TECHNICAL WHITE PAPER:
July 2026

Migrating Workloads Between VKS Clusters

A Professional Services Guide for Migrating Stateful and Stateless Workloads Between VKS Clusters

Contents

Executive Summary

This paper is one of four in the VKS migration series. Use this paper if your source is itself a VKS cluster on a different Supervisor or vCenter. For a source that is not running vSphere CSI, see Migrating Non-vSphere Kubernetes Workloads to VKS Using Filesystem Replication. For a source already on vSphere CSI but not itself VKS, see Migrating vSphere-Based Kubernetes Workloads to VKS Using Zero-Copy FCD Adoption for the equivalent zero-copy path. For the fully validated, benchmarked Velero+S3 reference methodology, see Migrating Kubernetes Workloads to VKS Using Velero and S3-Compatible Storage.

Migrating stateful workloads between VKS clusters becomes more complex when the clusters are managed by different Supervisors or vCenters. However, the application data ultimately resides on a vSphere First Class Disk (FCD). By safely dismantling and reconstructing the relationship between the VKS PV, Supervisor PVC/PV and FCD, the existing volume can be moved and adopted without an additional application- or file-level data copy.

This paper presents a controlled migration method that separates application metadata from storage movement. Velero transfers the Kubernetes manifests, while cross-vCenter vMotion moves the FCD using a temporary helper VM. The destination Supervisor then registers the disk through CnsRegisterVolume and presents it to the destination VKS cluster through a statically bound PV and PVC. The approach includes explicit controls for workload quiescence, storage retention, policy compatibility, validation and rollback.

Introduction

vSphere Kubernetes Service

VMware Cloud Foundation with vSphere Kubernetes Service (VKS) provides an enterprise-grade 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.

Key benefits include:

- **Built-in Governance and Security**: VKS as part of VMware Cloud Foundation, provides centralized policy enforcement, role-based access control, and network micro-segmentation through NSX, ensuring Kubernetes workloads meet enterprise security standards.
- **Consistent Infrastructure Management**: VKS leverages the same vSphere infrastructure that enterprises already trust, eliminating the operational complexity of managing separate Kubernetes and VM stacks.
- **Unified Networking and Storage**: VKS inherits vSphere's mature networking and storage capabilities, simplifying connectivity between cloud-native applications and existing enterprise systems.
- **Operational Familiarity**: IT teams can manage Kubernetes clusters using familiar vSphere tools and workflows, reducing the learning curve and operational risk associated with adopting container platforms.

By building on VMware Cloud Foundation, organizations establish a standardized, supportable Kubernetes foundation that bridges cloud-native development practices with enterprise operational requirements.

