Redefining HA for Kubernetes: Lightning-Fast Pod Failover with Lightbits RWX

Implementation Guide for HA Kubernetes in Active/Suspend mode

January 2026

Abstract

This white paper details a robust, scalable architecture that leverages the ReadWriteMany (RWX) capabilities of the Lightbits disaggregated, software-defined storage platform to establish a resilient Active/Passive High Availability (HA) framework for Kubernetes. By utilizing Lightbits' native support for clustered NVMe® over TCP (NVMe/TCP) storage, this solution enables seamless volume failover and persistent data access across multiple Kubernetes nodes. The integration focuses on eliminating single points of failure at the storage layer, ensuring that mission-critical stateful applications can automatically recover on standby nodes without data loss or manual intervention. The paper outlines the implementation of Lightbits RWX volumes, the configuration of Kubernetes pod anti-affinity rules for HA orchestration, and the best practices for achieving rapid failover recovery times. This combined solution demonstrates how to achieve enterprise-grade reliability and performance for containerized workloads by leveraging Lightbits' shared-storage efficiency within a dynamically orchestrated Kubernetes environment.

Table of Contents

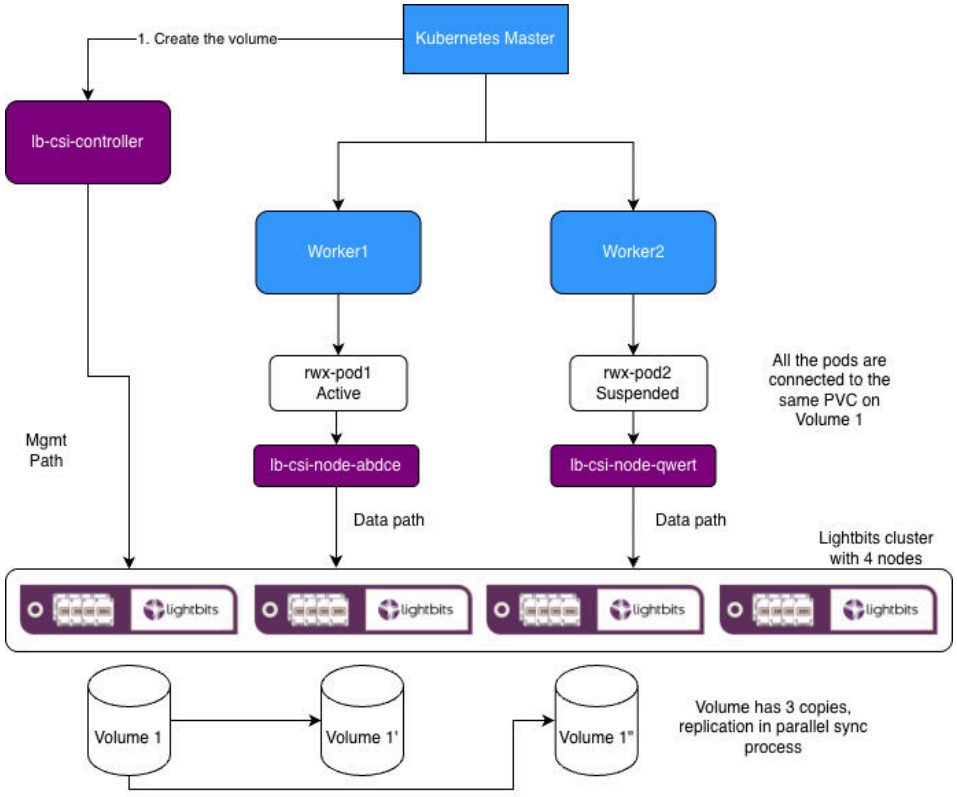
1. Introduction

This white paper provides a comprehensive overview and technical guidelines for implementing a High Availability (HA) Active/Passive environment within Kubernetes. By leveraging the ReadWriteMany (RWX) capabilities of the Lightbits Cloud-Native Storage Interface (CSI), organizations can achieve seamless failover with zero downtime from a storage connectivity perspective.

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 interface with 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. ReadWriteMany (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. Persistent Volume Claim (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 they see the same data set in real time.

