TECHNICAL WHITE PAPER: October 2026



Scality RING on vSphere Kubernetes Service on VMware Cloud Foundation

Reference Architecture

Table of Contents

Executive Summary 3
vSphere Kubernetes Service 3
Scality RING 4
Container Object Storage Interface and the Scality COSI driver 4
Solution Architecture 5
Kubernetes Environment 5
VKS Cluster Configuration 5
Software Requirements 6
Solution Validation 7
Install the Scality COSI driver on VKS Cluster 7
Step 1 — Switch to the VKS cluster context 7
Step 2 — Install the COSI CRDs and controller 7
Step 3 — Set the Pod Security profile of the driver namespace 8
Step 4 — Install the Scality COSI driver 8
Step 5 — Create the RING connection Secret 8
Step 6 — Define the bucket and access classes 9
Step 7 — Check the installation 10
Validate the Scality COSI driver on VKS Cluster 10
Step 1 — Create the application namespace 10
Step 2 — Request a bucket 10
Step 3 — Request credentials 11
Step 4 — Read and write objects from an application pod 11
Step 5 — Verify isolation, revocation, and deletion 13
Conclusion 16
Appendix A: Example Client VM Configuration & Tooling 17
Resources 17

Executive Summary

Modern applications on Kubernetes store more and more of their data as objects: backups, logs, media, analytics data sets and AI training corpora. Teams provisioning that storage by hand, with tickets and shared credentials, slow down delivery and spread long-lived keys across clusters. Platform teams need object storage to be requested, granted and revoked the same way as the rest of the platform: declaratively, per application, and under the cluster's own access controls.

This paper describes how Scality RING provides object storage to vSphere Kubernetes Service (VKS) through the Kubernetes Container Object Storage Interface (COSI). With the Scality COSI driver deployed in a VKS cluster, an application team creates a bucket and its access credentials with two Kubernetes objects. The driver provisions them on RING and delivers the credentials as a Secret in the application's namespace.

Key advantages of Scality RING with COSI on VKS:

Self-service Buckets: application teams request buckets with a BucketClaim, with no storage tickets and no hand-off to a storage administrator.

Per-application Credentials: each BucketAccess receives its own identity and keys, scoped to its bucket. Deleting the access revokes the keys.

Namespace Isolation: credentials issued in one namespace cannot read or write another namespace's bucket.

Lifecycle as Policy: BucketClasses decide whether a bucket is deleted or retained when its claim is removed.

Standard Kubernetes Tooling: installed with Helm and managed with kubectl on VKS, alongside the cluster's other workloads.

Lower TCO: With VKS, organizations can reduce silos, leverage existing tools and skill sets without having to retrain staff and/or change existing processes, utilizing unified lifecycle management across infrastructure components to stay up to date with the most recent patches and minimizing security risks.

Operational Simplicity: VKS is engineered for unparalleled operational simplicity, leveraging the familiarity of existing vSphere tools, skills, and workflows. This design philosophy significantly reduces the learning curve for IT teams and streamlines management processes. With VKS, organizations benefit from automated cluster provisioning, which accelerates deployment times and minimizes manual configuration errors. Furthermore, its robust capabilities extend to automated upgrades and comprehensive lifecycle management. This integrated approach ensures consistent operations, reduces overhead, and frees up valuable resources to focus on innovation rather than infrastructure maintenance.

Run and Manage Kubernetes at Scale: Effortlessly deploy and manage Kubernetes clusters at scale, leveraging a built-in, Cloud Native Computing Foundation (CNCF) certified Kubernetes distribution. VKS provides fully automated lifecycle management, streamlining operations from initial setup to ongoing maintenance and upgrades. This comprehensive approach ensures that organizations can harness the power of Kubernetes for their containerized applications with unparalleled efficiency and reliability, without the complexities typically associated with large-scale Kubernetes deployments.

vSphere Kubernetes Service

vSphere Kubernetes Service (VKS) is the Kubernetes runtime built directly into VMware Cloud Foundation (VCF). With CNCF certified Kubernetes, VKS enables platform engineers to deploy and manage Kubernetes clusters while leveraging a comprehensive set of cloud services in VCF. Cloud admins benefit from the support for N-2 Kubernetes versions, enterprise grade security, and simplified lifecycle management for modern apps adoption.