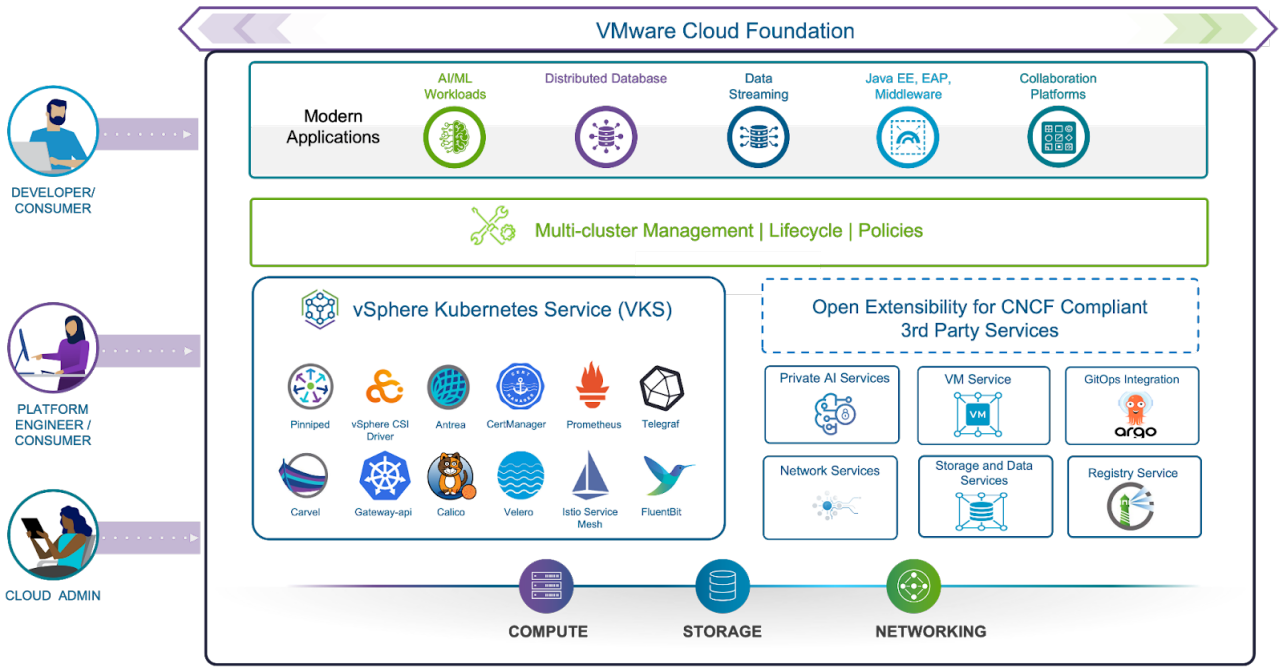


Figure 1: VMware Cloud Foundation with vSphere Kubernetes Service

The example manifests, Helm values, and supporting configuration referenced throughout this paper are available in the accompanying repository: https://github.com/vmware/vks-validated-solutions/tree/main/VKS-Migrations