1.3. Implementation Workflow

1. StorageClass Configuration: Define a StorageClass that specifies the Lightbits provisioner and sets the protocol to NVMe/TCP.
2. PVC Creation: Provision a PVC with accessModes: [ReadWriteMany].
3. Pod Deployment: * Deploy the Active Pod on Worker 1.
4. Deploy the Passive Pod on Worker 2.
5. Both pods mount the volume at the same target path.
6. Failover Logic: Use a liveness probe or an external orchestrator to manage which pod is processing traffic. If the Active Pod or Worker 1 fails, the Passive Pod on Worker 2—which already has a live connection to the storage—immediately assumes the workload.

Benefits of this Approach

Feature	Technical Impact
Zero Storage Latency	No waiting for volume detachment from a failed node.
Data Consistency	Shared block storage ensures 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 XFS 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 XFS 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 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"]
    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. Storage Definition (Persistent Volume Claim)

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

```shell
# 2. STORAGE (Block Mode RWX)
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: rwx-pvc
spec:
  storageClassName: "example-sc"
  accessModes: ["ReadWriteMany"]
  volumeMode: Block
  resources:
    requests:
      storage: 20Gi
---
```

3.3. Workload Orchestration (Worker 1 & Worker 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 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 XFS signature.
    - If no filesystem exists, it initializes the XFS volume.
    - It mounts the XFS filesystem to the designated mount point.
4. Graceful Termination/Suspension: When a leader yields their position:
    - The system executes a memory-to-disk flush (fsync).
    - A full synchronization is performed to ensure data integrity.
    - The XFS volume is unmounted, though the block device remains connected to the PVC for immediate takeover.

Configuration for rwx-pod1:

Shell

```
# 3. POD FOR WORKER 1
apiVersion: v1
kind: Pod
metadata:
  name: "rwx-pod1"
  labels:
    app: rwx-pods
spec:
  serviceAccountName: leader-election-sa
  nodeName: worker1
  containers:
  - name: workload
    image: docker.io/library/almalinux:latest
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: true
    command: ["/bin/bash", "-c"]
    args:
      - |
        LOCK_CM="block-device-lock"
        DEV="/dev/lbcsiblkdev"
        MNT="/mnt/data"
        mkdir -p $MNT
        NS=$(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/configmaps"

        while true; do
          curl -s -k -X POST "$K8S_API" -H "Authorization: Bearer $TOKEN" -H
"Content-Type: application/json" -d
"{\"metadata\":{\"name\":\"$LOCK_CM\"},\"data\":{\"leader\":\"$HOSTNAME\"}}" >
/dev/null
          RESPONSE=$(curl -s -k -X GET "$K8S_API/$LOCK_CM" -H "Authorization:
Bearer $TOKEN")
          CURRENT_LEADER=$(echo "$RESPONSE" | python3 -c "import sys, json;
print(json.load(sys.stdin).get('data', {}).get('leader', ''))" 2>/dev/null)

          if [ "$CURRENT_LEADER" = "$HOSTNAME" ]; then
            echo "$(date): [ACTIVE] I am the leader ($HOSTNAME)."
```

```
            if [ -b "$DEV" ]; then
              # Force kernel to recognize the device properly
              blockdev --flushbufs $DEV
              blockdev --rereadpt $DEV || true

              # SMART CHECK: Check for XFS signature
              if ! blkid $DEV | grep -q "TYPE=\"xfs\""; then
                echo "$(date): No XFS signature found on $DEV. Formatting
now..."
                mkfs.xfs -f $DEV
              else
                echo "$(date): Valid XFS signature detected. Skipping format."
              fi

              # MOUNT
              if ! mountpoint -q $MNT; then
                echo "$(date): Attempting mount..."
                mount -t xfs -o nouuid,wsync $DEV $MNT && echo "Mount Success!"
              fi

              # PERSISTENCE
              echo "Heartbeat from $HOSTNAME at $(date)" >> $MNT/heartbeat.txt
              sync $MNT
              xfs_freeze -f $MNT && xfs_freeze -u $MNT
            fi
            sleep 10
          else
            echo "$(date): [PASSIVE] Leader is $CURRENT_LEADER."
            if mountpoint -q $MNT; then
              sync $MNT
              umount $MNT || umount -l $MNT
            fi
            sleep 5
          fi
        done
    volumeDevices:
      - name: lb-csi-mount
        devicePath: /dev/lbcsiblkdev
  volumes:
    - name: lb-csi-mount
      persistentVolumeClaim:
        claimName: rwx-pvc
---
```