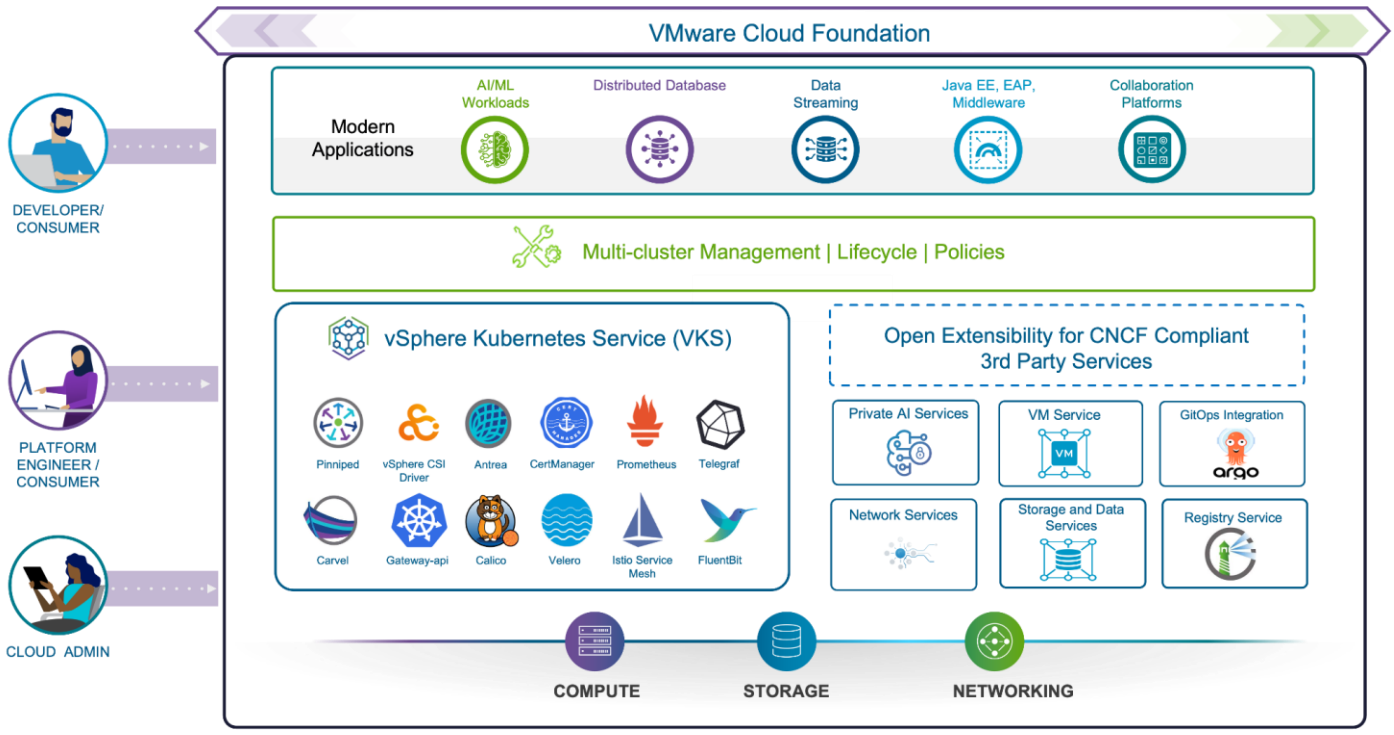


Figure 1: VKS on VCF Ecosystem

Scality RING

Scality RING is software-defined, scale-out object and file storage that runs on standard x86 servers. It exposes an S3-compatible API with an IAM service on the same endpoint, so applications built for S3 use it without code changes. RING scales from petabytes upward, and protects data with replication and erasure coding across nodes and sites.

In this architecture, RING runs outside the VKS cluster, either in the same data center or reachable over the network, and serves as the object store for VKS workloads.

Container Object Storage Interface and the Scality COSI driver

The Container Object Storage Interface (COSI) is the Kubernetes standard for provisioning object storage, the object-storage counterpart of CSI for block and file volumes. COSI adds five Kubernetes resources and a controller:

- **BucketClass** and **BucketAccessClass**: cluster-wide templates set by the platform team (which driver, which deletion policy, which credentials the driver uses).
- **BucketClaim**: a request for a bucket, created by an application team in its namespace.
- **BucketAccess**: a request for credentials to that bucket, delivered as a Kubernetes Secret.
- **Bucket**: the cluster's record of the provisioned bucket.

The Scality COSI driver implements COSI for RING. For each BucketClaim it creates a bucket on RING. For each BucketAccess it creates a dedicated IAM user, scoped to that bucket, with an access key, and writes the connection details into the application's Secret. Deleting the BucketAccess deletes the user and its keys.

The Scality RING workflow on Kubernetes follows four stages illustrated in Figure 2:

1. The platform team installs COSI and the Scality driver once and defines BucketClasses and a BucketAccessClass.
2. An application team creates a BucketClaim; the driver creates the bucket on RING.
3. The team creates a BucketAccess; the driver creates a scoped IAM user and keys, and the credentials land in a Secret in the namespace, under the key BucketInfo.
4. The application pod mounts the Secret and reads and writes RING directly over S3.

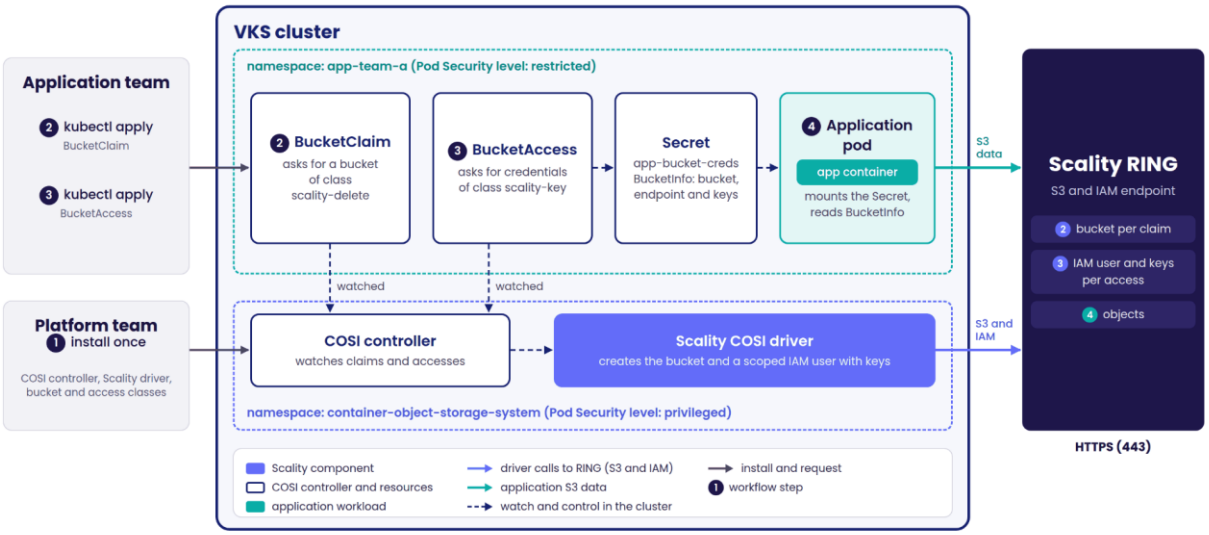


Figure 2: Scality COSI driver workflow. The numbers match the workflow steps above.

