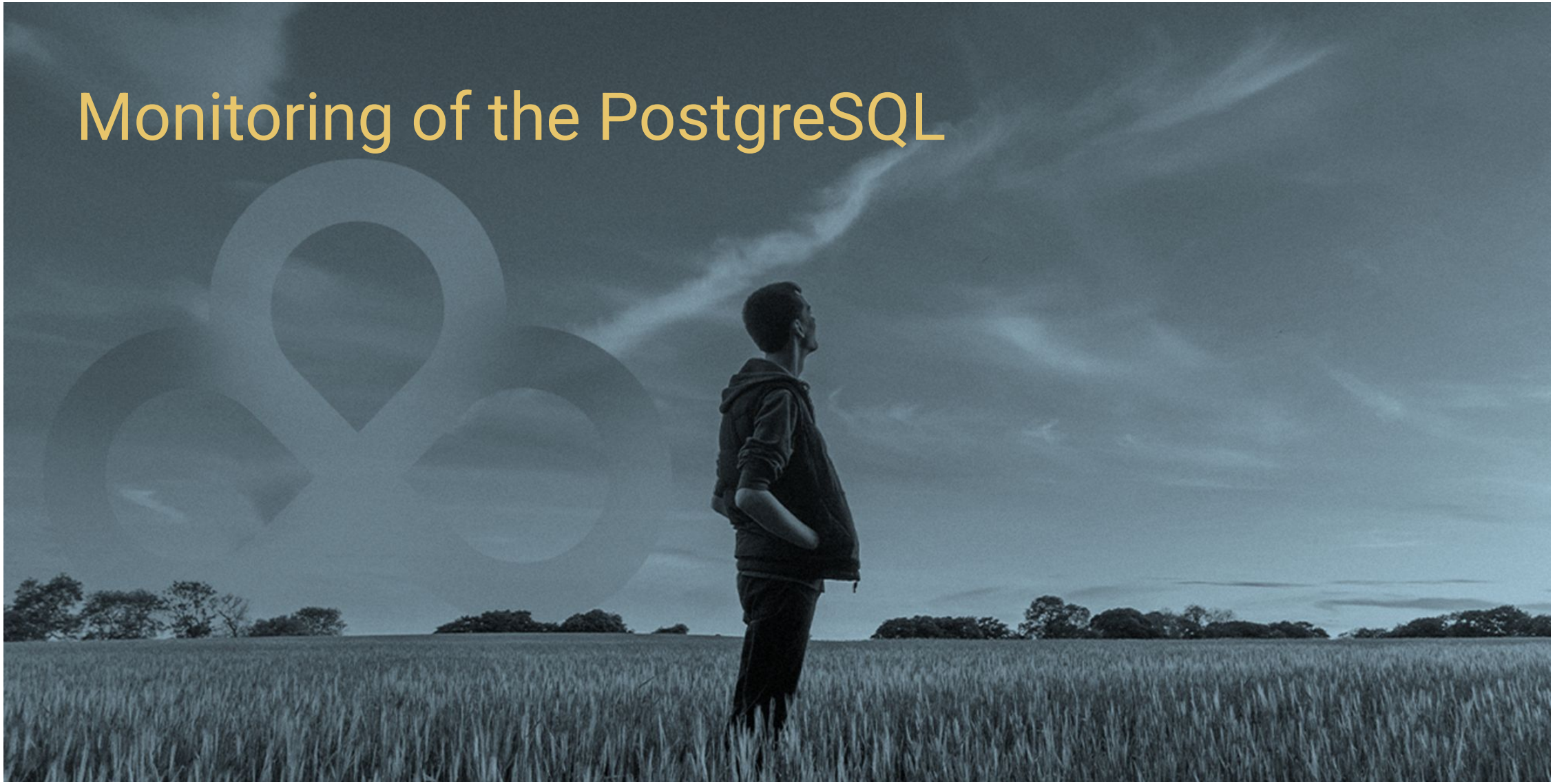
```
$ cat <<EOF > ext-example.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: image-example
spec:
  imageName:
ghcr.io/cloudnative-pg/postgresql:18-minimal-trixie
  instances: 1

  storage:
    size: 1Gi

  postgresql:
    extensions:
    - name: pgvector
      image:
        reference:
ghcr.io/cloudnative-pg/pgvector:0.8.1-18-trixie
EOF

$ kubectl apply -f ext-example.yaml

$ kubectl cnpg psql ext-example -- app -c "\dx"
```