Configuration for rwx-pod2:

Shell

```
# 4. POD FOR WORKER 2
apiVersion: v1
kind: Pod
metadata:
  name: "rwx-pod2"
  labels:
    app: rwx-pods
spec:
  serviceAccountName: leader-election-sa
  nodeName: worker2
  containers:
  - name: workload
    image: almalinux:latest
    imagePullPolicy: IfNotPresent
    securityContext:
      privileged: true
    command: ["/bin/bash", "-c"]
    args:
      - |
        LOCK_CM="block-device-lock"
        DEV="/dev/lbcsiblkdev"
        MNT="/mnt/data"
        mkdir -p $MNT
        NS=$(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/configmaps"

        while true; do
          curl -s -k -X POST "$K8S_API" -H "Authorization: Bearer $TOKEN" -H
"Content-Type: application/json" -d
"{\"metadata\":{\"name\":\"$LOCK_CM\"},\"data\":{\"leader\":\"$HOSTNAME\"}}" >
/dev/null
          RESPONSE=$(curl -s -k -X GET "$K8S_API/$LOCK_CM" -H "Authorization:
Bearer $TOKEN")
          CURRENT_LEADER=$(echo "$RESPONSE" | python3 -c "import sys, json;
print(json.load(sys.stdin).get('data', {}).get('leader', ''))" 2>/dev/null)

          if [ "$CURRENT_LEADER" = "$HOSTNAME" ]; then
            echo "$(date): [ACTIVE] I am the leader."
```

```
            if [ -b "$DEV" ]; then
              blockdev --flushbufs $DEV
              blockdev --rereadpt $DEV || true
              if ! blkid $DEV | grep -q "TYPE=\"xfs\""; then
                mkfs.xfs -f $DEV
              fi
              if ! mountpoint -q $MNT; then
                mount -t xfs -o nouuid,wsync $DEV $MNT && echo "Mount Success!"
              fi
              echo "Heartbeat from $HOSTNAME at $(date)" >> $MNT/heartbeat.txt
              sync $MNT
              xfs_freeze -f $MNT && xfs_freeze -u $MNT
            fi
            sleep 10
          else
            echo "$(date): [PASSIVE] Leader is $CURRENT_LEADER."
            if mountpoint -q $MNT; then
              sync $MNT
              umount $MNT || umount -l $MNT
            fi
            sleep 5
          fi
        done
    volumeDevices:
      - name: lb-csi-mount
        devicePath: /dev/lbcsiblkdev
  volumes:
    - name: lb-csi-mount
      persistentVolumeClaim:
        claimName: rwx-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.

4.1. Provisioning the Pods and RWX Storage

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

```
Shell
kubectl create -f ActiveSuspend.yaml
serviceaccount/leader-election-sa created
role.rbac.authorization.k8s.io/leader-election-role created
rolebinding.rbac.authorization.k8s.io/leader-election-rolebinding created
persistentvolumeclaim/rwx-pvc created
pod/rwx-pod1 created
pod/rwx-pod2 created
```

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

```
Shell
kubectl get pod -o wide
NAME       READY   STATUS    RESTARTS   AGE   IP            NODE      NOMINATED NODE   READINESS GATES
rwx-pod1   1/1     Running   0          49s   10.244.2.77   worker1   <none>           <none>
rwx-pod2   1/1     Running   0          49s   10.244.1.59   worker2   <none>           <none>
```

Leader Initialization

Upon startup, the Pods compete for the leader lock. In this instance, rwx-pod1 acquires leadership, detects the unformatted Lightbits block device, initializes it with XFS, and mounts the file system.

Logs from rwx-pod1 (Active):

