Redefining High Availability for PostgreSQL on Kubernetes with Lightbits RWX

Implementation Guide for High Availability PostgreSQL on Kubernetes in Active/Suspend Mode

March 2026

Author: Rob Bloemendal, Principal Solution Consultant

Abstract

This white paper illustrates a high-performance, distributed architectural framework utilizing the ReadWriteMany (RWX) primitives of the Lightbits disaggregated, software-defined storage (SDS) system to implement a resilient Active/Passive High Availability (HA) substrate for PostgreSQL within Kubernetes. By harnessing native clustered NVMe-over-TCP (NVMe/TCP) protocols, the platform facilitates deterministic volume failover and continuous data persistence across disparate Kubernetes nodes, abstracting the underlying storage complexities from the compute layer.

The integration model prioritizes mitigating storage-tier vulnerabilities, ensuring that mission-critical stateful workloads execute automated reconciliation on secondary nodes with zero data corruption and minimal recovery time objectives (RTO). The technical discourse details the provisioning of Lightbits RWX volumes, the application of pod anti-affinity scheduling constraints for optimized orchestration, and the specific PostgreSQL container parameters required for rapid failover. Ultimately, this synergy illustrates the convergence of enterprise-grade reliability and low-latency performance by integrating Lightbits' shared-storage efficiencies with the dynamic orchestration capabilities of a Kubernetes-native PostgreSQL cluster.