Declarative Extensions

1. Create the `ext-example-db.yaml` manifest
2. Apply the `ext-example-db.yaml` manifest
3. Check for the changes in Postgres

```
$ cat <<EOF > ext-example-db.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: ext-example-app
spec:
  name: app
  owner: app
  cluster:
    name: ext-example
  extensions:
  - name: vector
    version: '0.8.1'
    ensure: present
EOF

$ kubectl apply -f ext-example-db.yaml

$ kubectl cnpg psql ext-example -- app -c “\dx”
```

Monitoring of the PostgreSQL



Postgres monitoring using CloudNativePG

- Each PostgreSQL pod exposes metrics via HTTP/HTTPS on port **9187**
- Operator exposes metrics on port **8080**
- Metrics are collected using the ***pg_monitor*** role
- Default cache TTL: 30 seconds (configurable)
- CloudNativePG's metrics system is inspired by postgres_exporter but not dependent on it.

PostgreSQL Monitoring

1. create a cluster with monitoring enabled
2. Apply the configMap and secret with queries
3. Apply the cluster manifest

```
$ cat <<EOF > ./monitored-cluster.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: monitored-cluster
spec:
  instances: 1
  storage:
    size: 1Gi
  monitoring:
    disableDefaultQueries: false
    metricsQueriesTTL: 30s
    customQueriesConfigMap:
      - name: custom-monitoring
        key: queries
    customQueriesSecret:
      - name: custom-monitoring-secret
        key: sensitive-queries
EOF

$ kubectl apply -f monitored-cluster.yaml
$ kubectl port-forward monitored-cluster-1 9187:9187
$ curl localhost:9187/metrics
```

How to manage PostgreSQL **settings**?

PostgreSQL Setting

1. create a 3 instance cluster
2. Apply the manifest

₹

```
$ cat <<EOF > ./setting-example.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: setting-example
spec:
  instances: 3
  storage:
    size: 1Gi

  postgresql:
     parameters:
        shared_buffers: 128MB
EOF

$ kubectl apply -f setting-example.yaml
```

https://cloudnative-pg.io/docs/1.28/postgresql_conf#the-postgresql-section

PostgreSQL Setting

1. Check the current value of shared_buffers in the cluster yaml
2. Check the current value in postgres
3. Add the desired value using cluster Manifest
4. watch the cluster pods/status parallely
5. verify the desired value in the cluster yaml
6. Check the desired value in postgres

```
$ kubectl get cluster setting-example -o yaml |grep
-A25 parameters:

$ kubectl exec -it setting-example-1 -c postgres --
psql -c "show shared_buffers;"

$ sed -i 's/128MB/256MB/' setting-example.yaml

$ cat setting-example.yaml

$ kubectl apply -f setting-example.yaml

$ kubectl get cluster setting-example -o yaml |grep
-A25 parameters:

$ kubectl exec -it setting-example-1 -c postgres --
psql -c "show shared_buffers;"
```

How to **import** a PostgreSQL **databases**?

What is database import and where it is useful ?

- A built-in mechanism to import an existing PostgreSQL database into a new CNPG cluster, using **pg_dump** over the network, no manual scripts needed.

Key scenarios:

- Migrating off a legacy VM/bare-metal PostgreSQL → Kubernetes

  e.g. old Postgres lives on an EC2 instance or on-prem server.

  Configure CNPG to pull the data from legacy database

  No manual dump/restore scripts.
- Major version upgrade (e.g. PG 15 → PG 18)

  CNPG bootstraps a new cluster at the target version,

  dumps from old, restores into new - handles the version gap automatically.

