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

Neel Patel

Principal Software Engineer at EDB

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
- 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:

- **LINKS.md**
  - contains the list of links mentioned during the course

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

Setup environment

cnpg-playground Requirements:

- Docker
- Kind
- kubectl
- CNPG Plugin
- git

Then:

- Set limits with `sysctl` (linux)
- Exec `setup.sh` script
- Export of the Kube config
- Set EU context
- Creare gli alias
- Test the connection

https://github.com/cloudnative-pg/cnpg-playground/

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

$ cd cnpg-playground

$ # Edit the ./k8s/kind-cluster.yaml
$ # Add the following configuration
kind: Cluster
apiVersion: kind.x-k8s.io/v1alpha4
name: cnpg
featureGates:
  ImageVolume: true

$ ./scripts/setup.sh eu

$ export
KUBECONFIG=<path-to>/cnpg-playground/k8s/kube-config.yaml

$ kubectl config use-context kind-k8s-eu

$ . <path-to>/bash_aliases.sh

$ kubectl get pods -A
$ keu get pods -A
```

CloudNativePG

- A CNCF **Sandbox** Operator
- **Manages** the whole lifecycle of a PostgreSQL cluster
- **Declarative** Configuration
- Runs on **many** Kubernetes distribution:
  - Vanilla
  - OpenShift
  - Cloud Service Providers K8S env
  - kind
  - …

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

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.

CloudNativePG - Main Features

https://cloudnative-pg.io/docs/devel/#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, including temporary
- **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, …)

CloudNativePG - Latest Release: 1.28

https://cloudnative-pg.io/docs/1.28/release_notes/v1.28

- **Features**
  -
- **Changes**
  -

How to install it?

Using the manifest

1. Install the CNPG **Operator v1.27.2**
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-1.27/releases/cnpg-1.27.2.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 1.27.2 \
  | kubectl apply -f - --server-side
```