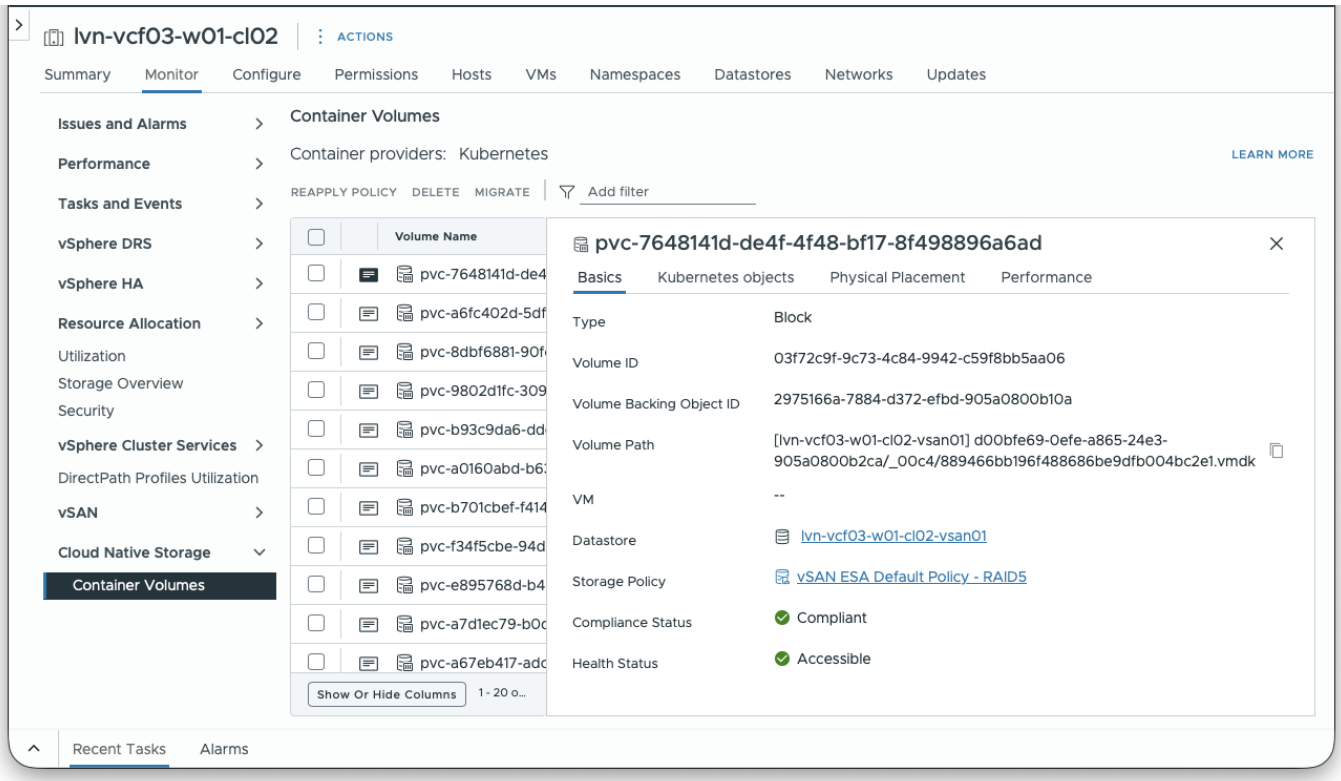
Solution Architecture

vSphere FCD, CSI Driver and Persistent Storage

The CSI driver handles storage requests from VKS, effectively translating requests for persistent volumes into vSphere virtual-disk provisioning. These virtual disks are provisioned as 'First Class Disks' in vSphere.

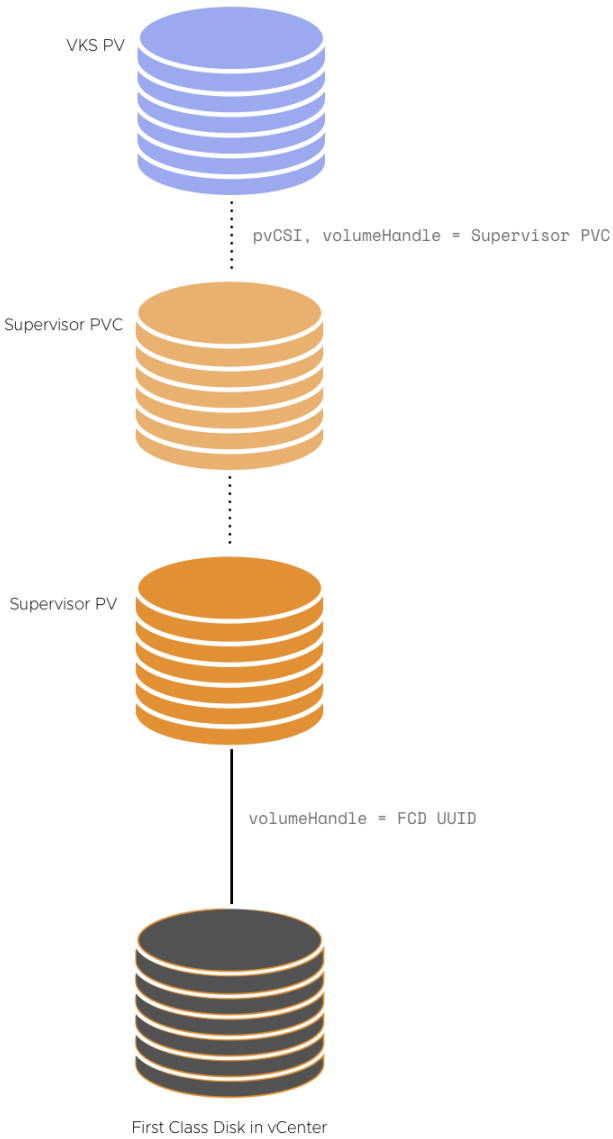
A First Class Disk, or FCD, is a vSphere-managed virtual disk with an identity and lifecycle independent of any particular Virtual Machine and unlike a conventional Virtual Machine disk, each FCD is a standalone storage object rather than as a disk permanently owned by a Virtual Machine. Note that our here scope is specific to Read Write Once (RWO) volumes.

