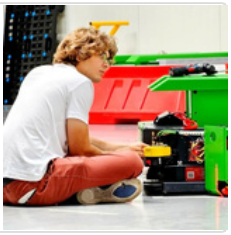
Kubernetes Databases: HA PostgreSQL with CNPG, schema management via Atlas, backups, monitor DB with Prometheus & Grafana.

Check GitHub for helpful DevOps tools:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



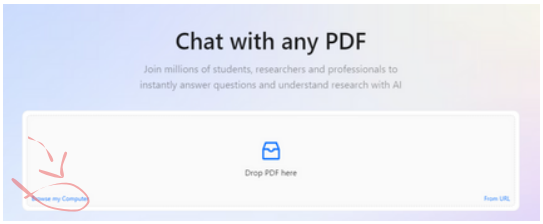
Ask Personal AI Document assistant to learn interactively (FASTER)!

1. Download PDF
2. Go to website
3. Browse file
4. Chat with Document

1 https://github.com/MichaelRobotics/DevOpsTools/blob/main/KubernetesDatabases.pdf

Click there to go to ChatPdf website 2

3



Ask questions about document! 4

Complety new to Linux and Networking?

Essential for this PDF is a thorough knowledge of networking. I highly recommend the HTB platform's networking module, which offers extensive information to help build a comprehensive understanding.

HTB - Your Cyber Performance Center

We provide a human-first platform creating and maintaining high performing cybersecurity individuals and organizations.

https://www.hackthebox.com/

What is Kubernetes?

Kubernetes is an open-source platform that automates the deployment, scaling, and management of containerized applications. It helps manage clusters of nodes running containers, ensuring efficient and reliable operation.

How Kubernetes clusters are made?

Kubernetes clusters consist of a control plane and multiple worker nodes. The control plane manages cluster operations, while worker nodes run the actual container workloads.

Why and When use Kubernetes

Kubernetes is ideal for deploying scalable, resilient, and automated containerized applications. It is used when managing multiple containers across different environments is necessary.

Example: Running a microservices-based e-commerce platform that scales up during peak hours.

System Requirements

- RAM: 2 GB per node (1 GB can work for testing but may lead to limited performance)
- 10 GB free storage
- Ubuntu

Kubernetes: Main components & packages

- **kube-apiserver:** Central management component that exposes the Kubernetes API; acts as the front-end for the cluster.
- **etcd:** Distributed key-value store for storing all cluster data, ensuring data consistency across nodes.
- **kube-scheduler:** Assigns pods to available nodes based on resource requirements and policies.
- **kube-controller-manager:** Manages core controllers that handle various functions like node status, replication, and endpoints.
- **kubelet:** Agent that runs on each node, responsible for managing pods and their containers.
- **kube-proxy:** Manages networking on each node, ensuring communication between pods and services within the cluster.

Kubernetes Packages: HA PostgreSQL with CNPG and Prometheus & Grafana

1) Intro

Running databases in Kubernetes is not recommended – this was true in the past when databases could only be managed with Kubernetes native resources like StatefulSets. To successfully run databases in production, we need proper backups, observability, failovers, and promotions. Simply running a database on a StatefulSet is no longer sufficient! This is where CNPG comes in!

2) Setup

get repo:

```
git clone https://github.com/vfarcic/cloud-native-pg-demo
cd cloud-native-pg-demo
```

Install cnpg and prometheus with helm:

```
helm repo add cnpg https://cloudnative-pg.github.io/charts

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

```
helm repo update

helm upgrade --install cnpg cnpg/cloudnative-pg \
--namespace cnpg-system --create-namespace --wait
```

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

```
kubectl --namespace observability apply \
--filename grafana-configmap.yaml
```

kubectl create namespace demo

3) CNPG demo

Lets take a look at CNPG yaml:

```
cat cluster.yaml
```

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

spec:
  instances: 3
  storage:
    size: 1Gi
```

This configuration is minimal yet functional, Creates a PostgreSQL cluster with 3 instances with 1Gb: one primary for writes and two replicas for high availability and load balancing supported out of the box.