Table of Contents

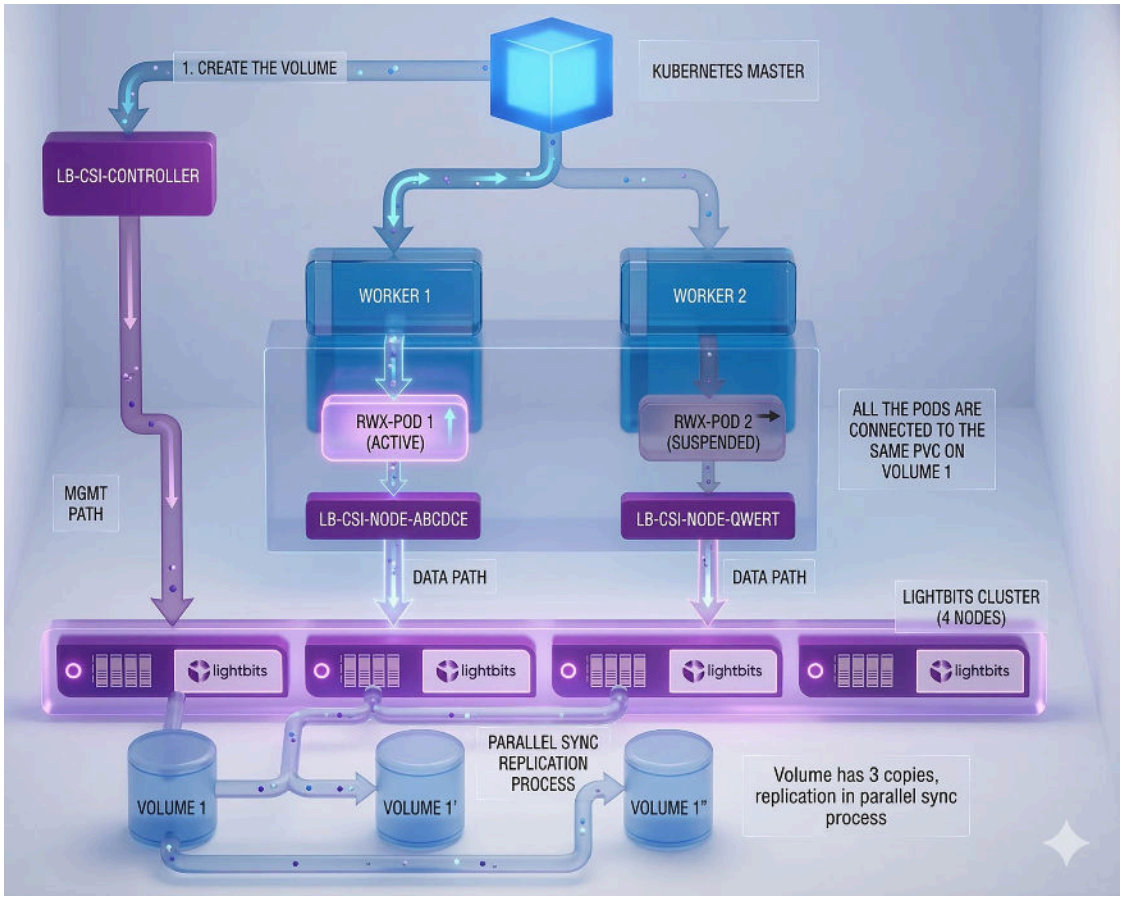
1. Introduction

This white paper provides a comprehensive overview and technical guidelines for implementing a High Availability (HA) Active/Passive environment for PostgreSQL on a Kubernetes cluster. By leveraging the ReadWriteMany (RWX) capabilities of the Lightbits Cloud-Native Storage Interface (CSI), organizations can achieve seamless failover with zero downtime from PostgreSQL active/passive clusters on Kubernetes. The paper provides an example of how to achieve HA when the failover time is critical and must be as short as possible (failover duration is the time it takes to migrate pod (s) from one worker node to another), and when the application controls it.

1.1. Architectural Overview

The solution architecture consists of a standard Kubernetes cluster configuration optimized for storage resilience:

- **Control Plane:** One master node managing cluster orchestration.
- **Data Plane:** Two distinct worker nodes providing the execution environment for workloads.
- **Workload:** One pod deployed on each worker node (active and passive instances).
- **Storage Layer:** Both pods share a single Persistent Volume Claim (PVC) backed by the Lightbits storage cluster.



1.2. Key Technical Components

To ensure non-disruptive storage handovers, the following components are utilized.

1.2.1. Lightbits CSI Driver

The Lightbits CSI driver facilitates the communication between Kubernetes and the Lightbits storage cluster. It enables dynamic provisioning of volumes that support high-performance NVMe/TCP.

1.2.2. RWX Access Mode

Unlike standard ReadWriteOnce (RWO) volumes - which limit mounting to a single node - RWX volumes allow simultaneous mounting by multiple pods across different nodes. This is the "secret sauce" for Active/Passive HA: since the volume is already attached to the passive node, there is no "detach/attach" latency during a failover event.

1.2.3. PVC

The PVC acts as the abstract request for storage. In this architecture, the PVC points to a Lightbits StorageClass configured for RWX. Both the active and passive pods reference the same PVC, ensuring that they see the same dataset in real time.

1.3. Implementation Workflow

1. **Permissions:** Creating a service account to manage the leader.
2. **Leader Election Lock:** Locking down which pod is in the lead.
3. **Service:** The HA service for PostgreSQL.
4. **PVC Creation:** Provision a PVC with accessModes: [RWX].
5. **Initialization Job:** Install PostgreSQL, mount the volume, and create an ext4 filesystem.
6. **Create POD 1:** Create POD pg-worker-1, and if no lead yet, become the lead and start the PostgreSQL database.
7. **Create POD 2:** Create POD pg-worker-2, and if no lead yet, become the lead and start the PostgreSQL database.
8. **Failover Logic:** Delete the POD pg-worker-1 (current lead), while running insert activities from another POD as the driving application workload.

Benefits of this Approach

FEATURE	TECHNICAL IMPACT
Zero Storage Latency	No waiting for the volume to detach from a failed node.
Data Consistency	Shared block storage ensures that the passive node has the exact state of the active node.
Simplified Recovery	Eliminates complex "Force Detach" operations in Kubernetes.

2. Environment Initialization and Container Orchestration

The foundational step in this deployment is preparing a specialized container image that manages ext4 file systems and storage utilities. We chose AlmaLinux 9.5 as the base operating system for its enterprise-grade stability and compatibility.

2.1. Container Image Construction

To handle file system operations within the Kubernetes pods, we constructed a custom Docker image. The Dockerfile includes the necessary binaries for ext4 manipulation and volume management:

```
Shell
FROM almalinux:9.5

# Install all required tools permanently
RUN dnf install -y --allowerasing \
    curl \
    xfsprogs \
    util-linux \
    python3 \
    e2fsprogs \
    && dnf clean all

# Set the path to ensure system binaries are always available
ENV PATH="/usr/sbin:/sbin:/usr/local/sbin:${PATH}"

ENTRYPOINT ["/bin/bash"]
```

The image was built locally on the master node using the following command:

```
Shell
sudo docker build -t almalinux:latest .
```

2.2. Image Distribution Workflow

In environments where a central container registry is unavailable or for initial staging, we used a manual distribution method to ensure image parity across the cluster.