Solution Architecture

Kubernetes Environment

The COSI controller and the Scality COSI driver run in one namespace of a VKS cluster. Applications in other namespaces request buckets and credentials through COSI resources. Object data flows directly between application pods and Scality RING over HTTPS.

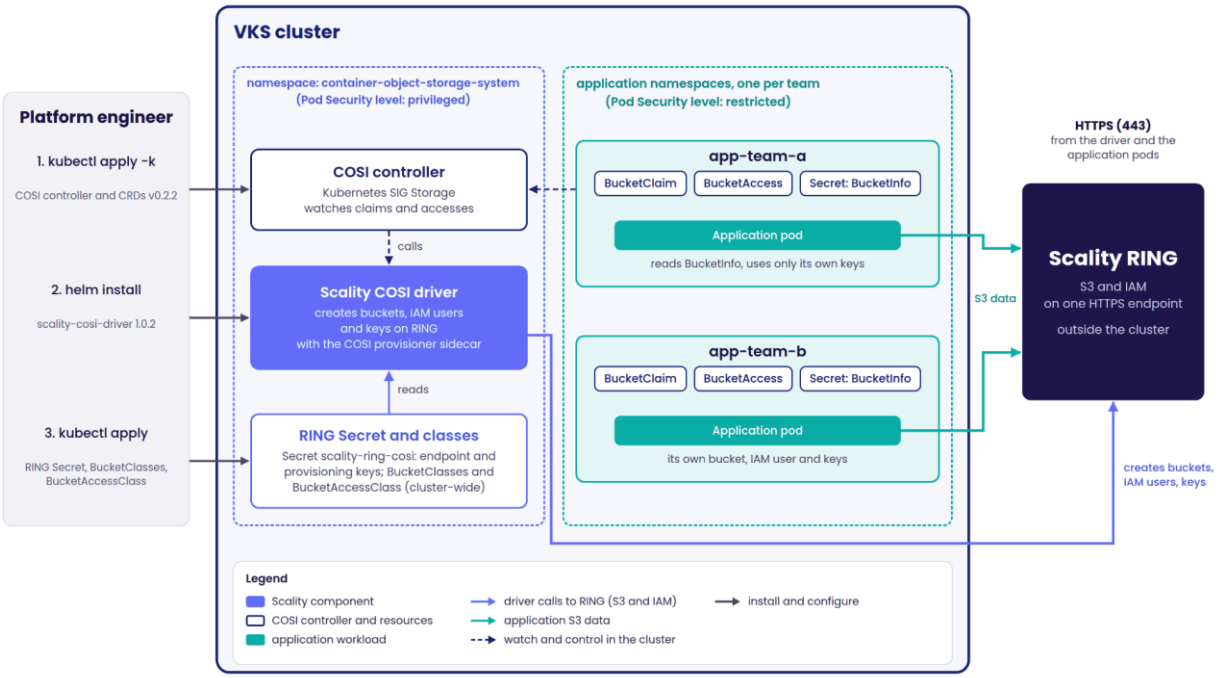


Figure 3: Scality RING with the COSI driver on VKS

VKS Cluster Configuration

The cluster comprised one control plane node and three worker nodes running Kubernetes v1.36.2+vmware.2 on Ubuntu 24.04. Storage for the cluster was provided by vSAN Express Storage Architecture (ESA), using the default RAID 5 storage policy, which the vSphere CSI driver exposes as the storage class vsan-esa-default-policy-raid5. The COSI driver needs no persistent volumes; object data is stored on Scality RING.

The cluster was managed from a client VM (Appendix A).

Table 1: Kubernetes Cluster Configuration Summary

Component	Configuration	Notes
Kubernetes Control Plane	1 replica; 2x vCPU and 8GB RAM	VM Class: best-effort-medium; Storage Class: vsan-esa-default-policy-raid5
Kubernetes Worker Nodes	3 replicas; 4x vCPU and 16GB RAM	VM Class: best-effort-large; Storage Class: vsan-esa-default-policy-raid5
Kubernetes Release	v1.36.2+vmware.2	Container runtime containerd 2.3.2+vmware.3-fips
OS Image	Ubuntu 24.04.4 LTS	Kernel version 6.17.0-35-generic
Object storage	Scality RING 9.5.3	S3 and IAM on one HTTPS endpoint, outside the cluster

```
holuser@console:~$ kubectl version
Client Version: v1.33.3
Kustomize Version: v5.6.0
Server Version: v1.36.2+vmware.2
WARNING: version difference between client (1.33) and server (1.36) exceeds the supported minor version skew of +/-1
holuser@console:~$ kubectl get nodes -o wide
NAME                                  STATUS   ROLES           AGE     VERSION            INTERNAL-IP    EXTERNAL-IP   OS-IMAGE             KERNEL-VERSION              CONTAINER-RUNTIME
scality-c1-jh95n-vskfv                Ready    control-plane   5d17h   v1.36.2+vmware.2   172.30.0.2     <none>        Ubuntu 24.04.4 LTS   6.17.0-35-generic (amd64)   containerd://2.3.2
+vmware.3-fips
scality-c1-node-pool-1-bhnbj-4nhks-52lt8   Ready    <none>          5d17h   v1.36.2+vmware.2   172.30.0.5     <none>        Ubuntu 24.04.4 LTS   6.17.0-35-generic (amd64)   containerd://2.3.2
+vmware.3-fips
scality-c1-node-pool-1-bhnbj-4nhks-79v6k   Ready    <none>          5d17h   v1.36.2+vmware.2   172.30.0.3     <none>        Ubuntu 24.04.4 LTS   6.17.0-35-generic (amd64)   containerd://2.3.2
+vmware.3-fips
scality-c1-node-pool-1-bhnbj-4nhks-dtn9b   Ready    <none>          5d17h   v1.36.2+vmware.2   172.30.0.4     <none>        Ubuntu 24.04.4 LTS   6.17.0-35-generic (amd64)   containerd://2.3.2
+vmware.3-fips
holuser@console:~$
```