apply

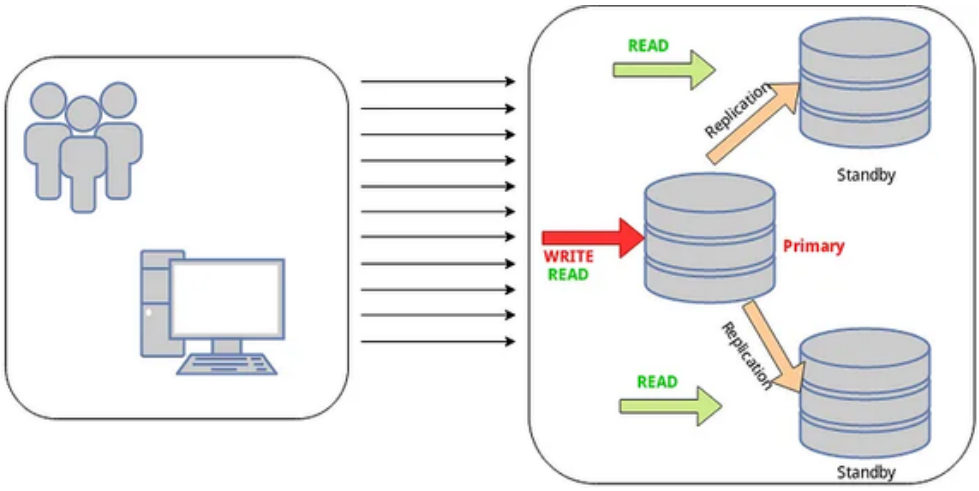
```
kubectl --namespace demo apply --filename cluster.yaml
```

Check Cluster resources:

```
kubectl --namespace demo get clusters
```

```
NAME         AGE   INSTANCES   READY   STATUS              PRIMARY
silly-demo   79s   1                   Setting up primary
controlplane $ kubectl --namespace demo get clusters
NAME         AGE     INSTANCES   READY   STATUS                  PRIMARY
silly-demo   2m31s   2           1       Creating a new replica   silly-demo-1
controlplane $ kubectl --namespace demo get clusters
NAME         AGE     INSTANCES   READY   STATUS                  PRIMARY
silly-demo   2m46s   2           1       Creating a new replica   silly-demo-1
controlplane $ kubectl --namespace demo get clusters
NAME         AGE     INSTANCES   READY   STATUS                    PRIMARY
silly-demo   4m10s   3           3       Cluster in healthy state   silly-demo-1
```

CNPG automatically handles load balancing and high availability. It begins by creating the primary database, then adds second and third replicas. This setup mirrors a standard HA database configuration, where read traffic is distributed across all three, while writes go only to the primary. If the primary database fails, a secondary is promoted.



`kubectl --namespace demo get pods`

```
controlplane $ kubectl --namespace demo get pods
NAME          READY   STATUS    RESTARTS   AGE
silly-demo-1  1/1     Running   0          18m
silly-demo-2  1/1     Running   0          17m
silly-demo-3  1/1     Running   0          16m
```

Each pod corresponds to each database

`kubectl --namespace demo get statefulset`

```
controlplane $ kubectl --namespace demo get statefulsets
No resources found in demo namespace.
```

There are no statefullsets! Databases are managed by CR defined by CNPG.

`kubectl --namespace demo get services`

```
No resources found in demo namespace.
controlplane $ kubectl --namespace demo get services
NAME            TYPE        CLUSTER-IP       EXTERNAL-IP   PORT(S)    AGE
silly-demo-r    ClusterIP   10.102.108.207   <none>        5432/TCP   20m
silly-demo-ro   ClusterIP   10.106.203.164   <none>        5432/TCP   20m
silly-demo-rw   ClusterIP   10.111.214.111   <none>        5432/TCP   20m
```

To access the main database, connect using the service endpoint ending in "rw."

`kubectl --namespace demo get secrets`

```
laptopdev@laptopdev2:~/cloud-native-pg-demo$ kubectl --namespace demo get secrets
NAME                     TYPE                       DATA   AGE
silly-demo-app           kubernetes.io/basic-auth   9      43m
silly-demo-ca            Opaque                     2      43m
silly-demo-replication   kubernetes.io/tls          2      43m
silly-demo-server        kubernetes.io/tls          2      43m
```

CNPG creates all secrets need to safe database setup. To acces our database, decode and use secrets stored in silly-demo-app.