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

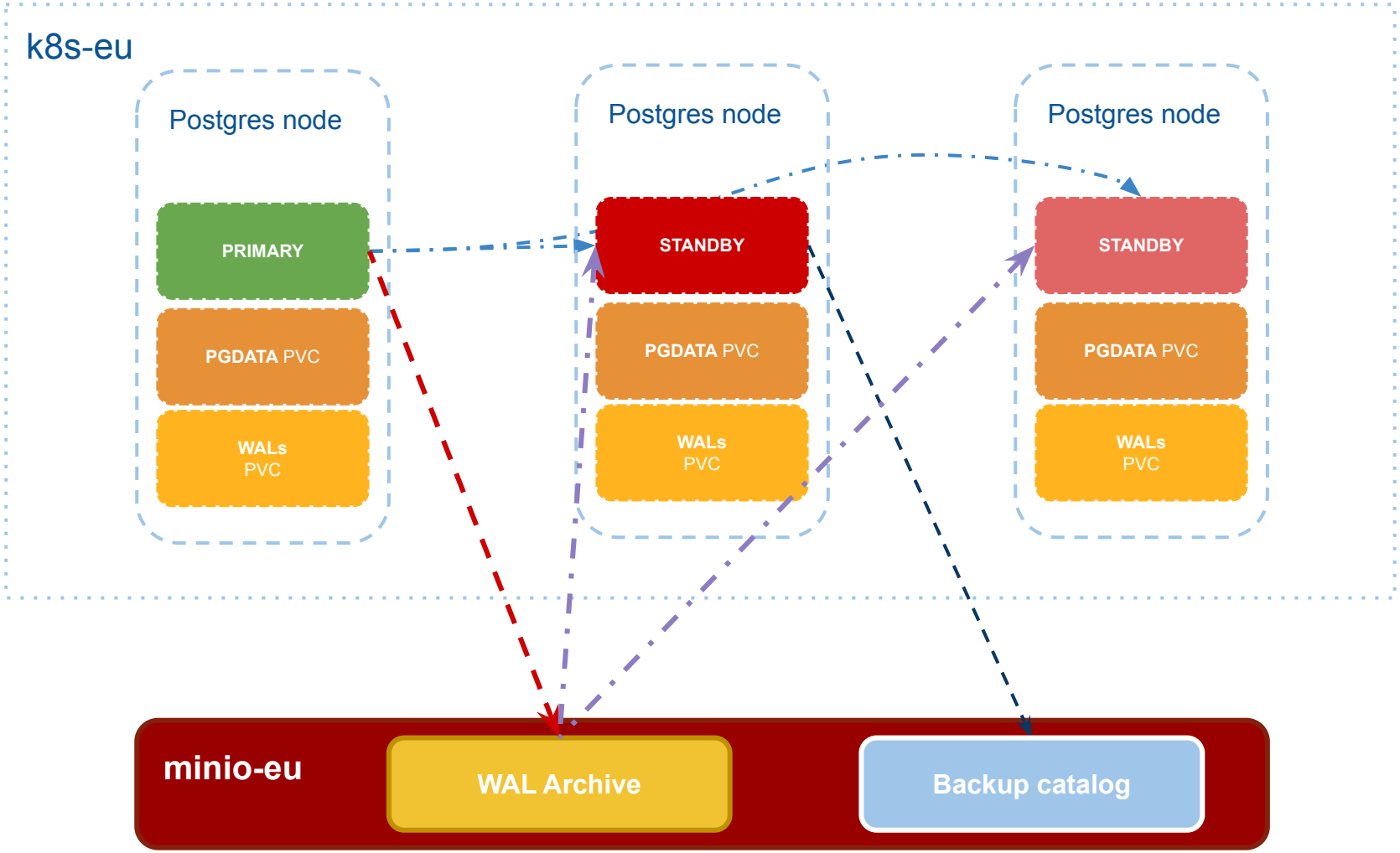
Deploy the first Cluster

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**?

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
    - plain text
    - md5 or scram (not usable by apps)
  - 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:
      - dante
      - petrarca
      reconciled:
      - ariosto
      reserved:
      - postgres
      - streaming_replica
    cannotReconcile:
      dante:
      - 'could not perform DELETE on role dante: owner
of database inferno'
      petrarca:
      - 'could not perform UPDATE_MEMBERSHIPS on role
petrarca: role "poets" does not exist'
```

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: dante
      ensure: present
      comment: Dante Alighieri
      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 cnpg psql cluster-example -- app \
  -c “ALTER ROLE dante NOLOGIN”

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

$ kubectl cnpg reload cluster-example

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

How to manage PostgreSQL **databases**?

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**?

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: Database
metadata:
  name: cluster-ext-app
spec:
  cluster:
    name: cluster-ext
  name: app
  owner: app
  extensions:
  - name: vector
    version: '0.8.1'
```

Declarative Extensions

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

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

$ kubectl apply -f cluster-ext.yaml

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

Declarative Extensions

1. Create the `cluster-ext_db-one.yaml` manifest
2. Apply the `cluster-ext_db-one.yaml` manifest
3. Check for the changes in Postgres

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

$ kubectl apply -f cluster-ext_db-app.yaml

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

Image Volume Extensions

- **Requires** at least:
  - CNPG v1.27
  - PostgreSQL v18
  - Kubernetes v1.33
- Allows dynamically adding extensions as dedicated **OCI images**
- **Avoids** heavy PostgreSQL image
- Decuple upgrades
  - PostgreSQL
  - Extensions
- **Easier** to maintain

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

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: cluster-example-app
spec:
  cluster:
    name: cluster-example-ext
  name: app
  owner: app
  extensions:
  - name: vector
    version: '0.8.1'
---
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-ext
spec:
  imageName: ghcr.io/cloudnative-pg/postgresql:18-minimal-trixie
  instances: 1

  storage:
    size: 1Gi

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

PostGIS - Single Image

1. Select image:
    - Different PostgreSQL image with PostGIS included
2. Create a Cluster resource in a YAML file
3. Create a Database resource in a YAML file