Step A: Export and Compression. The image was serialized into a compressed tarball to facilitate transport:

```
Shell
sudo docker save almalinux:latest | gzip > almalinux.tar.gz
```

Step B: Network Transfer. The archive was securely transferred from the master node to both **Worker 1** and **Worker 2** via SCP:

```
Shell
scp almalinux.tar.gz demo@worker1:
scp almalinux.tar.gz demo@worker2:
```

Step C: Image Ingestion. On each worker node, the image was loaded into the local Docker engine:

```
Shell
sudo docker load < ./almalinux.tar.gz
```

2.3. Verification of Node Readiness

To confirm that the workers were prepared for the HA workload, we verified the presence of the image using the Docker CLI:

```
Shell
sudo docker images
```

Expected Output:

```
Shell
IMAGE              ID             DISK USAGE   CONTENT SIZE   EXTRA
almalinux:latest   8e43303e302a       373MB         94.5MB
```

With the standardized runtime environment installed on all worker nodes, the infrastructure is now ready for the Kubernetes HA configuration and the integration of Lightbits RWX volumes.

3. Kubernetes High Availability Configuration

To implement the Active/Passive failover logic, we define a unified manifest comprising the RBAC permissions, storage requirements, and workload specifications. This configuration ensures that Kubernetes manages stateful transitions without corruption of storage.

3.1. Role-Based Access Control (RBAC) and Leader Election

The cornerstone of this HA architecture is the Leader Election mechanism. We use a dedicated ServiceAccount and Role to allow the pods to interact with Kubernetes ConfigMaps. These ConfigMaps act as a distributed lock; whichever pod holds the lock is the "Active" node, while the other remains "Suspended":

```
Shell
# 1. PERMISSIONS
apiVersion: v1
kind: ServiceAccount
metadata:
  name: leader-election-sa
---
apiVersion: rbac.authorization.k8s.io/v1
kind: Role
metadata:
  name: leader-election-role
rules:
  - apiGroups: [""]
    resources: ["configmaps", "pods"]
    verbs: ["get", "watch", "list", "create", "update", "patch", "delete"]
---
apiVersion: rbac.authorization.k8s.io/v1
```

```
kind: RoleBinding
metadata:
  name: leader-election-rolebinding
subjects:
  - kind: ServiceAccount
    name: leader-election-sa
roleRef:
  kind: Role
  name: leader-election-role
  apiGroup: rbac.authorization.k8s.io
```

3.2. Leader Election Lock

Having a leader election is important, but it also needs to be locked. To avoid having two leaders:

Shell

```
# 2. LEADER ELECTION LOCK
apiVersion: v1
kind: ConfigMap
metadata:
  name: block-device-lock
data:
  holder: ""
  renewTime: "0"
```

3.3. Service (Traffic Entry Point)

Creating an entry point is important. The applications continuously write to the cluster name and not to the individual pod:

Shell

```
# 3. SERVICE
apiVersion: v1
kind: Service
metadata:
  name: postgres-ha-svc
spec:
  ports:
```

```
    - port: 5432
      targetPort: 5432
  selector:
    app: postgres-ha
    role: active-leader
```

3.4. Storage Definition (PVC)

The PVC is configured in Block Mode with RWX access. Using volumeMode: Block allows the application to interact directly with the raw device provided by Lightbits, which is critical for high-performance ext4 management:

```
Shell
# 4. STORAGE
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
spec:
  storageClassName: "example-sc"
  accessModes: ["ReadWriteMany"]
  volumeMode: Block
  resources:
    requests:
      storage: 20Gi
```