4) CNPG backup and more

CNPG offers much and much more configuration possibilities. Download full example from their website:

```
curl -O https://cloudnative-pg.io/documentation/1.20/samples/cluster-example-full.yaml
```

Lets check different fields

```
cat cluster-example-full.yaml
```

```yaml
apiVersion: v1
data:
  password: VHhWZVE0bk44MlNTaVlIb3N3cU9VUlp2UURhTDRLcE5FbHNDRUVlOWJ3RHhNZDczS2NrSWVYelM1Y1U2TGlDMg==
  username: YXBw
kind: Secret
metadata:
  name: cluster-example-app-user
type: kubernetes.io/basic-auth
---
apiVersion: v1
data:
  password: dU4zaTFIaDBiWWJDYzRUeVZBYWNCaG1TemdxdHpxeG1PVmpBbjBRSUNoc0pyU211OVBZMmZ3MnE4RUtLTHBaOQ==
  username: cG9zdGdyZXM=
kind: Secret
metadata:
  name: cluster-example-superuser
type: kubernetes.io/basic-auth
---
apiVersion: v1
kind: Secret
metadata:
  name: backup-creds
data:
  ACCESS_KEY_ID: a2V5X2lk
  ACCESS_SECRET_KEY: c2VjcmV0X2tleQ==
```

This configuration defines a PostgreSQL cluster using the CloudNativePG operator's CRD, with settings for secrets, storage, backups, and resources. It includes base64-encoded secrets for the application user and superuser credentials, as well as backup credentials for AWS S3 storage.

```yaml
apiVersion: postgresql.cnpg.io/v1
kind: Cluster
metadata:
  name: cluster-example-full
spec:
  description: "Example of cluster"
  imageName: ghcr.io/cloudnative-pg/postgresql:16.1
  # imagePullSecret is only required if the images are located in a private registry
  # imagePullSecrets:
  #   - name: private_registry_access
  instances: 3
  startDelay: 300
  stopDelay: 300
  primaryUpdateStrategy: unsupervised
```

cluster use the ghcr.io/cloudnative-pg/postgresql:16.1 image. Sets up 3 instances for high availability, with a 300-second delay for both start and stop actions, and an unsupervised update strategy for the primary instance. CNPG can pull images from private registry.

```yaml
  postgresql:
    parameters:
      shared_buffers: 256MB
      pg_stat_statements.max: '10000'
      pg_stat_statements.track: all
      auto_explain.log_min_duration: '10s'
    pg_hba:
      - host all all 10.244.0.0/16 md5
```

The configuration allocates 256MB of memory for caching data. It tracks up to 10,000 query plans (pg_stat_statements.max) and monitors all SQL statements with pg_stat_statements.track: all.
Queries taking longer than 10 seconds are logged (auto_explain.log_min_duration: '10s'). The pg_hba: connections from the 10.244.0.0/16 subnet with MD5 password authentication. The first all refers to all databases, and the second all refers to all users, allowing any user to connect to any database.

```
  bootstrap:
    initdb:
      database: app
      owner: app
      secret:
        name: cluster-example-app-user
    # Alternative bootstrap method: start from a backup
    #recovery:
    #  backup:
    #    name: backup-example
```

Initialization process sets up a new database named app. Only App DB user can acces this DB, credentials are stored in secret (cluster-example-app-user). DB can be provided from a backup, where the recovery section would specify the backup to restore from.

```
  superuserSecret:
    name: cluster-example-superuser

  storage:
    storageClass: standard
    size: 1Gi
```

superuserSecret specifies the Kubernetes secret (cluster-example-superuser) with superuser credentials. You can decide what storageClass CNPG uses and how much space to allocate.

```
resources:
  requests:
    memory: "512Mi"
    cpu: "1"
  limits:
    memory: "1Gi"
    cpu: "2"

affinity:
  enablePodAntiAffinity: true
  topologyKey: failure-domain.beta.kubernetes.io/zone

nodeMaintenanceWindow:
  inProgress: false
  reusePVC: false
```