Figure 4: The validated VKS cluster, one control plane and three worker nodes

Software Requirements

Listed below is a set of software components required to successfully deploy Scality RING on VKS clusters. The software versions are current as of the validation testing and may not reflect the latest update.

For this deployment, the COSI controller was installed from the Kubernetes project's manifests, and the Scality COSI driver was installed with Helm. No VKS add-on is required.

Table 2: Kubernetes Cluster Packages

Package	Version	Notes
VKS	3.7.2	
Container Object Storage Interface (COSI) controller and CRDs	v0.2.2 (API objectstorage.k8s.io/v1alpha1)	Kubernetes SIG Storage; kustomize manifests
Scality COSI driver	1.0.2	Helm chart oci://ghcr.io/scality/cosi-driver/helm-charts/scality-cosi-driver; includes the COSI provisioner sidecar

Container images pulled by the cluster:

Table 3: Container images

Image	Registry
ghcr.io/scality/cosi-driver:1.0.2	GitHub Container Registry
gcr.io/k8s-staging-sig-storage/objectstorage-sidecar:v20241219-v0.1.0-60-g6a5a12c	Google Container Registry (Kubernetes staging)
gcr.io/k8s-staging-sig-storage/objectstorage-controller:v20250905-controllerv0.2.0-rc1-100-gd904c62	Google Container Registry (Kubernetes staging)

For clusters without internet access, mirror these images to Harbor. The driver image is a Helm value (image.repository). The sidecar image is set in the chart's Deployment template, so pointing it at Harbor needs a Helm post-renderer or a patched chart. The controller image is set in its kustomize manifest. Air-gapped installation was not part of this validation.

Every manifest and Helm command needed to reproduce this architecture is given in full in the Solution Validation section. They are also available in the accompanying repository:

https://github.com/vmware/vks-validated-solutions/tree/main/Scality

Solution Validation

Install the Scality COSI driver on VKS Cluster

Prerequisites:

- VKS cluster already provisioned, with a working kubectl context (see Appendix A for VMware's VCF CLI documentation), validated on Kubernetes v1.36.2+vmware.2 (see the driver documentation for the minimum version), with cluster-admin access.
- kubectl and Helm 3.8 or later on the client VM (Appendix A), plus git for the COSI manifests.
- Scality RING 9.4.2 or later (validated on 9.5.3), with an account whose keys can create buckets and IAM users.
- Connectivity: the client VM reaches github.com, ghcr.io and gcr.io; the cluster nodes reach ghcr.io, gcr.io and the RING S3 endpoint on port 443.

Step 1 — Switch to the VKS cluster context

```
# Create a VCF CLI context for the VKS cluster (once)
vcf context create <cluster-context> \
  --endpoint <supervisor-address> \
  --username <user@domain> \
  --workload-cluster-namespace=<vsphere-namespace> \
  --workload-cluster-name=<vks-cluster-name>

# Switch to the VKS cluster and check its nodes
vcf context use <cluster-context>:<vks-cluster-name>
kubectl get nodes
```

vcf context create signs in to the Supervisor as the given user and creates a CLI context for the VKS cluster in its vSphere namespace. *vcf context use <cluster-context>:<vks-cluster-name>* points kubectl at that cluster, so every command that follows targets it, and *kubectl get nodes* confirms that all nodes are ready.

If kubectl later reports an authentication error, the login has expired: run the following command to refresh context:

vcf context refresh --insecure-skip-tls-verify

The account behind the context needs cluster-admin rights, because the next steps create CRDs and cluster-wide RBAC.

Step 2 — Install the COSI CRDs and controller

The Scality COSI driver 1.0.2 uses the COSI API *v1alpha1*. Install the matching controller release, v0.2.2; the upstream main branch carries *v1alpha2* and is not compatible.

```
# Install the COSI CRDs and controller
# (creates the namespace container-object-storage-system)
kubectl apply -k \
  "github.com/kubernetes-sigs/container-object-storage-interface//?ref=v0.2.2"

kubectl -n container-object-storage-system wait deploy --all \
  --for=condition=Available --timeout=300s
```

What these commands do:

- *kubectl apply -k "github.com/…?ref=v0.2.2"* builds the COSI manifests with kustomize straight from the Kubernetes project's repository, pinned to release v0.2.2. It creates the namespace container-object-storage-system, the five COSI CRDs (BucketClass, BucketClaim, Bucket, BucketAccessClass, BucketAccess), the controller's RBAC and the controller deployment.
- The COSI controller watches BucketClaims and BucketAccesses in every namespace and turns them into calls to the storage driver installed in Step 4.
- *kubectl wait --for=condition=Available* blocks until the controller is running (at most 5 minutes), so the next steps do not start before it is ready.