3.5. Initialization job

The initialization job will take care of the mounting of the PVC, creating a filesystem on the PVC, installing PostgreSQL, and finally creating a database on top of that new filesystem:

```
Shell
# 5. INITIALIZATION JOB
apiVersion: batch/v1
kind: Job
metadata:
  name: postgres-init-job
spec:
```

```
  template:
    spec:
      containers:
      - name: init
        image: docker.io/library/almalinux:latest
        securityContext:
          privileged: true
        command: ["/bin/bash", "-c"]
        args:
          - |
            DEV="/dev/lbcsiblkdev"
            MNT="/mnt/data"
            dnf install -y e2fsprogs util-linux postgresql-server > /dev/null
2>&1
            mkdir -p $MNT
            if ! blkid $DEV | grep -q "TYPE="; then
              mkfs.ext4 -F $DEV
            fi
            mount $DEV $MNT
            if [ ! -d "$MNT/pgdata" ]; then
              mkdir -p $MNT/pgdata
              chown -R postgres:postgres $MNT
              su - postgres -c "initdb -D /mnt/data/pgdata"
              echo "host all all 0.0.0.0/0 trust" >>
/mnt/data/pgdata/pg_hba.conf
              echo "listen_addresses = '*'" >> /mnt/data/pgdata/postgresql.conf
            fi
            sync; umount $MNT
        volumeDevices:
          - name: block-data
            devicePath: /dev/lbcsiblkdev
      restartPolicy: OnFailure
      volumes:
        - name: block-data
          persistentVolumeClaim:
            claimName: postgres-pvc
```

3.6. Create POD 1

The pod specifications are designed to handle the transitions between active and passive states. Each pod is pinned to a specific worker node to ensure physical redundancy. The Pod will install PostgreSQL on its own. It will check whether it owns the Lead and then mount the filesystem. The database is automatically created.

The operational logic is as follows:

1. **Node Affinity**: Explicitly targets a specific worker node.
2. **Leader Election**: The pod checks its status via the ServiceAccount. If it is not the leader, it remains in a "hot standby" state.
3. **Active Transition**: Once a pod acquires leadership:
    - It scans the block device for an existing ext4 signature.
    - If no filesystem exists, it initializes the ext4 volume.
    - It then mounts the ext4 filesystem to the designated mount point.
4. **Graceful Termination/Suspension**: When a leader yields their position:
    a. The system executes a memory-to-disk flush (fsync).
    b. A full synchronization is performed to ensure data integrity.
    c. The ext4 volume is unmounted, though the block device remains connected to the PVC for immediate takeover.
5. **Application Transactions**: These will remain in the timeout buffer built into the Linux kernels for the write activity. There will be a suspend period, but the timeout buffer on Linux will avoid data loss. The handover - by simply deleting the pod - is completely handled by Kubernetes. Kubernetes, with the POD definition, will ensure that the data is synced to disk before the POD is closed.

Configuration for pg-worker-1:

```shell
# 6. WORKER POD 1
apiVersion: v1
kind: Pod
metadata:
  name: pg-worker-1
  labels:
  containers:
  - name: workload
    image: docker.io/library/almalinux:latest
    securityContext:
```

```
      privileged: true
    lifecycle:
      preStop:
        exec:
    command: ["/bin/bash", "-c"]
    args:
      - |
        # Wait for Job to finish
        LOCK_CM="block-device-lock"
        NS=$(cat /var/run/secrets/kubernetes.io/serviceaccount/namespace)
        TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
        K8S_API="https://kubernetes.default.svc/api/v1/namespaces/$NS"

        dnf install -y e2fsprogs util-linux postgresql-server python3 >
/dev/null 2>&1
        mkdir -p $MNT

        while true; do
          NOW=$(date +%s)
          CM_DATA=$(curl -s -k -H "Authorization: Bearer $TOKEN"
"$K8S_API/configmaps/$LOCK_CM")
          HOLDER=$(parse_json "$CM_DATA" "holder" "data")
          LAST_RENEW=$(parse_json "$CM_DATA" "renewTime" "data")
          REV=$(parse_json "$CM_DATA" "resourceVersion" "metadata")

          if [ -z "$HOLDER" ] || [ "$HOLDER" == "$HOSTNAME" ] || [ $((NOW -
LAST_RENEW)) -gt 25 ]; then

PAYLOAD="{\"metadata\":{\"resourceVersion\":\"$REV\"},\"data\":{\"holder\":\"$H
OSTNAME\",\"renewTime\":\"$NOW\"}}"
            RESPONSE=$(curl -s -k -X PATCH -H "Authorization: Bearer $TOKEN" -H
"Content-Type: application/merge-patch+json" --data "$PAYLOAD"
"$K8S_API/configmaps/$LOCK_CM")
            HOLDER=$(parse_json "$RESPONSE" "holder" "data")
          fi

          if [ "$HOLDER" == "$HOSTNAME" ]; then
            if [ -b "$DEV" ] && mount -t ext4 -o sync,noatime $DEV $MNT; then
                rm -f $PGDATA/postmaster.pid
                if ! su - postgres -c "pg_ctl status -D $PGDATA" >/dev/null
2>&1; then
                   mkdir -p /var/run/postgresql && chown postgres:postgres
/var/run/postgresql
```

```
                    su - postgres -c "pg_ctl start -D $PGDATA -o '-c
listen_addresses=\"*\"' -l /tmp/pg.log"
                    curl -s -k -X PATCH -H "Authorization: Bearer $TOKEN" -H
"Content-Type: application/strategic-merge-patch+json" --data
'{"metadata":{"labels":{"role":"active-leader"}}}' "$K8S_API/pods/$HOSTNAME" >
/dev/null
                fi
              fi
            else
              curl -s -k -X PATCH -H "Authorization: Bearer $TOKEN" -H
"Content-Type: application/strategic-merge-patch+json" --data
'{"metadata":{"labels":{"role":null}}}' "$K8S_API/pods/$HOSTNAME" > /dev/null
              if su - postgres -c "pg_ctl status -D $PGDATA" >/dev/null 2>&1;
then su - postgres -c "pg_ctl stop -D $PGDATA -m fast"; fi
              if mountpoint -q $MNT; then sync; umount -l $MNT; fi
            fi
            sleep 5
          done
      volumeDevices: [{name: block-data, devicePath: /dev/lbcsiblkdev}]
    volumes:
      - name: block-data
        persistentVolumeClaim:
          claimName: postgres-pvc
```