Once the CSI driver provisions the storage, it will return a `volumeHandle` identifier for each disk, and used for subsequent storage operations (see `CSIPersistentVolumeSource` in the Kubernetes PersistentVolume API spec at https://kubernetes.io/docs/reference/kubernetes-api/core/persistent-volume-v1 and https://github.com/kubernetes-sigs/vsphere-csi-driver for the CSI driver).



VKS and Para-Virtual CSI

VKS clusters use a modified path to connect to the underlying storage, using a para-virtualized version of the CSI driver. Here, each PV in VKS points to a PVC in the Supervisor cluster (within the same Supervisor namespace as the VKS cluster); that Supervisor PVC then binds to a Supervisor PV whose volumeHandle identifies the actual FCD.



For more details visit https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vcf-consumption/latest/managing-vsphere-kuberenetes-service-clusters-and-workloads/deploying-workloads-on-tkg-service-clusters/storage-concepts-for-tkg-service-clusters.html

FCD Migration and Registration

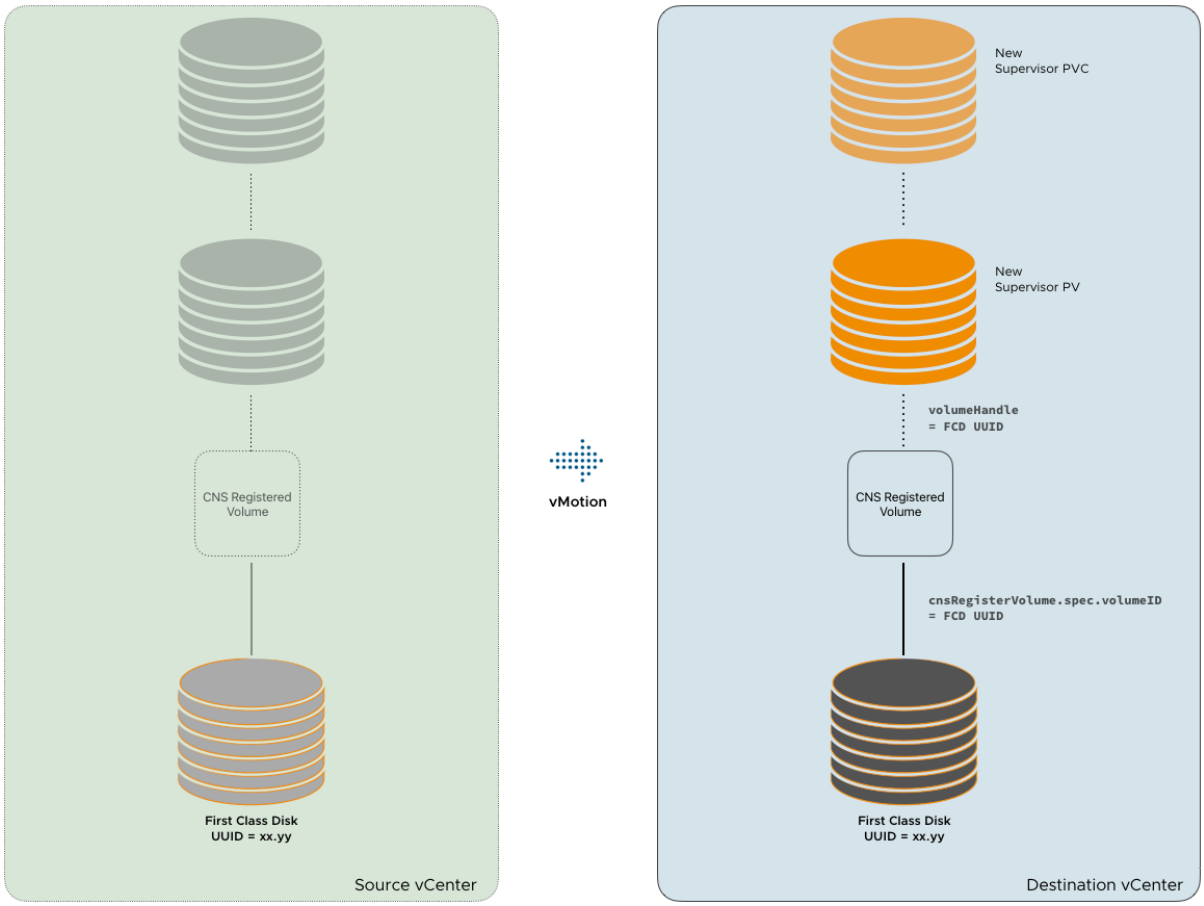
Just like any other virtual disk, FCDs can be migrated between clusters, datastores, etc. If this migration is within the same vCenter (i.e. where the FCD was created) the situation is trivial and the virtual disk can be accessed without any registration changes, as the disk is known and *registered* with the vCenter CNS database. Furthermore, if the migration is within the same datastore no actual data copy is required.