You can set resource requestss and limits for pods. enablePodAntiAffinity: true ensures that pods are spread across different nodestopologyKey: failure-domain.beta.kubernetes.io/zone ensures pod distribution across different Kubernetes availability zones. Label inProgress: false means no maintenance is ongoing, and reusePVC: false ensures existing Persistent Volumes are not reused during maintenance.

```yaml
backup:
  barmanObjectStore:
    destinationPath: s3://cluster-example-full-backup/
    endpointURL: http://custom-endpoint:1234
    s3Credentials:
      accessKeyId:
        name: backup-creds
        key: ACCESS_KEY_ID
      secretAccessKey:
        name: backup-creds
        key: ACCESS_SECRET_KEY
    wal:
      compression: gzip
      encryption: AES256
    data:
      compression: gzip
      encryption: AES256
      immediateCheckpoint: false
      jobs: 2
  retentionPolicy: "30d"
```

Backup is configured with Barman tool. The backups are stored in an S3. destinationPath points to your AWS s3, you can use other storages or Clouds like GCP or Azure. To acces s3, get credentials from secret. The Write-Ahead Log (WAL) and data backups are compressed using gzip to save storage space, and encrypted using AES256. The backup process runs with two parallel jobs to optimize performance, and it does not trigger an immediate checkpoint during the backup.

Additionally, the configuration includes a retention policy, set to keep backups for 30 days ("30d"). After this period, older backups are automatically deleted. The system ensures that backup data is securely handled through encryption and compression.

7) Other functionalities

CNPG completedy destroys statefulsset-way of DB implementation in k8s. It have many more functionalities and part of them are listed there, for more, check their website:

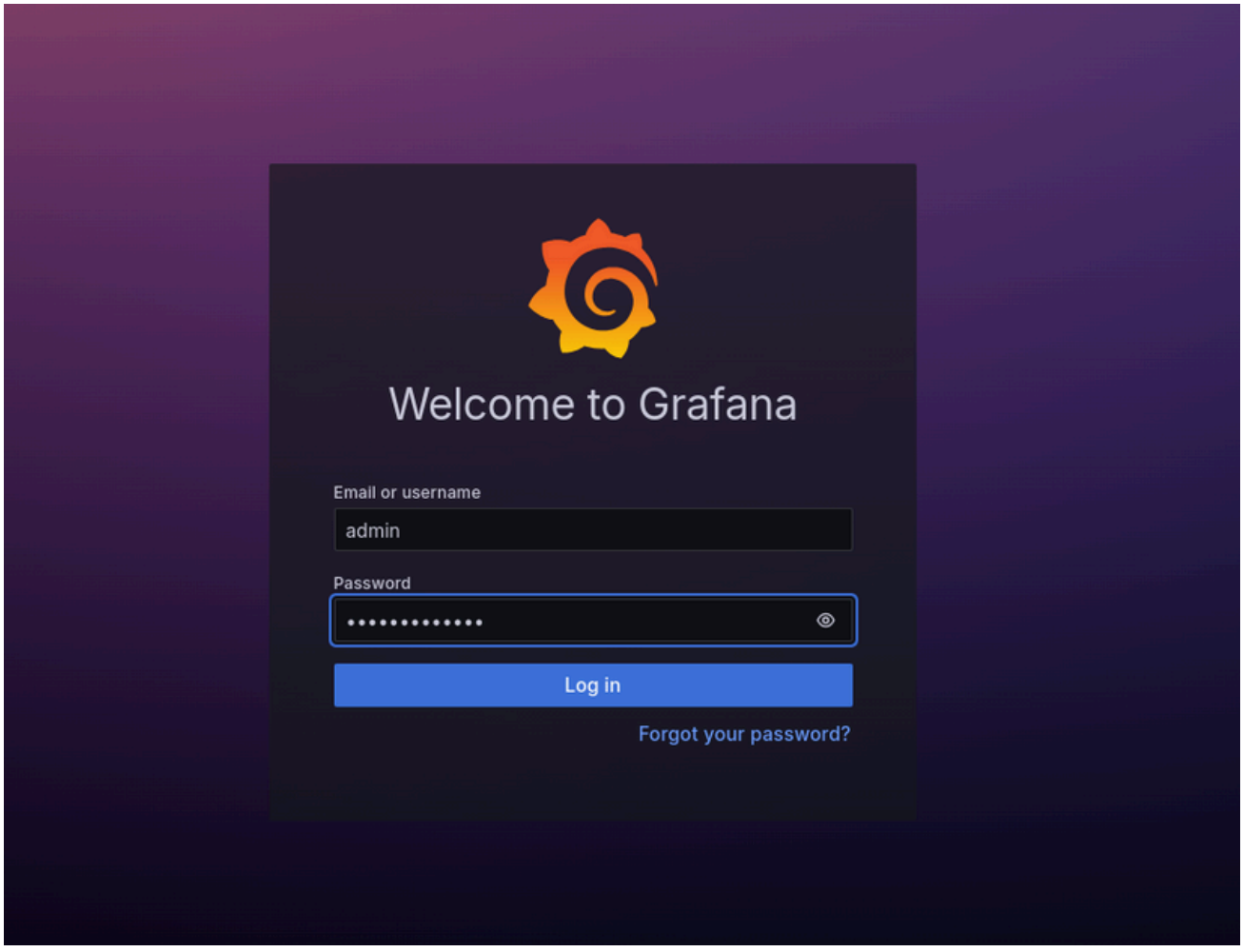
- Declarative management of PostgreSQL configuration, including certain popular Postgres - extensions through the cluster spec: pgaudit, auto_explain, pg_stat_statements, and pg_failover_slots
- Declarative management of Postgres databases
- Support for Local Persistent Volumes with PVC templates
- Reuse of Persistent Volumes storage in Pods
- Separate volumes for WAL files and tablespaces
- Rolling updates for PostgreSQL minor versions
- TLS connections and client certificate authentication
- Offline, Offline import of existing PostgreSQL databases, including major upgrades of PostgreSQL
- Backup from a standby
- Backup retention policies (based on recovery window, only on object stores)
- enabling private, public, hybrid, and multi-cloud architectures with support for controlled switchover.
- Connection pooling with PgBouncer
- cnpg plugin for kubectl
- Simple bind and search+bind LDAP client authentication
- Multi-arch format container images