  Old cluster stays live during the process.
- No downtime on the source cluster

Import database

1. Create a table and insert data in source cluster in app db.
2. Enable super user in source cluster

```
$ kubectl cnpg psql setting-example -- app

app=# CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  username VARCHAR(50) NOT NULL,
  email VARCHAR(100) NOT NULL,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

INSERT INTO users (username, email) VALUES
  ('john_doe', 'john@example.com'),
  ('jane_smith', 'jane@example.com'),
  ('bob_jones', 'bob@example.com');

-- Verify the data
SELECT * FROM users;

$ kubectl patch cluster setting-example --type merge -p
'{"spec":{"enableSuperuserAccess":true}}'
```

https://cloudnative-pg.io/docs/1.29/database_import#the-microservice-type

Import database

1. Configure the yaml for import using app db

```
$ cat <<EOF > ./import.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-import-microservice
spec:
  instances: 1
  bootstrap:
    initdb:
      import:
        type: microservice
        databases:
          - app
        source:
          externalCluster: setting-example
  storage:
    size: 1Gi
  externalClusters:
    - name: setting-example
      connectionParameters:
        host: setting-example-rw.default.svc.cluster.local
        user: postgres
        dbname: postgres
      password:
        name: setting-example-superuser
        key: password
EOF
```

Import database

1. Apply the cluster manifest
2. Verify data.

```
$ kubectl apply -f import.yaml

$ kubectl get pods -w

$ kubectl get cluster -w

$ kubectl cnpg psql cluster-db-import -- \
  app -c "SELECT * FROM users"
```

What happens during an **incident**?

Incident Simulation

Different approaches:

- **Pod** deletion:
  - Delete of the PostgreSQL's primary Pod
- **PVC** deletion:
  - Delete the PVC associated to PostgreSQL's primary Pod
- **Node** deletion (**not covered** in this course):
  - Turn off or detach the Kubernetes node where the PostgreSQL's primary Pod is running

https://cloudnative-pg.io/docs/devel/failure_modes/

Failover

1. Check the cluster status and primary
2. Delete primary pod
3. Watch the cluster status

```
$ kubectl get cluster setting-example

$ kubectl get cluster setting-example -w

#Replace Primary pod no.
$ kubectl delete po setting-example-N

$ kubectl cnpg status setting-example
```

Failover

1. Check the cluster status and primary
2. Delete primary pod and pvc
3. Watch the cluster pod
4. Watch the cluster status

```
$ kubectl get cluster setting-example

$ kubectl cluster status -w

$ kubectl get pod -w|grep "setting-example"

$ kubectl delete pvc,pod setting-example-N
$ # where N is the ID of the primary pod

$ kubectl cnpg status setting-example
```

How to read the **logs**?