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

```
$ cat <<EOF > cluster-postgis.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: postgis-example
spec:
  instances: 1
  imageName: ghcr.io/cloudnative-pg/postgis:18-3.6-system-trixie
  storage:
    size: 1Gi
  postgresql:
    parameters:
      log_statement: ddl
EOF

$ cat <<EOF > cluster-postgis_db-app.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: postgis-example-app
spec:
  name: app
  owner: app
  cluster:
    name: postgis-example
  extensions:
  - name: postgis
  - name: postgis_topology
  - name: fuzzystrmatch
  - name: postgis_tiger_geocoder
EOF
```

PostGIS - Single Image

1. Apply the `cluster-postgis.yaml` manifest for the cluster
2. Apply the `cluster-postgis_db-app.yaml` manifest for the database
3. Check for the changes in Postgres

```
$ kubectl apply -f cluster-postgis.yaml

$ kubectl apply -f cluster-postgis_db-app.yaml

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

PostGIS - Image Volume Extension

1. Select images:
   - PostgreSQL minimal image
   - PostGIS as Image Volume Extension
2. Create a Cluster resource in a YAML file
3. Apply the `cluster-postgis-ive.yaml` manifest

https://github.com/cloudnative-pg/postgres-extensions-containers/

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

  storage:
    size: 1Gi

  postgresql:
    extensions:
    - name: postgis
      image:
        reference:
ghcr.io/cloudnative-pg/postgis-extension:3.6.1-18-trixie
      ld_library_path:
      - system
EOF

$ kubectl apply -f cluster-ive.yaml
```

PostGIS - Image Volume Extension

1. Create a Database resource in a YAML file
2. Apply the `postgis-ive_db-app` manifest for the database
3. Check for the changes in Postgres

```
$ cat <<EOF > cluster-ive_db-app.yaml
apiVersion: postgresql.cnpg.io/v1
kind: Database
metadata:
  name: postgis-ive-app
spec:
  name: app
  owner: app
  cluster:
    name: cluster-postgis-ive
  extensions:
  - name: postgis
    version: '3.6.1'
  - name: postgis_raster
  - name: postgis_sfcgal
  - name: fuzzystrmatch
  - name: address_standardizer
  - name: address_standardizer_data_us
  - name: postgis_tiger_geocoder
  - name: postgis_topology
EOF

$ kubectl apply -f postgis-ive_db-app.yaml
```

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**?

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/1.28/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-import-microservice -- \
  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/1.28/kubectl-plugin#logs

How to manage CNPG **upgrades**?

CNPG Operator Rolling Updates (default)

Two **phases**:

1. **Operator's** upgrade
   - Operator's container image
   - Custom Resource Definition upgrade
2. **Instance Manager** upgrade
   - Within the PostgreSQL instance Pod

Three **methods**:

1. **Automatic** (default)
   - Automatic **switchover or restart** of the PostgreSQL's primary Pod
2. **Semi-automatic**
   - Automatic restart of the PostgreSQL's replica Pod(s)
   - **Manual** switchover or restart of the PostgreSQL's primary Pod
3. **In-place binary replacement**
   - Breaks **immutability** concept
   - Controlled by Operator's Configuration