8) Monitoring

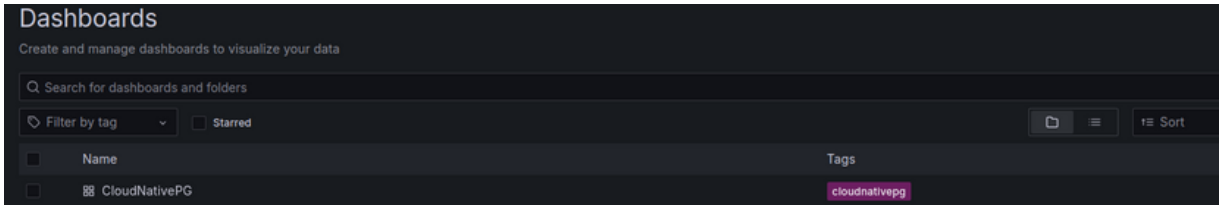
port forward grafana, to acces it on your pc:

```
kubectl --namespace observability port-forward \
  service/prometheus-community-grafana 8080:80
```

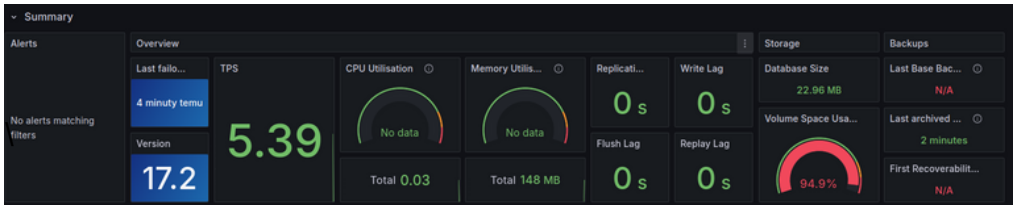
Open http://localhost:8080 in a browser and Use `admin` as the username and `prom-operator` as the password.