However, if the FCD is migrated across a vCenter boundary, the destination vCenter will need information about the disk in order for the destination CSI driver to address it. This can be achieved by leveraging the `CnsRegisterVolume` Custom Resource (CR).

The simple manifest below shows an example of invoking the `CnsRegisterVolume` CR to register an FCD:

```
apiVersion: cns.vmware.com/v1alpha1
kind: CnsRegisterVolume
metadata:
  name: register-application-data
  namespace: destination-supervisor-namespace
spec:
  pvcName: application-data-adopted
  volumeID: 2c5999e4-e4a7-4cf8-9220-ac9f2d04f4b1
  accessMode: ReadWriteOnce
```

The `volumeID` here is the FCD UUID, which is retained across the disk migration path. The `pvcName` is used to create new destination Supervisor PV/PVC objects (the Supervisor PV `volumeHandle` matches the `volumeID` of the registered volume/FCD UUID, and it addresses the FCD directly).



The resulting Supervisor PVC can then be referenced from a statically created VKS PV:

```
spec:
  csi:
    driver: csi.vsphere.vmware.com
    volumeHandle: application-data-adopted
```

Source Object Retention

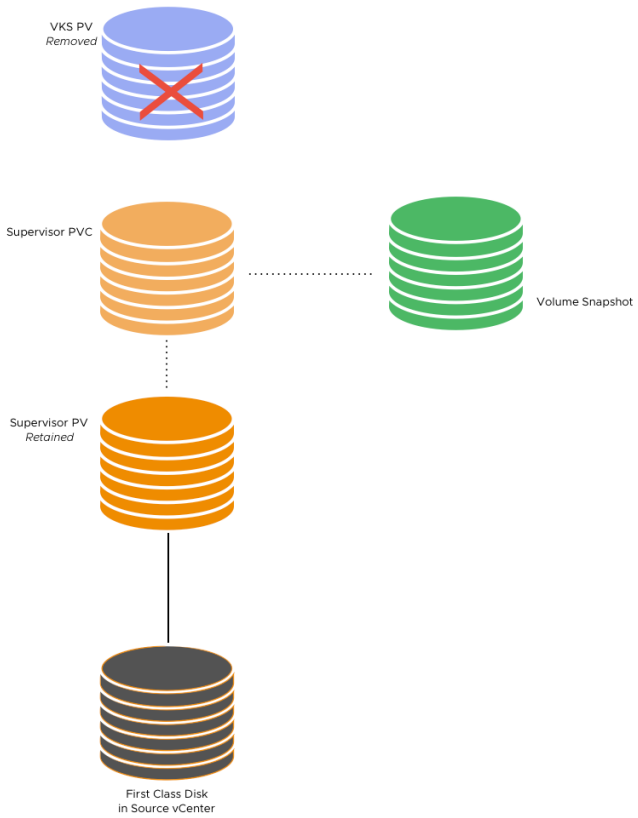
Prior to migrating the FCD, the VKS Workload on the source vCenter will need to stop writes to the disk. After this, the VKS PVC will need to be removed; on vCenter this effectively unmounts the disk from the specific VKS node (this is required before we can vMotion the FCD).