Step 3 — Set the Pod Security profile of the driver namespace

The driver pod runs without a restricted security context, so its namespace needs a Pod Security level above restricted on clusters that enforce one. The validation used *privileged*.

```
kubectl label ns --overwrite container-object-storage-system \
  pod-security.kubernetes.io/enforce=privileged
```

The label sets the Pod Security Admission level enforced in this namespace only. Without it, on a cluster that enforces baseline or restricted by default, the API server would reject the driver pod at creation. Application namespaces are not affected and can stay restricted.

Step 4 — Install the Scality COSI driver

```
helm upgrade --install scality-cosi-driver \
  oci://ghcr.io/scality/cosi-driver/helm-charts/scality-cosi-driver --version 1.0.2 \
  --namespace container-object-storage-system –wait
```

helm upgrade --install installs the release, or upgrades it if it already exists, from Scality's OCI Helm chart, pinned to version 1.0.2. The chart creates the driver Deployment, its ServiceAccount and RBAC. Each driver pod runs two containers: the Scality COSI driver, which talks to RING, and the COSI provisioner sidecar, which connects the driver to the COSI controller. *--wait* returns once the pod is ready.

Step 5 — Create the RING connection Secret

The driver reads the RING endpoint and the keys it uses for provisioning from a Secret. On RING, S3 and IAM share one endpoint. Add *tlsCert* (PEM) if the endpoint uses a private CA.

```
kubectl apply -f - <<'EOF'
apiVersion: v1
kind: Secret
metadata:
  name: scality-ring-cosi
  namespace: container-object-storage-system
type: Opaque
stringData:
  accessKeyId: <ACCESS_KEY_ID>
  secretAccessKey: <SECRET_ACCESS_KEY>
  endpoint: https://s3.example.com
  iamEndpoint: https://s3.example.com
```

```
  region: us-east-1
EOF
```

The Secret holds the RING account the driver acts as:

- *accessKeyId* and *secretAccessKey*: keys allowed to create and delete buckets and IAM users. The driver uses them for provisioning only; applications never see them.
- *endpoint*: the RING S3 URL that buckets are created on.
- *iamEndpoint*: the RING IAM URL used to create one user per BucketAccess. On RING it is the same URL as endpoint.
- *region*: the region name sent in S3 and IAM requests.

Keep this Secret in the driver namespace and restrict who can read it: it carries the provisioning account's keys.

Step 6 — Define the bucket and access classes

The platform team defines the classes once. The class name becomes the prefix of each bucket name. *deletionPolicy* decides what happens to a bucket when its claim is deleted.

```
kubectl apply -f - <<'EOF'
kind: BucketClass
apiVersion: objectstorage.k8s.io/v1alpha1
metadata:
  name: scality-delete
driverName: cosi.scality.com
deletionPolicy: Delete
parameters:
  objectStorageSecretName: scality-ring-cosi
  objectStorageSecretNamespace: container-object-storage-system
---
kind: BucketClass
apiVersion: objectstorage.k8s.io/v1alpha1
metadata:
  name: scality-retain
driverName: cosi.scality.com
deletionPolicy: Retain
parameters:
  objectStorageSecretName: scality-ring-cosi
  objectStorageSecretNamespace: container-object-storage-system
---
kind: BucketAccessClass
apiVersion: objectstorage.k8s.io/v1alpha1
metadata:
  name: scality-key
driverName: cosi.scality.com
authenticationType: KEY
parameters:
  objectStorageSecretName: scality-ring-cosi
  objectStorageSecretNamespace: container-object-storage-system
EOF
```

What the classes define:

- A **BucketClass** is the template for new buckets: which driver serves them (cosi.scality.com), what happens to the bucket when its claim is deleted (*deletionPolicy*), and which Secret the driver uses to create it. Two classes let application teams choose between disposable (*scality-delete*) and retained (*scality-retain*) buckets.
- A **BucketAccessClass** is the template for credentials. *authenticationType*: KEY means each access gets an S3 access key and secret key.
- Both are cluster-scoped and created once by the platform team; application teams only reference them by name.

Step 7 — Check the installation

```
kubectl get pods -n container-object-storage-system

kubectl get crd | grep objectstorage.k8s.io
```

The driver pod runs two containers, the Scality driver and the COSI provisioner sidecar.

```
holuser@console:~$ kubectl get pods -n container-object-storage-system
NAME                                                READY   STATUS    RESTARTS   AGE
container-object-storage-controller-7c455d55bf-ppmv5   1/1     Running   0          72s
scality-cosi-driver-5876b7bc45-fpqqc                2/2     Running   0          65s
holuser@console:~$ kubectl get crd | grep objectstorage.k8s.io
bucketaccessclasses.objectstorage.k8s.io                    2026-10-20T08:01:17Z
bucketaccesses.objectstorage.k8s.io                         2026-10-20T08:01:17Z
bucketclaims.objectstorage.k8s.io                           2026-10-20T08:01:17Z
bucketclasses.objectstorage.k8s.io                          2026-10-20T08:01:17Z
buckets.objectstorage.k8s.io                                2026-10-20T08:01:17Z
holuser@console:~$
```

Figure 5: COSI controller, Scality COSI driver and COSI CRDs on the VKS cluster

Validate the Scality COSI driver on VKS Cluster

The validation plays the part of an application team. It requests a bucket and credentials in its own namespace, then writes and reads objects using only those credentials.

Step 1 — Create the application namespace

```
kubectl create namespace app-team-a
```

The namespace is the application team's boundary: the bucket, the credentials Secret and the pods that use them all live in it, and the credentials issued here work only on this team's bucket.