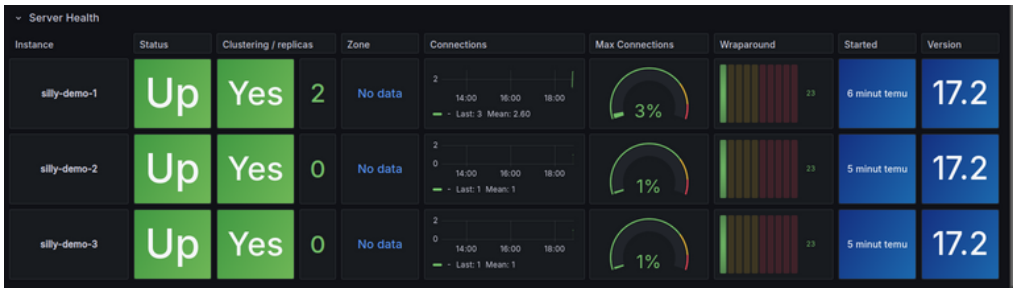
The CloudNativePG dashboard provides all the information you need to understand the performance and health of your cluster!



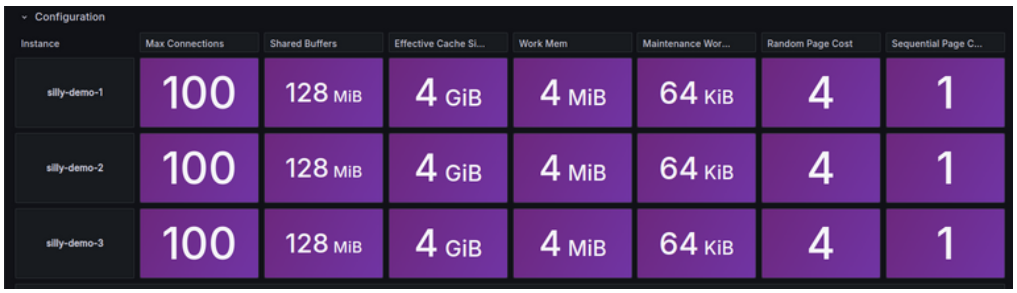
A general overview includes key aspects of resource utilization, backups, latency, past performance, and versioning:



The current database setup provides insights into configuration and performance, helping to quickly identify potential real-time issues



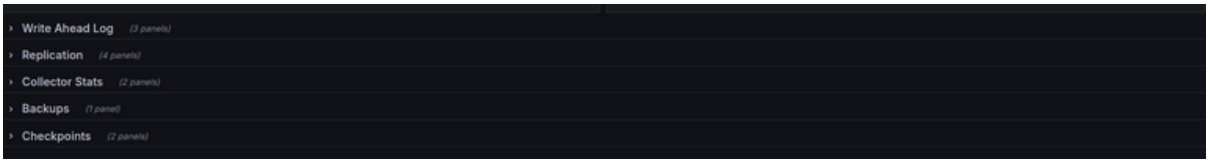
The current database setup enables quick identification of potential real-time configuration-related performance issues



Database-specific storage allocation metrics

Database specific storage read/write metrics

And much more!



Kubernetes Databases: DB Schema management via Atlas operator

1) Intro

Managing database schemas effectively in a dynamic, containerized environment like Kubernetes can be challenging. The Atlas Kubernetes Operator is a powerful tool designed to simplify this process. It automates the lifecycle of database schemas, ensuring that schema management is integrated seamlessly into Kubernetes workflows.