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

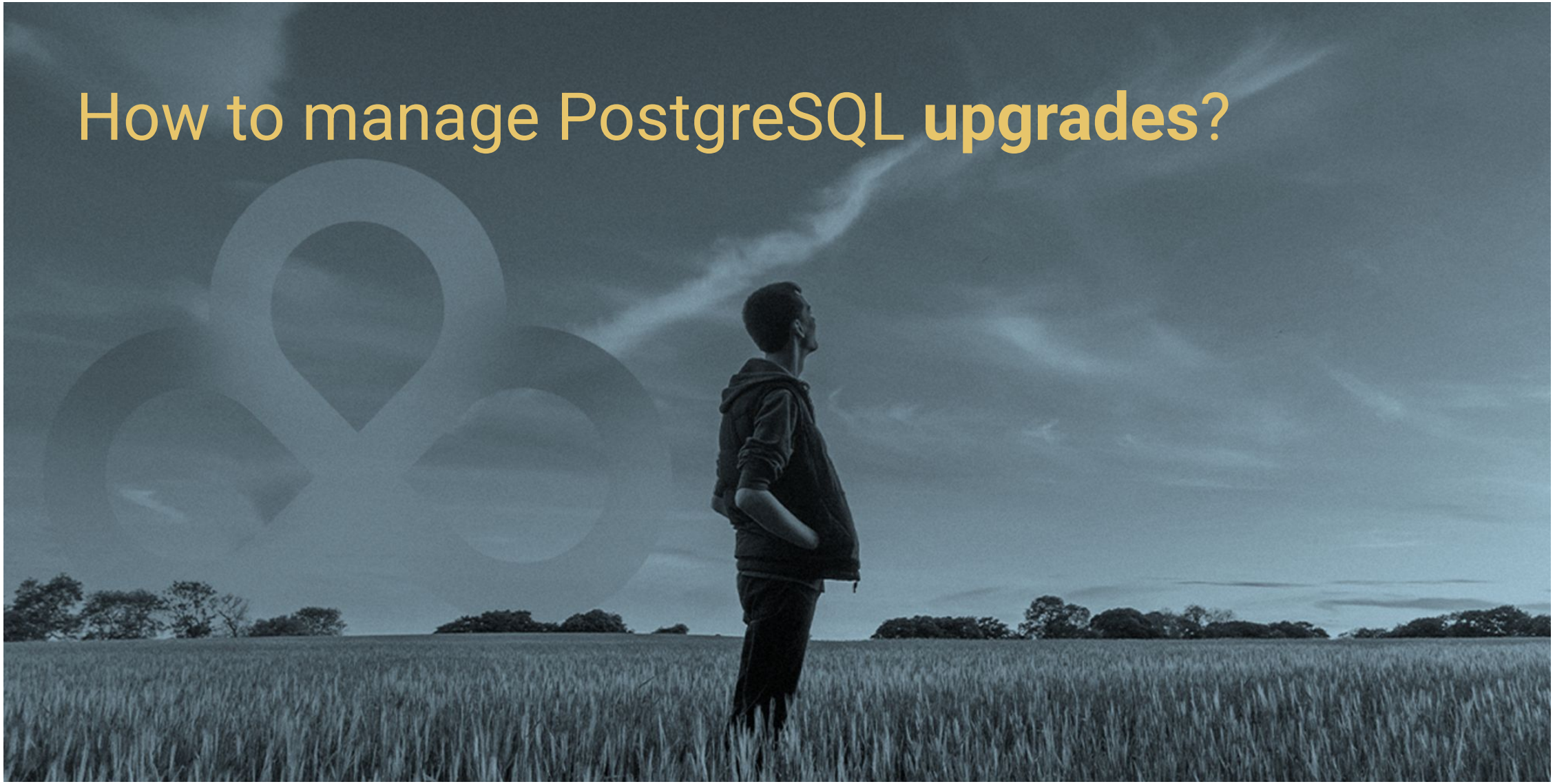
CNPG Operator's Upgrade

Current version running: CNPG **v1.27.2**

1. Monitor resources from two different terminals
    - One for `cnpg-system` namespace
    - One for the cluster namespace (`default`)
2. Install the new CNPG version: **v1.28.0**

```
$ kubectl get pods -n cnpg-system -w

$ kubectl get pods -w

$ kubectl apply --server-side -f \

https://raw.githubusercontent.com/cloudnative-pg/cloudn
ative-pg/release-1.28/releases/cnpg-1.28.0.yaml
```

How to manage PostgreSQL **upgrades**?



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

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

Major Upgrade's **methods**:

1. **Offline**
    - **Logical dump/restore** from one instance to another - **duplicate resources' consumption**
    - **In-place upgrade** - data directory replacement (**pg_upgrade --link**)
2. **Online**
    - **PostgreSQL's** native **Logical Replication** from one instance to another - **duplicate resources' consumption**