When the source VKS PV is removed, the Supervisor objects on the source will automatically be reconciled and thus deleted too. To stop this, the canonical method is to patch the source Supervisor PV to 'Retain':

```
kubectl patch pv "$SRC_SUP_PV" \
  --type=merge \
  -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'
```

Alternatively, if the migration is either within the same Supervisor namespace, or if the FCD is migrated elsewhere, we can create a snapshot of the Supervisor PVC before removing the VKS objects. The existence of this snapshot will mean that the Supervisor PVC and PV cannot be deleted (until the snapshot is removed). The only instance where this method cannot be used is within the same Supervisor, but different namespace.

Note that this behavior should first be tested on the source cluster before progressing with the migration.



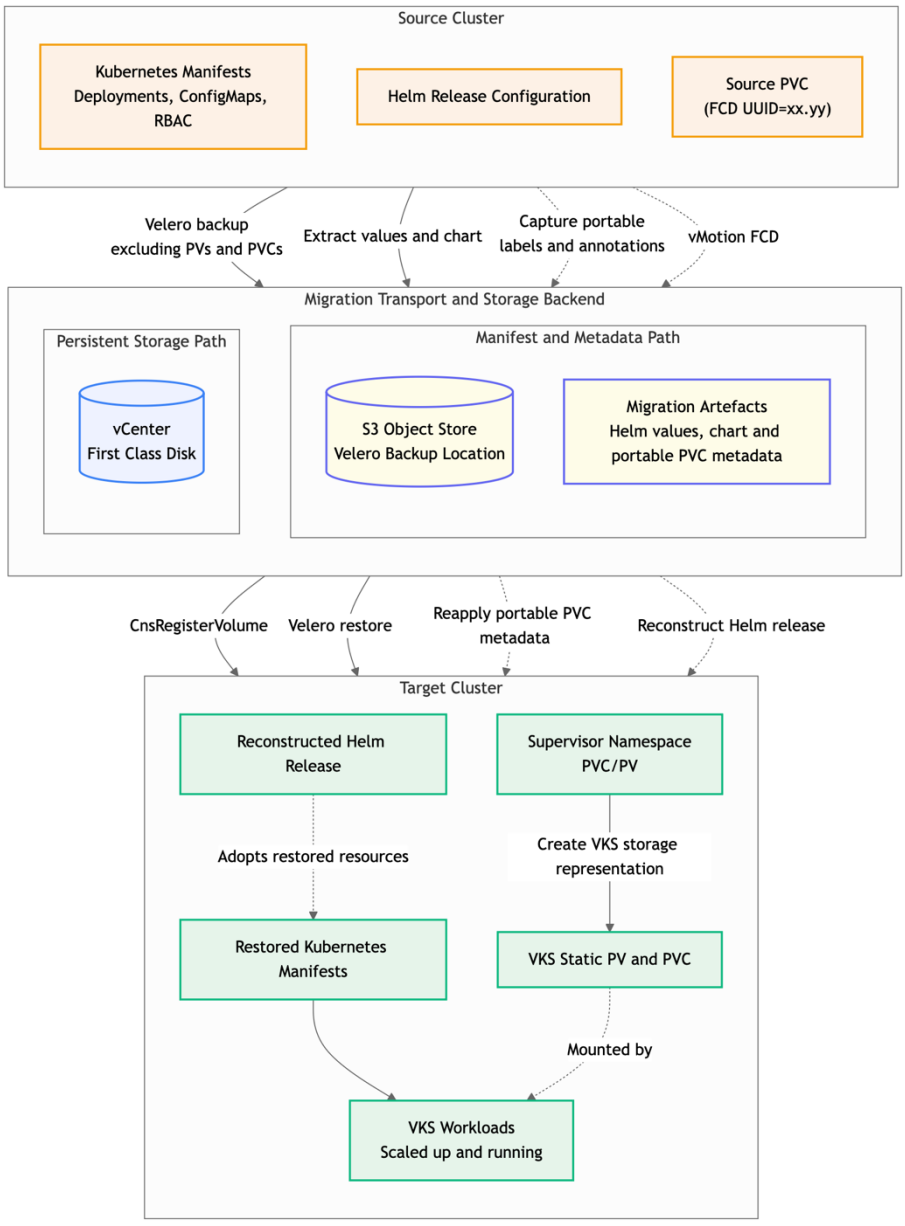
Migration Architecture Overview

Given that we are able to seamlessly migrate and register FCD volumes using vMotion and the Supervisor CR, we can weave this into a sound migration strategy.