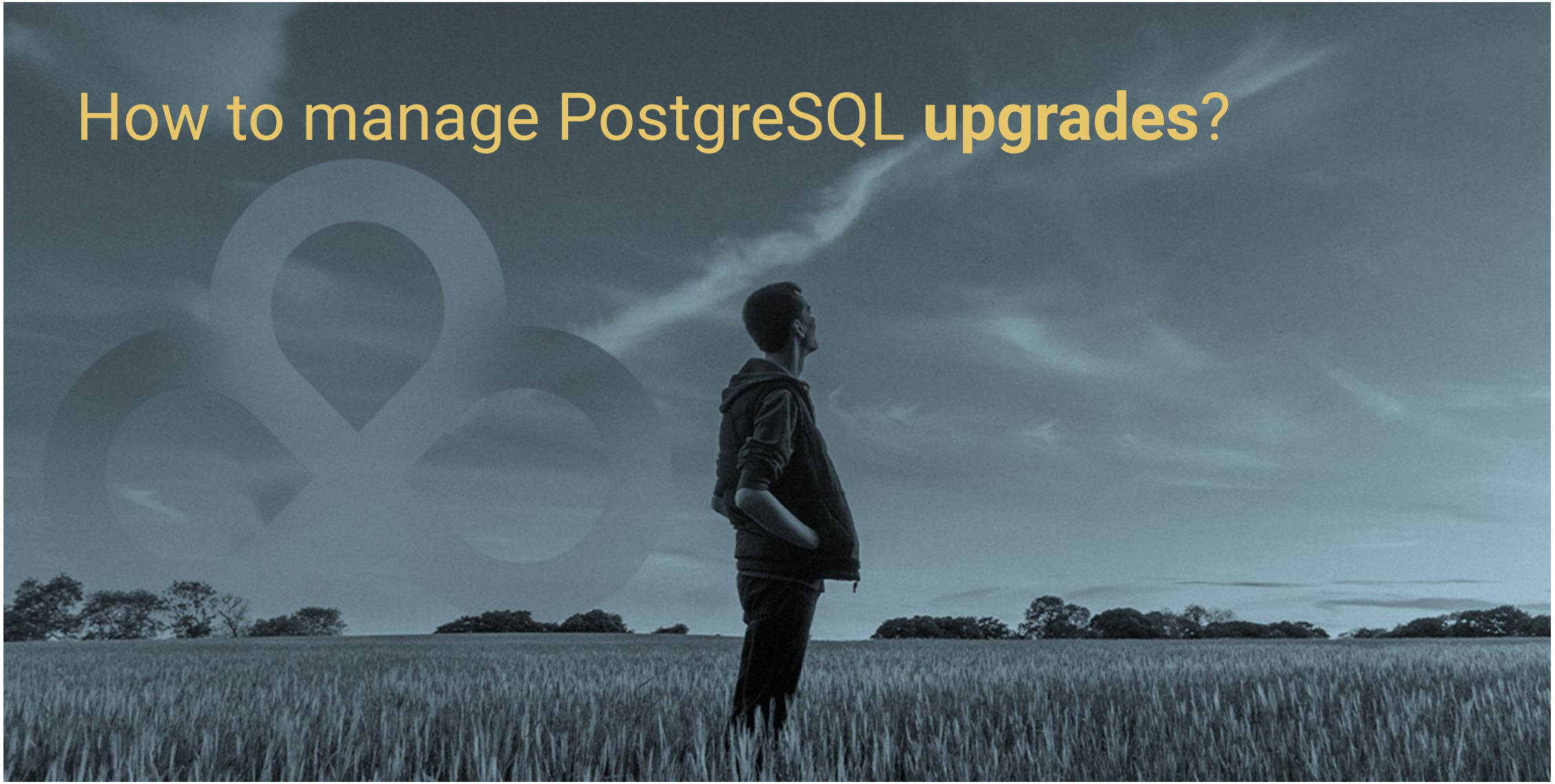
Logs

1. Cluster logs
2. Operator logs
3. Postgres logs

```
$ kubectl get cluster setting-example -o yaml | grep
logLevel

$ kubectl cnpg report operator -n cnpg-system

$ kubectl logs deploy/cnpg-controller-manager -n \
 cnpg-system

$ kubectl logs setting-example-1

$ kubectl cnpg logs cluster setting-example |\
  kubectl cnpg logs pretty
```

https://cloudnative-pg.io/docs/1.29/kubectl-plugin#logs

How to manage PostgreSQL **upgrades**?



EDB POSTGRES AI

PostgreSQL Rolling Updates

Tow **contexts**:

1. **Minor Version** upgrades:
    - Managed as a **Rolling Updates** like the Operator's upgrade
    - Simple PostgreSQL's container image replacement
2. **MAJOR Version** upgrades:
    - More complex operation
    - Different methods

Major Upgrade's **methods**:

1. **Offline**
    - Cluster unavailable during upgrade
2. **Online**
    - Cluster stays available during upgrade
3. **In-place upgrade:**
    - Same cluster, same pods, same PVCs upgraded where they exist

https://cloudnative-pg.io/docs/devel/postgres_upgrades/

Postgres upgrade methods

Method	Risk	Rollback	Types	Best for	Tools used
Rolling update (minor)	Low	Easy	Online (In-place)	Patches, minor version of postgres	Binary swap
pg_upgrade (major)	High	Hard	Offline (In-place)	dev & testing env	pg_upgrade with –link
Blue-green (major)	Low	Easy	Online	Production env	logical replication
Backup & Restore (major)	Medium	Medium	Offline	New infrastructure	pg_basebackup + pg_upgrade
Online import	Low	Easy	Online	Migrate to CNPG from baremetal postgres	pg_basebackup + streaming