3.7. Create POD 2

The pod specifications are designed to handle the transitions between active and passive states. Each pod is pinned to a specific worker node to ensure physical redundancy. The Pod will install PostgreSQL on its own. It will check whether it owns the Lead and then mount the filesystem. The database is automatically created.

The operational logic is as follows:
Node Affinity: Explicitly targets a specific worker node.
Leader Election: The pod checks its status via the ServiceAccount. If it is not the leader, it remains in a "hot standby" state.
Active Transition: Once a pod acquires leadership:

a. It scans the block device for an existing ext4 signature.
b. If no filesystem exists, it initializes the ext4 volume.
c. It mounts the ext4 filesystem to the designated mount point.

Graceful Termination/Suspension: When a leader yields their position:

a. The system executes a memory-to-disk flush (fsync).
b. A full synchronization is performed to ensure data integrity.
c. The ext4 volume is unmounted, though the block device remains connected to the PVC for immediate takeover.

Application Transactions: These will remain in the timeout buffer built into the Linux kernel for the write activity. There will be a suspension period, but the Linux timeout buffer will prevent data loss. The handover, by simply deleting the pod, is completely handled by Kubernetes. Kubernetes, with the POD definition, ensures that data is synced to disk before the POD is closed.
Configuration for pg-worker-2:

```
Shell
# 7. WORKER POD 2
apiVersion: v1
kind: Pod
metadata:
  name: pg-worker-2
  labels:
    app: postgres-ha
spec:
  serviceAccountName: leader-election-sa
  nodeName: worker2
  containers:
  - name: workload
    image: docker.io/library/almalinux:latest
    securityContext:
      preStop:
        exec:
    command: ["/bin/bash", "-c"]
        sleep 30
        MNT="/mnt/data"
        DEV="/dev/lbcsiblkdev"
        PGDATA="$MNT/pgdata"
        LOCK_CM="block-device-lock"
        NS=$(cat /var/run/secrets/kubernetes.io/serviceaccount/namespace)
        TOKEN=$(cat /var/run/secrets/kubernetes.io/serviceaccount/token)
        K8S_API="https://kubernetes.default.svc/api/v1/namespaces/$NS"
        dnf install -y e2fsprogs util-linux postgresql-server python3 >
/dev/null 2>&1
        mkdir -p $MNT
```

```
        parse_json() { echo "$1" | python3 -c "import sys, json;
data=json.load(sys.stdin); print(data.get('data', {}).get('$2', '') if
'$3'=='data' else data.get('metadata', {}).get('$2', ''))"; }
        while true; do
          NOW=$(date +%s)
          CM_DATA=$(curl -s -k -H "Authorization: Bearer $TOKEN"
"$K8S_API/configmaps/$LOCK_CM")
          HOLDER=$(parse_json "$CM_DATA" "holder" "data")
          LAST_RENEW=$(parse_json "$CM_DATA" "renewTime" "data")
          REV=$(parse_json "$CM_DATA" "resourceVersion" "metadata")
          if [ -z "$HOLDER" ] || [ "$HOLDER" == "$HOSTNAME" ] || [ $((NOW -
LAST_RENEW)) -gt 25 ]; then

PAYLOAD="{\"metadata\":{\"resourceVersion\":\"$REV\"},\"data\":{\"holder\":\"$H
OSTNAME\",\"renewTime\":\"$NOW\"}}"
            RESPONSE=$(curl -s -k -X PATCH -H "Authorization: Bearer $TOKEN" -H
"Content-Type: application/merge-patch+json" --data "$PAYLOAD"
"$K8S_API/configmaps/$LOCK_CM")
            HOLDER=$(parse_json "$RESPONSE" "holder" "data")