Step 2 — Request a bucket

```
kubectl apply -f - <<'EOF'
kind: BucketClaim
apiVersion: objectstorage.k8s.io/v1alpha1
metadata:
  name: app-bucket
  namespace: app-team-a
spec:
  bucketClassName: scality-delete
  protocols: [s3]
EOF
```

```
kubectl -n app-team-a wait bucketclaim/app-bucket \
  --for=jsonpath='{.status.bucketReady}'=true --timeout=300s
```

The BucketClaim asks for one bucket of class scality-delete. The COSI controller creates a cluster-level Bucket object for it, and the driver creates the bucket on RING, named after the class, for example *scality-delete<uuid>. kubectl wait … bucketReady=true* returns once the bucket exists.

Step 3 — Request credentials

```
kubectl apply -f - <<'EOF'
kind: BucketAccess
apiVersion: objectstorage.k8s.io/v1alpha1
metadata:
  name: app-access
  namespace: app-team-a
spec:
  bucketClaimName: app-bucket
  bucketAccessClassName: scality-key
  credentialsSecretName: app-bucket-creds
  protocol: s3
EOF

kubectl -n app-team-a wait bucketaccess/app-access \
  --for=jsonpath='{.status.accessGranted}'=true --timeout=300s
kubectl get bucketclaim,bucketaccess -A

NAMESPACE    NAME                                        AGE
app-team-a   bucketclaim.objectstorage.k8s.io/app-bucket   30s

NAMESPACE    NAME                                         AGE
app-team-a   bucketaccess.objectstorage.k8s.io/app-access   17s
```

The driver has created an IAM user for this access on RING, with a policy limited to the bucket, and the Secret *app-bucket-creds* now holds a *BucketInfo* document with the bucket name, endpoint, region and keys.

Step 4 — Read and write objects from an application pod

The pod mounts the Secret and reads its connection details from *BucketInfo*. It never sees the keys the driver uses. The container runs as a non-root user and meets the Kubernetes *restricted* Pod Security profile. Path-style addressing is set for the S3 client, which works with any RING endpoint name.

```
kubectl -n app-team-a apply -f - <<'EOF'
apiVersion: v1
kind: Pod
metadata:
  name: app-a
spec:
  restartPolicy: Never
  securityContext:
    runAsNonRoot: true
```

```
    runAsUser: 1000
    seccompProfile: {type: RuntimeDefault}
  containers:
    - name: app
      image: public.ecr.aws/aws-cli/aws-cli:2.17.0
      env:
        - {name: HOME, value: /tmp}
      command: ["/bin/bash", "-c"]
      args:
        - |
          set -euo pipefail
          # Read the connection details COSI wrote into the Secret
          INFO=$(cat /cosi/BucketInfo)
          get() { echo "$INFO" | grep -o "\"$1\": *\"[^\"]*\"" \
                  | head -1 | sed 's/.*: *"\(.*\)"/\1/'; }
          export AWS_ACCESS_KEY_ID=$(get accessKeyID)
          export AWS_SECRET_ACCESS_KEY=$(get accessSecretKey)
          REGION=$(get region); export AWS_DEFAULT_REGION=${REGION:-us-east-1}
          BUCKET=$(get bucketName); ENDPOINT=$(get endpoint)
          aws configure set default.s3.addressing_style path
          s3() { aws --endpoint-url "$ENDPOINT" s3 "$@"; }
          # Write, list and read back an object
          echo "hello from VKS" > /tmp/hello.txt
          s3 cp /tmp/hello.txt "s3://$BUCKET/cosi-test/hello.txt"
          s3 ls "s3://$BUCKET/cosi-test/"
          s3 cp "s3://$BUCKET/cosi-test/hello.txt" -
      securityContext:
        allowPrivilegeEscalation: false
        capabilities: {drop: [ALL]}
      volumeMounts:
        - {name: cosi, mountPath: /cosi, readOnly: true}
  volumes:
    - name: cosi
      secret: {secretName: app-bucket-creds}
EOF

kubectl -n app-team-a wait pod/app-a \
  --for=jsonpath='{.status.phase}'=Succeeded --timeout=120s
kubectl -n app-team-a logs app-a
```

What the pod does, line by line:

- The Secret *app-bucket-creds* is mounted read-only at */cosi*. *INFO=$(cat /cosi/BucketInfo)* reads the JSON document COSI wrote into it.
- *get()* pulls one field out of that JSON with grep and sed, because the AWS CLI image has no JSON tool. It reads the access key, secret key, region, bucket name and endpoint.
- The keys and region are exported as the standard AWS CLI environment variables. aws configure set *default.s3.addressing_style path* makes the CLI address buckets as *https://<endpoint>/<bucket>*, which works with any RING endpoint name and certificate.
- *s3()* is a short wrapper for *aws --endpoint-url "$ENDPOINT" s3*. *s3 cp* uploads a small file, *s3 ls* lists the folder and *s3 cp … -* prints the object back.
- *kubectl wait … Succeeded* waits for the pod to finish (at most 2 minutes), and *kubectl logs* shows its output.

If the COSI components are set up correctly, the log looks similar to:

```
kubectl -n app-team-a logs app-a
upload: ../tmp/hello.txt to s3://scality-delete<uuid>/cosi-test/hello.txt
2026-10-20 08:06:35         15 hello.txt
hello from VKS
```

Step 5 — Verify isolation, revocation, and deletion

Repeat Steps 1 to 3 in a second namespace, *app-team-b*, with BucketClaim *app-bucket-b*, BucketAccess *app-access-b* and Secret *app-bucket-b-creds*. Then check the following.