Minor Version Upgrade

1. Having a 3 PostgreSQL instance cluster at version 17
2. Monitor resources from a separate terminal
3. Apply the manifest and wait for the cluster to be ready

```
$ cat <<EOF > ./cluster-upgrade.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-upgrade
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:17.0
  instances: 3
  storage:
    size: 1Gi
EOF

$ kubectl get pods -w

$ kubectl apply -f ./cluster-upgrade.yaml
```

Minor Version Upgrade

1. Change the PostgreSQL image TAG from 17.0 to **17.2** in the YAML manifest
2. Apply the `cluster-upgrade.yaml` manifest and verify the cluster status with the CNPG plugin
3. Repeat the plugin's `status` command a few times

```
$ sed -i 's/17\.0/17\.2/' cluster-upgrade.yaml

$ # Verify the changes
$ cat ./cluster-upgrade.yaml

$ kubectl apply -f ./cluster-upgrade.yaml \
  && kubectl cnpg status cluster-upgrade

$ kubectl cnpg status cluster-upgrade
```

Upgrade: Declarative way using CloudNativePG

- **What user have to do?**
  - User can specify new image with higher postgresql 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.

MAJOR Version Upgrade

1. Using the cluster created in the previous exercise at version 17.2
2. Monitor resources from a separate terminal
3. Change the PostgreSQL image TAG from 17.2 to **18.0** in the YAML manifest
4. Apply the `cluster-upgrade.yaml` manifest and verify the cluster status with the CNPG plugin
5. Repeat the plugin's `status` command a few times

```
$ kubectl get pods -w

$ sed -i 's/17\.2/18\.0/' cluster-upgrade.yaml

$ # Verify the changes
$ cat ./cluster-upgrade.yaml

$ kubectl apply -f ./cluster-upgrade.yaml \
  && kubectl cnpg status cluster-upgrade

$ kubectl cnpg status cluster-upgrade
```