          fi
          if [ "$HOLDER" == "$HOSTNAME" ]; then
            if [ -b "$DEV" ] && mount -t ext4 -o sync,noatime $DEV $MNT; then
                rm -f $PGDATA/postmaster.pid
                if ! su - postgres -c "pg_ctl status -D $PGDATA" >/dev/null
2>&1; then
                   mkdir -p /var/run/postgresql && chown postgres:postgres
/var/run/postgresql
                   su - postgres -c "pg_ctl start -D $PGDATA -o '-c
listen_addresses=\"*\"' -l /tmp/pg.log"
                   curl -s -k -X PATCH -H "Authorization: Bearer $TOKEN" -H
"Content-Type: application/strategic-merge-patch+json" --data
'{"metadata":{"labels":{"role":"active-leader"}}}' "$K8S_API/pods/$HOSTNAME" >
/dev/null
                fi
            fi
          else
            curl -s -k -X PATCH -H "Authorization: Bearer $TOKEN" -H
"Content-Type: application/strategic-merge-patch+json" --data
'{"metadata":{"labels":{"role":null}}}' "$K8S_API/pods/$HOSTNAME" > /dev/null
            if su - postgres -c "pg_ctl status -D $PGDATA" >/dev/null 2>&1;
then su - postgres -c "pg_ctl stop -D $PGDATA -m fast"; fi
            if mountpoint -q $MNT; then sync; umount -l $MNT; fi
          fi
          sleep 5
```

```
        done
    volumeDevices: [{name: block-data, devicePath: /dev/lbcsiblkdev}]
  volumes:
    - name: block-data
      persistentVolumeClaim:
        claimName: postgres-pvc
```

4. Deployment and Failover Validation

To validate the high-availability architecture, we perform a series of tests demonstrating automated storage provisioning, leader election, and non-disruptive data migration between nodes. At the same time, records are injected into the PostgreSQL database cluster.

4.1. Provisioning the Pods and RWX Storage

Starting from an empty namespace, we apply the PostgresActiveStandby.yaml manifest. This triggers the concurrent creation of the RBAC roles, the Lightbits-backed PVC, and the two worker pods:

Shell

```
kubectl create -f PostgresActiveStandby.yaml
serviceaccount/leader-election-sa created
role.rbac.authorization.k8s.io/leader-election-role created
rolebinding.rbac.authorization.k8s.io/leader-election-rolebinding created
configmap/block-device-lock created
service/postgres-ha-svc created
persistentvolumeclaim/postgres-pvc created
job.batch/postgres-init-job created
pod/pg-worker-1 created
pod/pg-worker-2 created
```

Once applied, we verify that the scheduler has correctly distributed the workloads across different physical nodes:

Shell

```
kubectl get pods -o wide
NAME                    READY   STATUS      RESTARTS   AGE     IP           NODE      NOMINATED NODE   READINESS
GATES
```

```
pg-worker-1              1/1     Running     0          3m49s   10.244.2.131   worker1   <none>           <none>
pg-worker-2              1/1     Running     0          3m49s   10.244.1.201   worker2   <none>           <none>
postgres-init-job-rfpww  0/1     Completed   0          3m49s   10.244.2.130   worker1   <none>           <none>
```

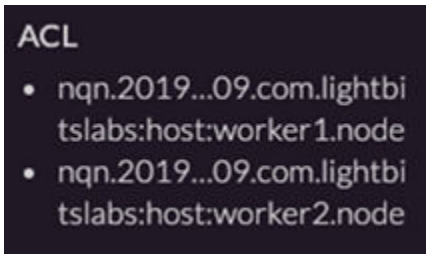
The postgres-init-job has also run successfully. To check the cluster service and which node is in the lead:

Shell

```
kubectl get endpoints postgres-ha-svc
NAME              ENDPOINTS           AGE
postgres-ha-svc   10.244.2.131:5432   4m48s
```

In this case, pg-worker-1 is in the lead.