Isolation. Credentials from *app-team-b* must not reach the bucket of *app-team-a*. Look up the name of team A's bucket on RING, then try to list it with team B's credentials:

```
BUCKET_A=$(kubectl get bucket \
  $(kubectl -n app-team-a get bucketclaim app-bucket \
      -o jsonpath='{.status.bucketName}') \
  -o jsonpath='{.status.bucketID}')

kubectl -n app-team-b apply -f - <<EOF
apiVersion: v1
kind: Pod
metadata:
  name: isolation-check
spec:
  restartPolicy: Never
  securityContext:
    runAsNonRoot: true
    runAsUser: 1000
    seccompProfile: {type: RuntimeDefault}
  containers:
    - name: check
      image: public.ecr.aws/aws-cli/aws-cli:2.17.0
      env:
        - {name: HOME, value: /tmp}
        - {name: OTHER_BUCKET, value: "${BUCKET_A}"}
      command: ["/bin/bash", "-c"]
      args:
        - |
          INFO=\$(cat /cosi/BucketInfo)
          get() { echo "\$INFO" | grep -o "\"\$1\": *\"[^\"]*\"" \\
                  | head -1 | sed 's/.*: *"\(.*\)"/\1/'; }
          export AWS_ACCESS_KEY_ID=\$(get accessKeyID)
          export AWS_SECRET_ACCESS_KEY=\$(get accessSecretKey)
          export AWS_DEFAULT_REGION=us-east-1
          aws configure set default.s3.addressing_style path
          if aws --endpoint-url "\$(get endpoint)" s3 ls "s3://\$OTHER_BUCKET/"
          then echo "FAIL: access allowed"
          else echo "OK: access denied"
          fi
```

```
      securityContext:
        allowPrivilegeEscalation: false
        capabilities: {drop: [ALL]}
      volumeMounts: [{name: cosi, mountPath: /cosi, readOnly: true}]
  volumes: [{name: cosi, secret: {secretName: app-bucket-b-creds}}]
EOF

kubectl -n app-team-b wait pod/isolation-check \
  --for=jsonpath='{.status.phase}'=Succeeded --timeout=120s
kubectl -n app-team-b logs isolation-check | tail -1   # OK: access denied
```

What these commands do:

- *BUCKET_A=…* finds the real name of team A's bucket on RING: the BucketClaim points to a cluster-level Bucket object, whose *status.bucketID* is the bucket's name on RING.
- The pod manifest is passed through an unquoted heredoc, so the shell fills in *${BUCKET_A}*. The escaped *\$* keep the pod's own variables intact for the script inside the pod.
- The pod mounts team B's Secret *app-bucket-b-creds* and tries to list team A's bucket with those keys. RING answers *AccessDenied*, and the script prints *OK: access denied. FAIL: access allowed* would mean the isolation does not hold.