Blue-green deployment example

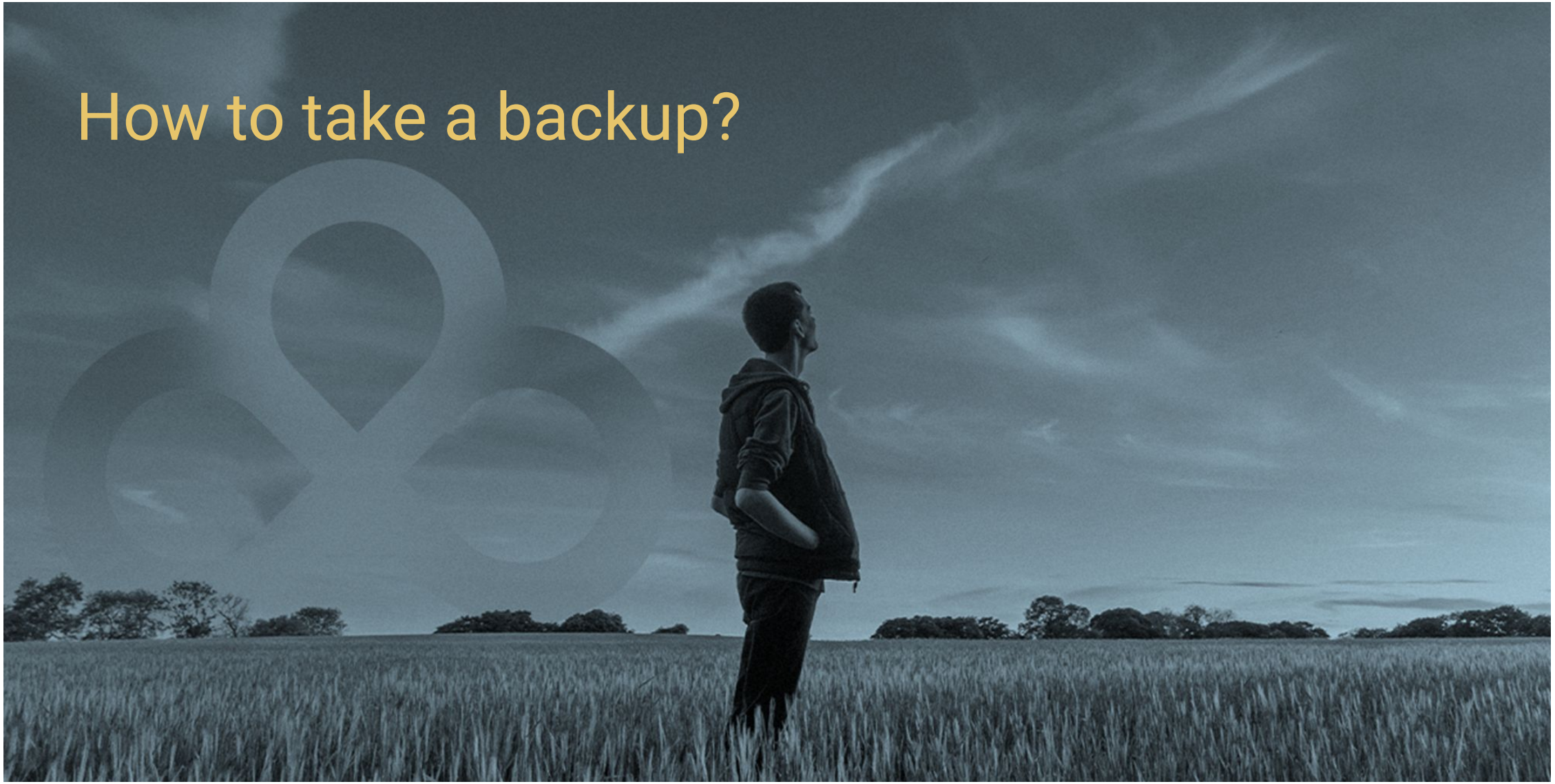
Blue.yaml

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: blue-cluster
spec:
  imageName:
ghcr.io/cloudnative-pg/postgresql:16

  instances: 3
  storage:
    size: 1Gi

  postgresql:
    parameters:
      wal_level: logical
```

Green.yaml

```
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: green-cluster
spec:
  imageName:
ghcr.io/cloudnative-pg/postgresql:17
  instances: 3
  replica:
    enabled: true
    source: blue-cluster

  externalClusters:
    - name: blue-cluster
      connectionParameters:
        host: blue-cluster-rw
        user: postgres
        dbname: app
      password:
        name: blue-cluster-superuser
        key: password
```

How to take a backup?



CNPG Cluster Backup

Methods:

- `barmanObjectStore` (**legacy**)
  - Uses the **in-core Barman Cloud** included in PostgreSQL images to take backups
  - Deprecated
- `volumeSnapshots`
  - Uses Kubernetes native API to take snapshots of the volumes, **when supported** by CSI
  - Fastest to take
  - Still requires WAL archiving to achieve PITR
- `plugins`
  - Uses external **CNPG-I plugins** (es: Plugin Barman Cloud)

https://cloudnative-pg.io/docs/devel/backup/

CNPG-I Plugin Barman Cloud

Features:

- Hot (**online**) backups
  - Compression, Encryption, Parallelism
- Continuous WAL archiving
  - Object Stores
  - Compression, Encryption, Parallelism
- Retention Policies
- **Data Restore**
  - Full Recovery
  - Point In Time Recovery

How it works:

- From CNPG **v1.26.0**
- Integration with CNPG Cluster
  - Container **Sidecar**
- **New** CRD
  - `ObjectStore`
- Backups and WAL files
  - tags
  - extra barman options

https://cloudnative-pg.io/plugin-barman-cloud/docs/intro/

Deploy Barman Cloud

1. Deploy cert manager
2. Deploy minio
3. Connect minio to kind(K8s) network
4. Deploy the Plugin Barman Cloud
5. Wait for the deployment to be ready
6. From the cnpg-playground repo, deploy the ObjectStore manifest for EU minio
7. Check for the ObjectStore resource

```
$kubectl apply -f
https://github.com/cert-manager/cert-manager/releases/download/v1
.19.4/cert-manager.yaml

$docker run --name minio -e "MINIO_ROOT_USER=cnpg" -e
"MINIO_ROOT_PASSWORD=Cl0udNativePGRocks" \
   -p 9001:9001 \
         -d \
   quay.io/minio/minio:latest server /data --console-address
":9001"

$docker network connect kind minio

$ kubectl apply -f
https://github.com/cloudnative-pg/plugin-barman-cloud/releases/do
wnload/v0.10.0/manifest.yaml