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

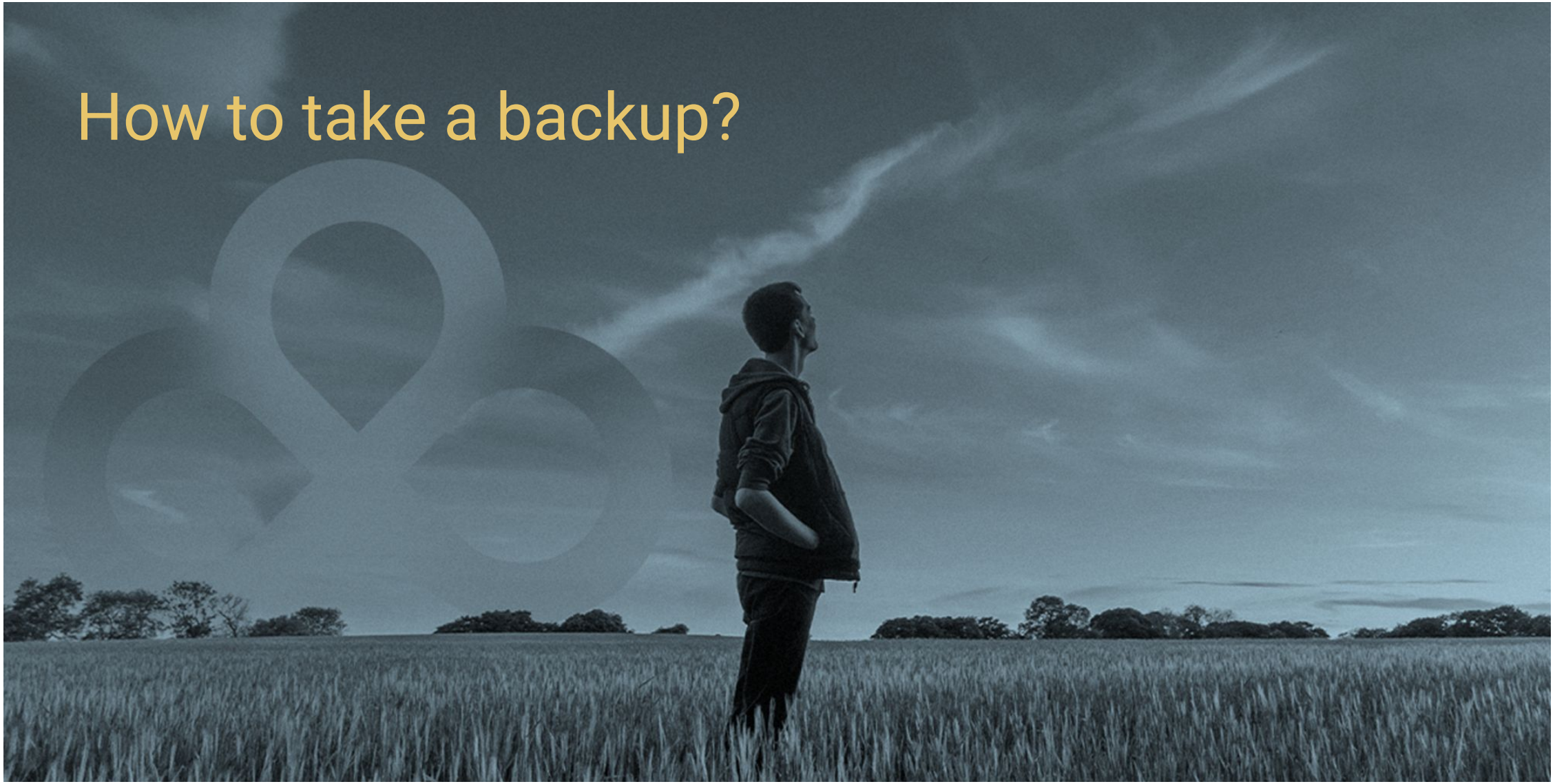
```
$ sed -i 's/17\.0/17\.2/' cluster-upgrade.yaml

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

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

$ kubectl cnpg status cluster-upgrade
```

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
```

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 the Plugin Barman Cloud
2. Wait for the deployment to be ready
3. From the cnpg-playground repo, deploy the ObjectStore manifest for EU minio
4. Check for the ObjectStore resource

```
$ kubectl apply -f
https://github.com/cloudnative-pg/plugin-barman-cloud/r
eleases/download/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-eu
      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