```
Shell
kubectl logs rwx-pod1
Thu Jan  8 08:58:29 UTC 2026: [ACTIVE] I am the leader (rwx-pod1).
Thu Jan  8 08:58:29 UTC 2026: No XFS signature found on /dev/lbcsiblkdev.
Formatting now...
meta-data=/dev/lbcsiblkdev       isize=512    agcount=4, agsize=1310720 blks
         =                       sectsz=4096  attr=2, projid32bit=1
         =                       crc=1        finobt=1, sparse=1, rmapbt=0
         =                       reflink=1    bigtime=1 inobtcount=1 nrext64=0
data     =                       bsize=4096   blocks=5242880, imaxpct=25
         =                       sunit=0      swidth=0 blks
naming   =version 2              bsize=4096   ascii-ci=0, ftype=1
log      =internal log           bsize=4096   blocks=16384, version=2
         =                       sectsz=4096  sunit=1 blks, lazy-count=1
realtime =none                   extsz=4096   blocks=0, rtextents=0
Discarding blocks...Done.
Thu Jan  8 08:58:29 UTC 2026: Attempting mount...
Mount Success!
```

Logs from rwx-pod2 (Suspended):

```
Shell
kubectl logs rwx-pod2
Thu Jan  8 08:58:27 UTC 2026: [PASSIVE] Leader is rwx-pod1.
```

At this stage, both worker nodes are physically connected to the Lightbits PVC via NVMe/TCP, but only the Active node has the XFS mount point engaged. As shown in the screenshot below from Photon, our management UI:

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: Data Creation on Active Node

We access rwx-pod1 to create a test directory and a persistent file:

```
Shell
kubectl exec -it rwx-pod1 -- /bin/bash
cd /mnt/data
[root@rwx-pod1 data]# ls
heartbeat.txt
[root@rwx-pod1 data]# mkdir Test
[root@rwx-pod1 data]# cd Test
[root@rwx-pod1 Test]# vi failover.text
[root@rwx-pod1 Test]# cat failover.text
This line is created from pod: rwx-pod1.
[root@rwx-pod1 Test]# exit
```

Step 2: Executing the Failover

We simulate a failover by manually patching the ConfigMap to transfer leadership to rwx-pod2.

```
Shell
kubectl patch configmap block-device-lock --type merge -p
'{"data":{"leader":"rwx-pod2"}}'
```

Output:

```
Shell
configmap/block-device-lock patched
```

Observation:

- **rwx-pod1** transitions to [PASSIVE], flushes buffers, and unmounts the volume.
- **rwx-pod2** transitions to [ACTIVE], detects the existing XFS signature, and mounts the volume instantly.

Step 3: Verifying Data on New Leader

We verify that rwx-pod2 has inherited the state from the previous leader:

Bash

```
Shell
kubectl exec -it rwx-pod2 -- cat /mnt/data/Test/failover.text
# Output: This line is created from pod: rwx-pod1.
```

Step 4: Failback and Consistency Check

Finally, we add data on rwx-pod2 and perform a failback to rwx-pod1.

Bash

```
Shell
# Add data on pod2
echo "This line is created from pod: rwx-pod2." >> /mnt/data/Test/failover.text

# Failback to pod1
kubectl patch configmap block-device-lock --type merge -p
'{"data":{"leader":"rwx-pod1"}}'
```

Upon checking rwx-pod1, the file contains the full history of edits from both pods, confirming that the Lightbits RWX capability ensures zero data loss and seamless storage connectivity during node transitions.

5. Conclusion

Implementing an Active/Passive High Availability environment with Lightbits RWX storage successfully addresses the critical challenge of storage persistence during Kubernetes failover events. By using the NVMe/TCP storage protocol and the ReadWriteMany access mode, we have demonstrated that storage connectivity can be maintained across multiple worker nodes simultaneously. This architectural choice eliminates the traditional "detach-and-attach" latencies associated with standard block storage, providing a foundation for near-instantaneous workload transitions.

The integration of a Leader Election mechanism via Kubernetes RBAC and ConfigMaps provides a sophisticated layer of orchestration that prevents data corruption while ensuring continuous availability. As evidenced by our failover and failback testing, the transition between the Active and Suspended states is managed gracefully—incorporating essential data integrity operations such as memory-to-disk flushes and file system synchronization. This ensures that the passive node is always "warm" and ready to assume the primary role with a consistent view of the XFS file system.

Ultimately, this solution provides enterprise-grade resilience for stateful applications without the complexity of traditional clustering software. By combining the high-throughput of Lightbits' disaggregated storage with native Kubernetes orchestration, organizations can achieve a zero-downtime storage architecture. This approach not only simplifies disaster recovery workflows but also maximizes infrastructure utilization, ensuring that mission-critical data remains accessible and intact even in the event of individual node failures.

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

LBWP20/2026/1