$ kubectl rollout status deployment \
  -n cnpg-system barman-cloud

$ kubectl apply -f \
  demo/yaml/object-stores/minio-eu.yaml

$ kubectl get objectstore
```

Deploy CNPG Cluster

1. Create a cluster YAML file with the backup configuration
2. Monitor resources in a separate terminal
3. Apply the manifest
4. Check for the cluster
    - Use the CNPG plugin `status` command
    - Use the `kubectl get` command to check for the sidecar container

```
$ cat <<EOF > ./cluster-with-backup.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-with-backup
spec:
  instances: 2
  storage:
    size: 1Gi

  plugins:
  - name: barman-cloud.cloudnative-pg.io
    isWALArchiver: true
    parameters:
      barmanObjectName: minio
      serverName: cluster-wit-backup
EOF

$ kubectl apply -f cluster-with-backup.yaml

$ kubectl cnpg status cluster-with-backup

$ kubectl get pod cluster-with-backup
```

First CNPG Backup

1. Generate data inside the database
    - Access to the `app` DB with the plugin
    - Create the `numbers` table
    - Insert **1.000.000 rows** into the table
2. Create the a backup manifest

https://cloudnative-pg.io/docs/devel/backup/#example-requesting-an-on-demand-backup

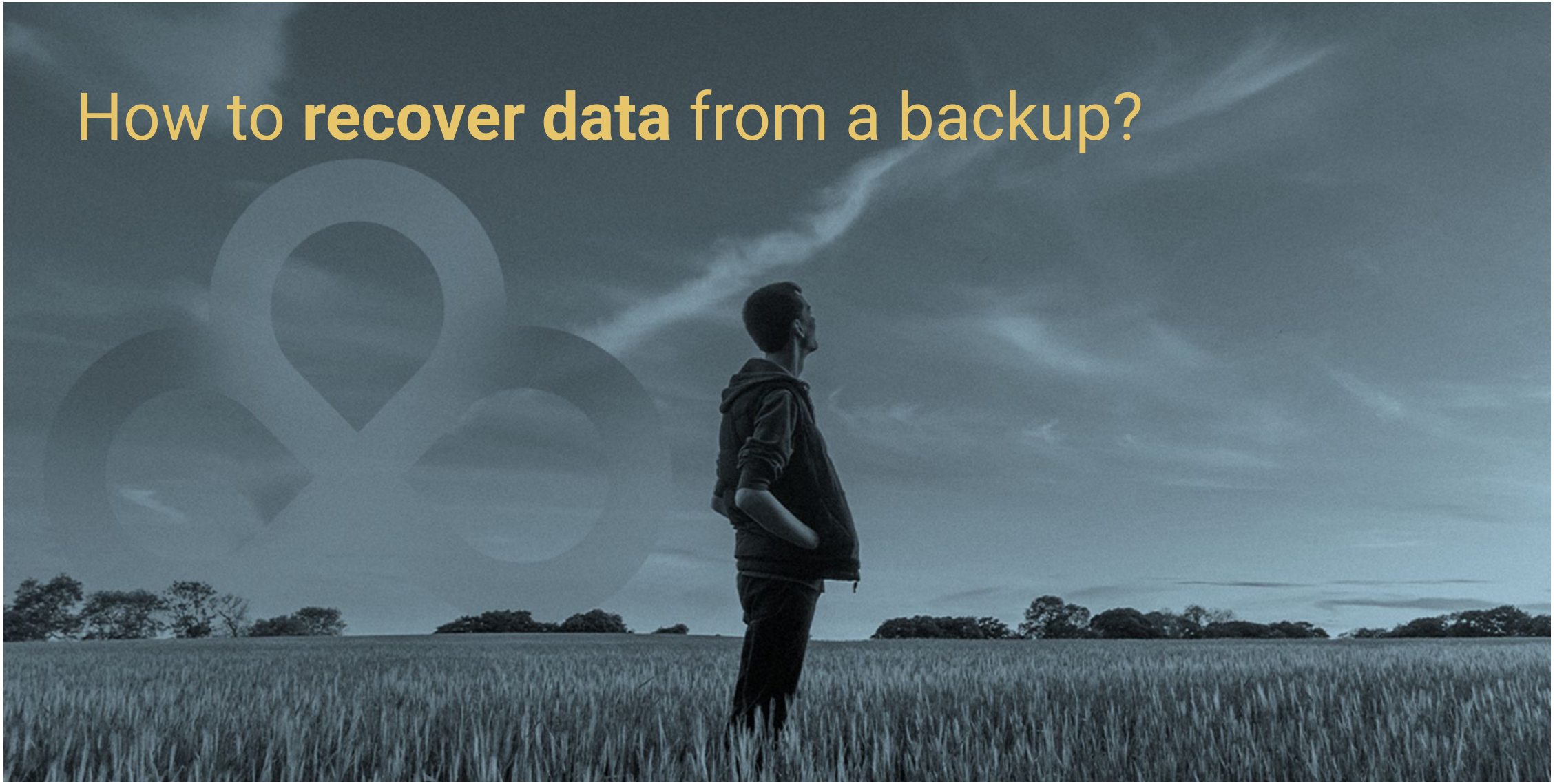
```
$ kubectl cnpg psql cluster-with-backup -- app