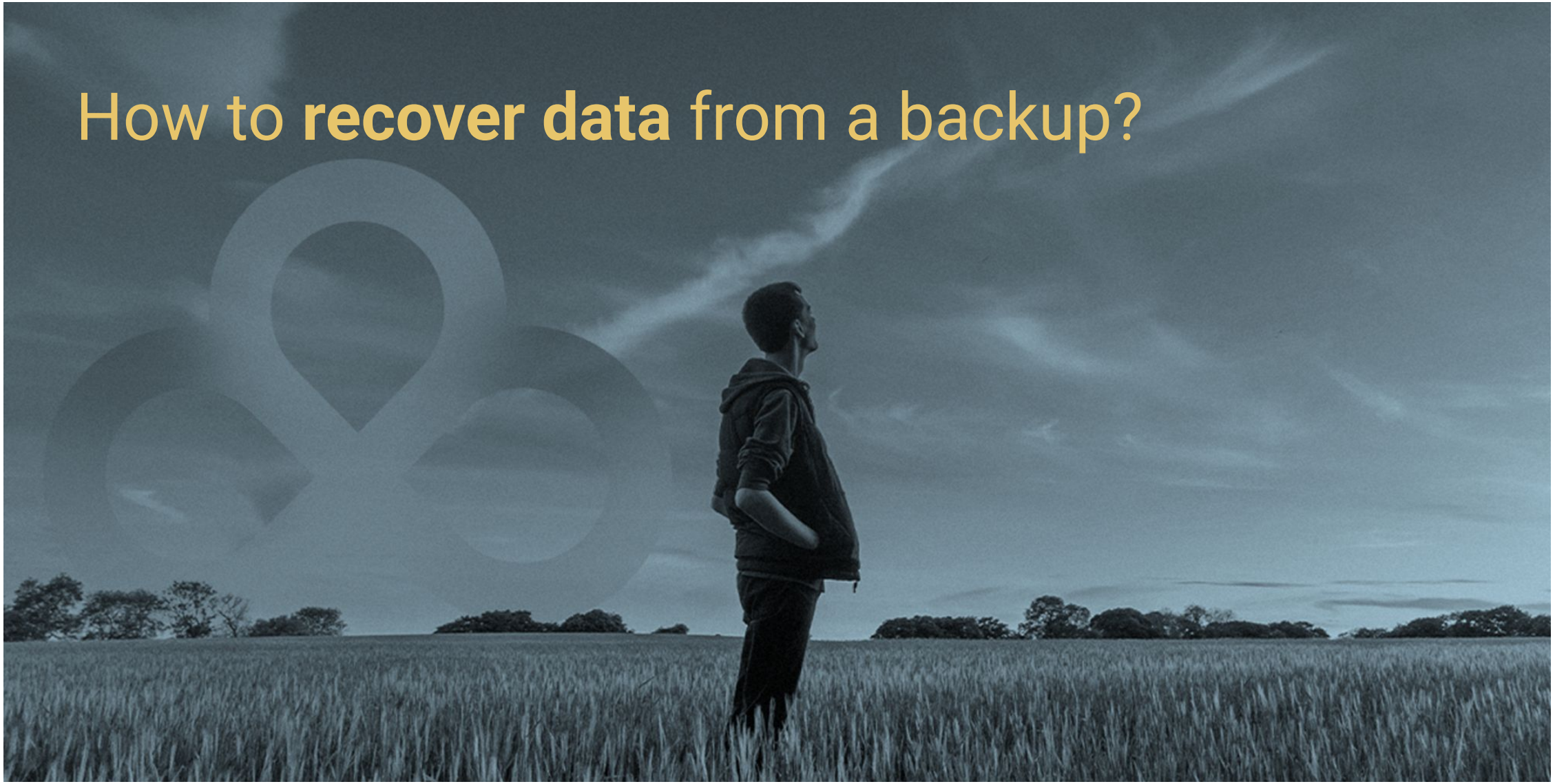
```
$ kubectl cnpg psql cluster-with-backup -- app

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