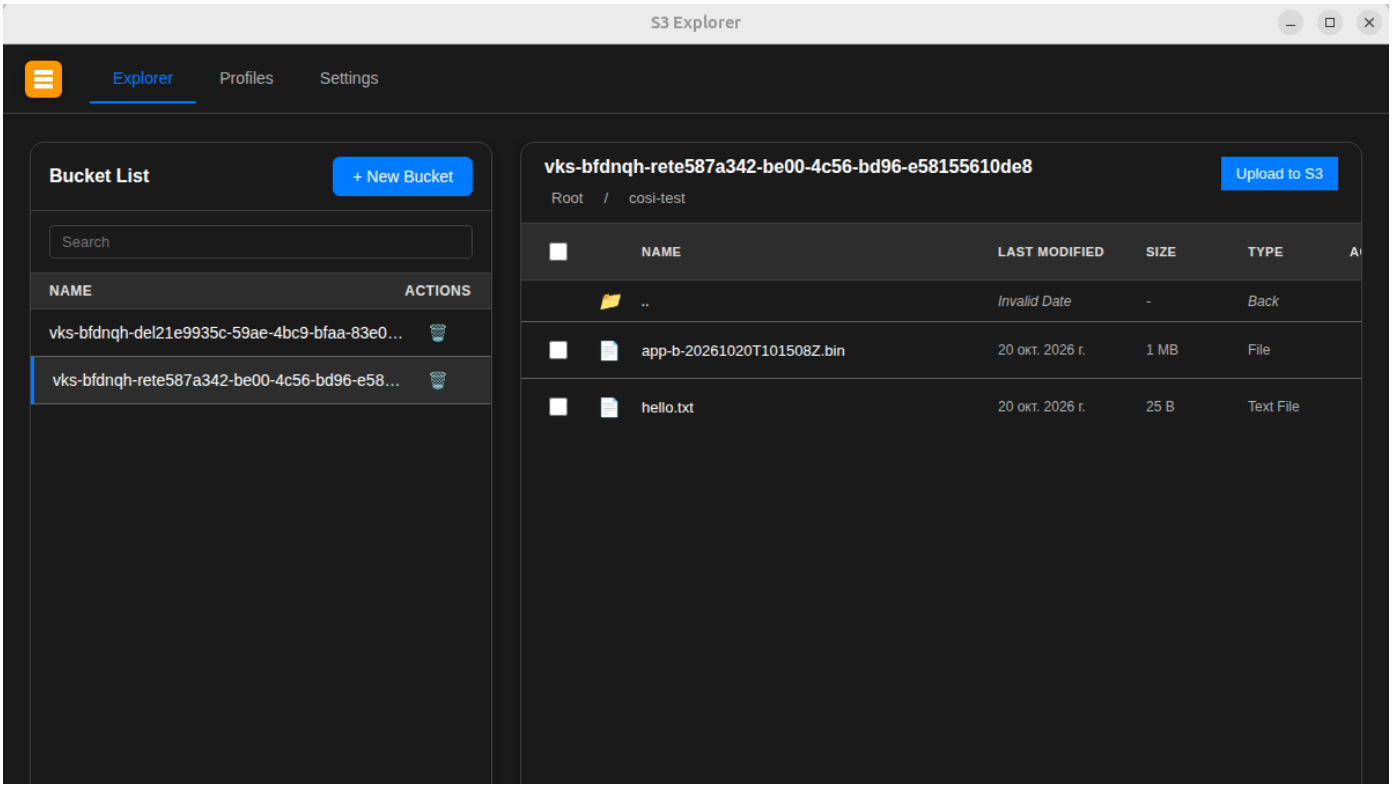


Figure 6: Each application namespace receives its own bucket on RING

Revocation. Deleting a BucketAccess makes the driver delete its IAM user and keys on RING, and the Secret is removed:

```
kubectl -n app-team-b delete bucketaccess app-access-b
kubectl -n app-team-b get secret app-bucket-b-creds   # NotFound
```

kubectl delete bucketaccess asks the driver to delete the IAM user created for this access and its keys on RING. The COSI controller then deletes the Secret, which *kubectl get secret* confirms with *NotFound*.

Any copy of the old keys is rejected by RING from then on.

Deletion policy. A claim of class *scality-delete* removes its bucket when the claim is deleted, provided the bucket is empty. A claim of class *scality-retain* keeps the bucket on RING, so it can be re-imported or removed by the storage administrator.

```
# Remove the objects first (for example from the app-a pod), then:
kubectl -n app-team-a delete bucketaccess app-access
kubectl -n app-team-a delete bucketclaim app-bucket
kubectl get bucket   # the Bucket record is gone
```

The BucketAccess is deleted first, so the driver removes the IAM user while the bucket still exists. Deleting the BucketClaim then makes the driver delete the bucket on RING, because its class uses *deletionPolicy: Delete*. *kubectl get bucket* confirms that the cluster-level Bucket record is gone. The driver does not delete a bucket that still holds objects.

Results of these checks in the validated environment:

Table 4: List of verification test results

Check	How	Result
Namespace isolation	A second namespace claims its own bucket; its credentials try to LIST and PUT on the first namespace's bucket	Both denied
Credential revocation	Delete the BucketAccess, then retry with a saved copy of its keys	LIST and PUT rejected
Delete policy	Empty the bucket, then delete a claim of class scality-delete	Bucket removed from RING
Retain policy	Delete a claim of class scality-retain	Bucket kept on RING

To clean up, delete the COSI objects before the namespaces. Their finalizers can only be cleared by the COSI controller and driver, so a namespace deleted first can stay in *Terminating*:

```
# Clean up the example, one namespace at a time
for ns in app-team-a app-team-b; do
  kubectl -n $ns delete pod --all           # stop the applications
  kubectl -n $ns delete bucketaccess --all   # removes IAM users and keys
  kubectl -n $ns delete bucketclaim --all    # removes Delete-policy buckets
  kubectl delete namespace $ns
done
```

Conclusion

This paper demonstrated Scality RING as the object storage of vSphere Kubernetes Service through the Kubernetes Container Object Storage Interface. With the COSI controller and the Scality COSI driver installed by Helm in a single namespace, application teams on VKS request buckets and credentials with standard Kubernetes resources. Each application gets its own bucket and its own scoped identity on RING, delivered as a Secret in its namespace.

The validation on VKS confirmed the full lifecycle: bucket provisioning, per-application credentials, read and write from an unprivileged pod, isolation between namespaces, credential revocation, and the Delete and Retain policies. No VKS-specific configuration was needed beyond the Pod Security level of the driver namespace.

Together, VKS and Scality RING give platform teams a self-service, policy-driven path to object storage for modern applications. Teams keep the familiar vSphere operating model for the clusters, and S3 storage that scales independently of them.

Appendix A: Example Client VM Configuration & Tooling

To provision a VKS cluster and obtain a kubectl context for it, follow VMware's VCF CLI documentation:

- Workflow for Provisioning VKS Clusters Using VCF CLI
- Connect to a VKS Cluster with VCF CLI
- VCF CLI Context, Architecture, and Configuration

This paper assumes a VKS cluster has already been provisioned and that the client VM has a working kubectl context for it, with cluster-admin rights. The validation was run from a Linux client VM with these tools:

Tool	Version used	Purpose
kubectl	v1.33.3	Cluster operations; a client within one minor version of the cluster is recommended
Helm	v3.16.4	Install the Scality COSI driver
Git	any	Fetch the COSI manifests with kubectl apply -k

Resources

- Scality COSI driver and its Helm installation guide
- Kubernetes Container Object Storage Interface
- Scality RING
- VMware vSphere Kubernetes Service
- vSphere Supervisor services documentation



Copyright © 2026 Broadcom. All rights reserved.
The term "Broadcom" refers to Broadcom Inc. and/or its subsidiaries. For more information, go to www.broadcom.com. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies. Broadcom reserves the right to make changes without further notice to any products or data herein to improve reliability, function, or design. Information furnished by Broadcom is believed to be accurate and reliable. However, Broadcom does not assume any liability arising out of the application or use of this information, nor the application or use of any product or circuit described herein, neither does it convey any license under its patent rights nor the rights of others.
Item No: vmw-bc-wp-tech-uslet-word-2026; Oct-26