2) Setup

Download repo

```
git clone https://github.com/vfarcic/atlas-kubernetes-demo

cd atlas-kubernetes-demo
```

Install cnpg and atlas-operator

```
helm upgrade --install cnpg cloudnative-pg \
  --repo https://cloudnative-pg.github.io/charts \
  --namespace cnpg-system --create-namespace --wait
```

```
helm upgrade --install atlas-operator \
  oci://ghcr.io/ariga/charts/atlas-operator \
  --namespace atlas-operator --create-namespace --wait

kubectl create namespace silly-demo
```

apply db

```
kubectl --namespace silly-demo apply --filename db.yaml
```

3) Atlas operator

Lets check Atlas operator yaml:

```
cat videos.yaml
apiVersion: db.atlasgo.io/v1alpha1
kind: AtlasSchema
metadata:
  name: silly-demo-videos
spec:
  credentials:
    scheme: postgres
    host: silly-demo-rw.silly-demo
    port: 5432
    user: app
    passwordFrom:
      secretKeyRef:
        key: password
        name: silly-demo-app
    database: app
    parameters:
      sslmode: disable
```

5

Atlas operator definition specifies the database credentials, including the host (silly-demo-rw.silly-demo), port (5432), username (app), and a password retrieved from a secret (silly-demo-app, key: password). The database name is app, and SSL mode is disabled.

Then, we have schema we want our database to update with:

```
schema:
  sql: |
    create table videos (
      id varchar(50) not null,
      title varchar(255) not null,
      primary key (id)
    );
    create table comments (
      id serial,
      video_id varchar(50) not null,
      description text not null,
      primary key (id),
      CONSTRAINT fk_videos FOREIGN KEY(video_id) REFERENCES videos(id)
    );
```

To apply a schema, the Atlas operator generates SQL queries for execution. While Atlas natively supports its HCL language, using pure SQL in Kubernetes environments is[5] often more practical and efficient.

It’s important to note that directly applying schema changes to the database can cause errors, such as attempting to create a table that already exists. The Atlas operator addresses this issue by creating an ephemeral database where schema changes are first applied. These changes are then validated and safely propagated to the actual database, ensuring error-free updates.

apply:

```
kubectl --namespace silly-demo apply --filename videos-1.yaml
```

and check state

```
kubectl --namespace silly-demo get atlasschemas
```

```
laptopdev@laptopdev2:~/atlas-kubernetes-demo$ kubectl --namespace silly-demo get atlasschemas
NAME                READY   REASON
silly-demo-videos   False   GettingDevDB
laptopdev@laptopdev2:~/atlas-kubernetes-demo$ kubectl --namespace silly-demo get atlasschemas
NAME                READY   REASON
silly-demo-videos   True    Applied
laptopdev@laptopdev2:~/atlas-kubernetes-demo$
```

The schema has been successfully applied. Open a second terminal session to verify that the changes have been reflected in the database. Additionally, check the database pod logs to confirm that the updates were processed correctly.

```
kubectl --namespace silly-demo exec -it silly-demo-1 -- sh
```

open postgresql shell

```
psql
```

log into app db

```
\c app
```

list tables

```
\dt
```

```
laptopdev@laptopdev2:~/atlas-kubernetes-demo$ kubectl --namespace silly-demo exec -it silly-demo-1 -- sh
Defaulted container "postgres" out of: postgres, bootstrap-controller (init)
$ psql
psql (17.2 (Debian 17.2-1.pgdg110+1))
Type "help" for help.

postgres=# \c app
You are now connected to database "app" as user "postgres".
app=# \dt
         List of relations
 Schema |   Name   | Type  | Owner
--------+----------+-------+-------
 public | comments | table | app
 public | videos   | table | app
(2 rows)
```

Great! tabels created!

While the Atlas operator offers significant benefits, it does come with a few drawbacks:

Configuration for Large Datasets: Defining SQL queries as configurations can be impractical for large datasets. If your clusters are dedicated solely to databases, consider exploring tools better suited for such use cases.

Limitations with Deleting Tables: The Atlas operator cannot delete tables that have already been created. Deleting a schema does not remove tables; it only allows creation and alterations to existing ones.