At this stage, both worker nodes are physically connected to the Lightbits PVC via NVMe/TCP, but only the active node has the ext4 mount point engaged. This is illustrated in the image below from the Lightbits Photon GUI.



4.2. Testing Failover and Data Integrity

The ultimate test of a storage-centric HA system is the ability to persist data across a "handover" event.

Step 1: Create a New Table in the Database

To create a new table in the database, another pod has been used to write to the cluster service. The following script has been used:

Shell

```
kubectl run pg-admin-client --rm -i --restart=Never
--image=docker.io/library/postgres:latest \
```

```
  -- psql -h postgres-ha-svc -U postgres -d postgres -c "CREATE TABLE IF NOT
EXISTS cluster_test (msg text);"
```

Output:

```
Shell
CREATE TABLE
pod "pg-admin-client" deleted
```

To check that the table exists (in this case, pg-worker-1):

```
Shell
kubectl exec pg-worker-1 -- su - postgres -c "psql -d postgres -c '\dt'"
```

Output:

```
Shell
           List of relations
 Schema |     Name     | Type  |  Owner
--------+--------------+-------+----------
 public | cluster_test | table | postgres
(1 row)
```

Step 2: Inserting a Record into the New Table

Before we delete the active pod, starting inserts into the table are essential. The inserts will continue to run while we delete the active pod. A new pod will be created every time a new insert is done. This represents application load and external connectivity:

```
Shell
for i in {1..10000}; do
  # Try to insert, if it fails, wait 5 seconds and try once more
  until echo "INSERT INTO cluster_test VALUES ('Success on record $i');" | \
```

```
    kubectl run pg-client-$i --rm -i --restart=Never
--image=docker.io/library/postgres:latest \
    -- psql -h postgres-ha-svc -U postgres -d postgres; do
    echo "Failover in progress... retrying in 5s"
    sleep 5
  done
done
```

Output from the cluster:

```
Shell
INSERT 0 1
pod "pg-client-1" deleted
INSERT 0 1
pod "pg-client-2" deleted
```

Step 3: Executing the Failover

The failover is done by a hard delete of the active pod; in this case, pg-worker-1. At the same time, the insert keeps going:

```
Shell
kubectl delete pod pg-worker-1
```

Output:

```
Shell
pod "pg-worker-1" deleted
```

In the meantime, the script for inserting the records is showing an error (see pg-client-22). It keeps retrying until pg-worker-2 has taken over the lead and is running the PostgreSQL database as active:

```shell
pod "pg-client-20" deleted
If you don't see a command prompt, try pressing enter.
INSERT 0 1
pod "pg-client-21" deleted
psql: error: connection to server at "postgres-ha-svc" (10.111.255.201), port
5432 failed: Connection refused
	Is the server running on that host and accepting TCP/IP connections?
pod "pg-client-22" deleted
pod default/pg-client-22 terminated (Error)
Failover in progress... retrying in 5s
psql: error: connection to server at "postgres-ha-svc" (10.111.255.201), port
5432 failed: Connection refused
	Is the server running on that host and accepting TCP/IP connections?
pod "pg-client-22" deleted
pod default/pg-client-22 terminated (Error)
Failover in progress... retrying in 5s
psql: error: connection to server at "postgres-ha-svc" (10.111.255.201), port
5432 failed: Connection refused
	Is the server running on that host and accepting TCP/IP connections?
pod "pg-client-22" deleted
pod default/pg-client-22 terminated (Error)
Failover in progress... retrying in 5s
psql: error: connection to server at "postgres-ha-svc" (10.111.255.201), port
5432 failed: Connection refused
	Is the server running on that host and accepting TCP/IP connections?
pod "pg-client-22" deleted
pod default/pg-client-22 terminated (Error)
Failover in progress... retrying in 5s
psql: error: connection to server at "postgres-ha-svc" (10.111.255.201), port
5432 failed: Connection refused
	Is the server running on that host and accepting TCP/IP connections?
pod "pg-client-22" deleted
pod default/pg-client-22 terminated (Error)
Failover in progress... retrying in 5s
INSERT 0 1
pod "pg-client-22" deleted
INSERT 0 1
pod "pg-client-23" deleted
```