However, so far we have only described data movement. To successfully migrate applications, we must also migrate application metadata. Here, we employ the industry standard backup tool Velero to capture details (to an S3 endpoint). Crucially, we instruct Velero not to consider any storage objects (PVC, PV and snapshots), as below (for simplicity, we show only the excluded values):

```
velero backup create "$BACKUP_NAME" \
  --include-cluster-resources=false \
  --snapshot-volumes=false \
  --exclude-resources pv,pvc,volumesnapshots,volumesnapshotcontents
```

We also need to consider how scaling down the source application will affect packaged applications (i.e Helm Charts). The chart information will need to be reconstructed on the destination cluster.



Thus the migration architecture decouples stateful data from stateless configuration, creating two parallel migration paths from the source Kubernetes cluster to the target VKS cluster on VCF 9. Stateless manifests and configurations are transported via Velero to an S3 object store backup location.

Meanwhile, the stateful vSphere FCDs are migrated using vMotion. The `CnsRegisterVolume` API is then utilized on the VCF 9 Supervisor to adopt these existing FCDs directly into the new VKS cluster. If the FCD is migrated within the vCenter boundary, and on the same datastore, there is no data copy needed. Otherwise the persistent data copy is managed by the vMotion process.

Note: because PVs and PVCs are excluded from the Velero restore, application-owned labels and annotations required by Helm, operators or workload controllers must be captured from the source claim and applied to the statically reconstructed destination PVC. Kubernetes-generated binding, provisioning and lifecycle metadata must not be copied.

Source Environment Resource Inventory

The following table captures the complete resource inventory expected from a ten-namespace production cluster discovered using the Phase 1 deep-scan methodology.

Resource Category	Typical Count	Migration Action	Risk
Deployments / StatefulSets	120 / 45	Velero backup → restore	Low
DaemonSets / Jobs / CronJobs	18 / 34 / 12	Velero backup → restore	Low
PVCs — vSphere CSI (FCD)	89	CnsRegisterVolume	Medium
CustomResourceDefinitions	37	Manual reinstall + CR restore	High
Operators / OLM Subscriptions	11	Helm/OLM reinstall	High
RBAC Roles + Bindings	280	Namespace-scoped Velero restore	Medium
Secrets (non-SA)	145	Encrypted Velero / Vault sync	High
Services + Ingress	98 + 23	Platform translation	Medium
NetworkPolicies	67	Velero + CNI validation	Medium
PCI/HIPAA/GDPR namespaces	4	Compliance sign-off required	Critical
Helm Releases	14	Pre-cutover extraction + Target reconstruction	High

Migration Methodology

Overview

The VKS Migration Methodology is a four-phase framework that orchestrates discovery, analysis, blueprinting, and wave-based execution with built-in validation gates and rollback checkpoints.

Phase	Objective	Primary Tooling	Gate Artifact
1: Discovery	Resource inventory, regulatory ID, risk scoring	kubectl, govc, Velero, Python scanner	Discovery Report JSON
2: Analyze	Platform translation, PVC strategy, sizing	Resource Translator, FCD scanner, govc	Risk Matrix + Wave Plan
3: Blueprint	SDF, dependency graphs, approval workflow	SDF Generator, Graphviz	Signed SDF Document
4: Execute	Wave-based migration with dry-runs, rollback	Velero, kubectl, CnsRegisterVolume	Migration Report
5: Monitor	Real-time validation, auto rollback triggers	Prometheus, kubectl, alert rules	Post-migration Report

Tooling

Executing a state-aware migration requires a tightly orchestrated ecosystem of CLI utilities. Rather than relying on a monolithic black-box migration appliance, this methodology utilizes native, heavily audited open-source binaries orchestrated by a strict bash-based state machine.