app=# CREATE TABLE numbers(x int);

app=# INSERT INTO numbers (SELECT
generate_series(1,1000000));

app=# \q

$ cat <<EOF > backup.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Backup
metadata:
  name: backup-example
spec:
  method: plugin
  cluster:
    name: cluster-with-backup
  pluginConfiguration:
    name: barman-cloud.cloudnative-pg.io
EOF
```

First CNPG Backup

1. Create the first two backups
   - Using the backup manifest
   - Using the plugin's `backup` command
2. Analyse the backup resources
3. Analyse the cluster status
   - Check how the `First Point of Recoverability` field has changed

```
$ kubectl apply -f backup.yaml

$ kubectl cnpg backup cluster-with-backup \
  --immediate-checkpoint true \
  --backup-target primary \
  --method plugin \
  --plugin-name barman-cloud.cloudnative-pg.io

$ kubectl get backup

$ kubectl get backup -o yaml

$ kubectl cnpg status cluster-with-backup
```

How to **recover data** from a backup?



CNPG Cluster Recovery

Methods:

- `barmanObjectStore`
- `volumeSnapshots`
- `plugin`

CloudNativePG recovery **must know**:

- Cannot recover a Cluster in-place
- It's a Bootstrap method for a different Cluster
- WAL archiving must be redirected to a different object store path (hence the new Cluster name)
  - to prevent WAL files conflicts (overwriting original ones)

https://cloudnative-pg.io/docs/devel/recovery/

CNPG Cluster Recovery

1. Create a manifest with:
    a. the `recovery` method for the `bootstrap` section that points to the right `externalClusters`
    b. the `externalClusters` section pointing to the right `barmanObjectName` and `serverName`.

```
$ cat <<EOF > ./cluster-recovery.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-recovery
spec:
  instances: 1
  storage:
    size: 1Gi

  bootstrap:
    recovery:
      source: origin

  externalClusters:
    - name: origin
      plugin:
        name: barman-cloud.cloudnative-pg.io
        parameters:
          barmanObjectName: minio-eu
          serverName: cluster-with-backup
EOF
```

CNPG Recovery

1. Monitor resources in a separate terminal
2. Apply the `cluster-recovery` manifest
3. Check for the cluster `status`
4. Verify the data in the `app` DB

```
$ kubectl get pods -w

$ kubectl apply -f ./cluster-recovery.yaml

$ kubectl cnpg status cluster-recovery

$ kubectl cnpg psql cluster-recovery -- app \
  -c “SELECT COUNT(*) numbers”
```

Questions?

Thank you!

Let’s keep in touch!

Website: cloudnative-pg.io

Blog: cloudnative-pg.io/blog/

GitHub Discussions: github.com/cloudnative-pg/cloudnative-pg/discussions

Slack: communityinviter.com/apps/cloud-native/cncf

LinkedIn: linkedin.com/company/cloudnative-pg/

Mastodon: @CloudNativePG@mastodon.social

Bluesky: @CloudNativePG.bsky.social