Lets apply modified videos-1.yaml schema. Modification, is a name of field descirpitoim:

```
diff videos-1.yaml videos-2.yaml
```

```
laptopdev@laptopdev2:~/atlas-kubernetes-demo$ diff videos-1.yaml videos-2.yaml
23a24
>         description text,
laptopdev@laptopdev2:~/atlas-kubernetes-demo$ cat videos-2.yaml
```

check to be sure if there is description column already:

```
app=# \d videos
                  Table "public.videos"
 Column |          Type          | Collation | Nullable | Default
--------+------------------------+-----------+----------+---------
 id     | character varying(50)  |           | not null |
 title  | character varying(255) |           | not null |
Indexes:
    "videos_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "comments" CONSTRAINT "fk_videos" FOREIGN KEY (video_id) REFERENCES videos(id)
```

There is not. Lets apply modified atlas schema:

```
kubectl --namespace silly-demo apply --filename videos-2.yaml
```

Check tables again:

```
\d videos
```

```
app=# \d videos
                      Table "public.videos"
   Column    |          Type          | Collation | Nullable | Default
-------------+------------------------+-----------+----------+---------
 id          | character varying(50)  |           | not null |
 title       | character varying(255) |           | not null |
 description | text                   |           |          |
Indexes:
    "videos_pkey" PRIMARY KEY, btree (id)
Referenced by:
    TABLE "comments" CONSTRAINT "fk_videos" FOREIGN KEY (video_id) REFERENCES videos(id)
```

Great! table was altered. We can check it in atlasschema description:

```
kubectl --namespace silly-demo describe \
atlasschema silly-demo-videos
```

```
    Last Transition Time:  2025-01-13T23:29:18Z
    Message:               The schema has been applied successfully. Apply response: {
lly-demo-rw.silly-demo:5432","Path":"/app","RawPath":"","OmitHost":false,"ForceQuery":
"Changes":{"Applied":["ALTER TABLE \"public\".\"videos\" ADD COLUMN \"description\" te
.917339521Z","Applied":{"Name":"20250113232918.sql","Version":"20250113232918","Start"
ed":0,"Applied":["ALTER TABLE \"public\".\"videos\" ADD COLUMN \"description\" text NU
heme":"postgres","Opaque":"","User":{},"Host":"silly-demo-rw.silly-demo:5432","Path":"
ble","Fragment":"","RawFragment":"","Schema":""}},"File":{"Name":"20250113232918","Fro
57NplHO2GT6h0fwywP2f/ms1FDYk=","Migration":"-- Modify \"videos\" table\nALTER TABLE \"
,"Text":"ALTER TABLE \"public\".\"videos\" ADD COLUMN \"description\" text NULL;","Com
    Reason:                Applied
```

At the end of Apply response, we can see ALTER TABLE query with description column

common troubleshooting

1) HA PostgreSQL Deployment Fails

Cause: Misconfigured CNPG cluster resource definition.
Solution: Verify the cluster CRD with kubectl describe and ensure the pg_hba and storage settings in the manifest are correct.

2) Schema Migration via Atlas Fails

Cause: Incorrect database connection settings or schema drift.
Solution: Test connectivity with atlas schema apply --dry-run and validate the target database's state using atlas schema inspect.

3) Backup Creation Error

Cause: Misconfigured backup storage (e.g., S3 bucket or PVC).
Solution: Check the CNPG configuration for backup paths, and validate access with kubectl logs on the backup job pod.

4) Check my Kubernetes Troubleshooting series:

Michael Robotics
Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics

Learn more about Kubernetes

Check Kubernetes and piyushsachdeva - great docs!

Setup a Multi Node Kubernetes Cluster

kubeadm is a tool to bootstrap the Kubernetes cluster

https://github.com/piyushsachdeva/CKA-2024/tree/main/Resources/Day27

Kubernetes Documentation

This section lists the different ways to set up and run Kubernetes

https://kubernetes.io/docs/setup/

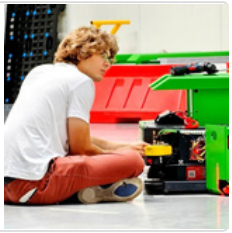
Share, comment, DM and check GitHub for scripts & playbooks created to automate process.

Check my GitHub

Michael Robotics

Hi, I'm Michal. I'm a Robotics Engineer and DevOps enthusiast. My mission is to create skill-learning platform that combats skill information overload by adhering to the set of principles: simplify, prioritize, and execute.

https://github.com/MichaelRobotics



PS.

If you need a playbook or bash script to manage KVM on a specific Linux distribution, feel free to ask me in the comments or send a direct message!