As soon as pg-worker-2 is up and running and the PostgreSQL database accepts the new inserts, pg-client-22 will create the record and the script will continue to pg-client-23. See below for the records after the deletion of pod pg-worker-1:

```
Shell
         msg
---------------------
 Success on record 1
 Success on record 2
 Success on record 3
 Success on record 4
 Success on record 5
 Success on record 6
 Success on record 7
 Success on record 8
 Success on record 9
 Success on record 10
 Success on record 11
 Success on record 12
 Success on record 13
 Success on record 14
 Success on record 15
 Success on record 16
 Success on record 17
 Success on record 18
 Success on record 19
 Success on record 20
 Success on record 21
 Success on record 22
 Success on record 23
 Success on record 24
 Success on record 25
 Success on record 26
 Success on record 27
 Success on record 28
 Success on record 29
 Success on record 30
(30 rows)
```

As you can see, not a single transaction was lost.

Observations:

- **pg-worker-1** stops the database, flushes buffers, and unmounts the volume and will be deleted.

- **pg-worker-2** transitions to the lead member, detects the existing ext4 signature, mounts the volume instantly, starts the PostgreSQL database, and the transaction from the pod pg-client-22 resumes and is accepted.

4.3. Performance and Impact Summary

FEATURE	LEGACY RWO APPROACH	LIGHTBITS RWX APPROACH
Volume Attachment	Serial (Detach then Attach)	Concurrent (Always Attached)
Failover Latency	High (Minutes)	Ultra-Low (Seconds)
Data Integrity	Risk of "Multi-Mount" corruption	Guaranteed via Leader Election Lock
Storage Protocol	Standard iSCSI/Fibre Channel	High-performance NVMe/TCP

5. Conclusion

The implementation of an Active/Passive HA environment utilizing Lightbits RWX storage effectively solves the persistent challenge of storage-induced downtime during Kubernetes failover. By leveraging the NVMe/TCP protocol in conjunction with RWX access modes, this architecture eliminates the "detach-and-reattach" latencies that typically plague standard block storage. The result is a deterministic failover process in which the storage remains concurrently connected to all participant nodes, enabling near-instantaneous workload migration and significantly reduced RTO.

The integration of a robust Leader Election mechanism via Kubernetes RBAC and ConfigMaps ensures a fail-safe orchestration layer that prevents "split-brain" scenarios and data corruption. Our validation testing confirms that the transition from a suspended to an active state is handled with surgical precision; the system performs essential memory-to-disk flushes and file system synchronizations before any handover. This approach guarantees that the passive node remains in a "warm" state, ready to mount the existing ext4 file system immediately upon assuming leadership, maintaining absolute data integrity throughout the process.

Ultimately, this technical framework provides enterprise-grade resilience for stateful PostgreSQL workloads without the overhead of complex, traditional clustering software. By harmonizing Lightbits' high-performance software-defined storage with native Kubernetes orchestration, organizations can deploy a highly available database substrate that is both performant and reliable. This solution not only simplifies disaster recovery workflows but also ensures that mission-critical data remains consistently accessible, even amid node-level failures, maximizing uptime across the entire application stack.

About Lightbits Labs

Lightbits Labs® (Lightbits) invented the NVMe over TCP protocol and offers best-in-class software-defined block storage that enables modernization of data center infrastructure for organizations building private or public clouds. Built from the ground up for low consistent latency, scalability, resiliency, and cost-efficiency, Lightbits software delivers the best price/performance for real-time analytics, transactional, and AI/ML workloads. Lightbits Labs is backed by enterprise technology leaders [Cisco Investments, Dell Technologies Capital, Intel Capital, Lenovo, and Micron] and is on a mission to deliver the fastest and most cost-efficient data storage for performance-sensitive workloads at scale.

www.lightbitslabs.com

info@lightbitslabs.com

US Offices
1830 The Alameda,
San Jose, CA 95126,
USA

Israel Office
17 Atir Yeda Street,
Kfar Saba 4464313,
Israel

LBWP23/2026/3