Tool	Primary Function in Migration
kubectl	Core Kubernetes API interaction. Handles workload quiescence, PVC/PV metadata extraction, API discovery, and live-state validation across both source and destination clusters.
velero	Stateless manifest transport. Executes manifest-only backups (excluding persistent volumes) and handles automated namespace mapping during target restoration.

helm	Application state reconstruction. Discovers source release versions, extracts supplied/computed values, downloads charts locally, and reconstructs the application on the target without conflicting with Velero.
jq / Python (PyYAML)	State management and schema validation. jq safely parses Kubernetes JSON outputs and maintains the immutable migration state file. Python recursively parses rendered Helm YAML to detect cluster-scoped resource violations.
govc (Optional)	Out-of-band vSphere validation. Used to independently verify First Class Disk (FCD) UUIDs directly against the vCenter database prior to CnsRegisterVolume execution.

Phase 1: Discovery

The discovery phase performs four focused scans of the source cluster: Kubernetes resources, network topology, security configuration and vSphere-backed storage.

Note: the kubectl commands in this section use the kubeconfig for the source cluster being analyzed. Ensure that the current context or `KUBECONFIG` environment variable points to the intended cluster.

Create a directory for the discovery artefacts:

```
mkdir -p /tmp/vks-discovery
```

Tier-1: Kubernetes API Inventory

Tier 1 discovers the namespaced resource types exposed by the source Kubernetes API and catalogues the objects that the current identity can list. Custom Resource Definitions are recorded separately to identify platform-specific or application-specific APIs.

```
## TIER 1 — Kubernetes API Inventory

kubectl api-resources \
  --verbs=list \
  --namespaced=true \
  -o name |
  while read -r resource; do
    kubectl get "$resource" --all-namespaces -o json 2>/dev/null
  done |
    jq -s '[.[] | .items[]?]' \
    > /tmp/vks-discovery/all_resources.json

kubectl get customresourcedefinitions -o json |
  jq '[
    .items[] |
    {
      name: .metadata.name,
      group: .spec.group,
      kind: .spec.names.kind,
      scope: .spec.scope,
```

```
      versions: [
        .spec.versions[]
        | select(.served == true)
        | .name
      ]
    }
  ]' > /tmp/vks-discovery/crds.json

✓ Tier 1 complete: 1,847 objects catalogued across 14 namespaces
```

Tier-2: Network Topology

Tier 2 records `Services`, `EndpointSlices`, Ingress resources and `NetworkPolicies`.

```
## TIER 2 — Network Topology

kubectl get svc,endpointslices,ing,netpol -A -o json \
  > /tmp/vks-discovery/network.json

✓ Tier 2 complete: 98 Services, 23 ingress or route resources and 67 NetworkPolicies mapped
```

Tier-3: RBAC and Secrets

Tier 3 records namespaced and cluster-scoped RBAC separately. `ServiceAccounts` are included because `RoleBindings` and workload identities commonly reference them.

Secrets are catalogued by namespace, name and type only. Secret values are not exported during discovery.

```
## TIER 3 — RBAC and Secret Metadata

kubectl get sa,roles,rolebindings -A -o json > /tmp/vks-discovery/rbac_namespaced.json
kubectl get clusterroles,clusterrolebindings -o json > /tmp/vks-discovery/rbac_cluster.json

jq -s '{
  namespaced: .[0].items,
  clusterScoped: .[1].items
}' \
  /tmp/vks-discovery/rbac_namespaced.json \
  /tmp/vks-discovery/rbac_cluster.json \
  > /tmp/vks-discovery/rbac.json

kubectl get secrets --all-namespaces -o json |
  jq '[
    .items[] |
    {
      namespace: .metadata.namespace,
      name: .metadata.name,
      type: .type
    }
  ]' > /tmp/vks-discovery/secrets_catalogue.json

✓ Tier 3 complete: 280 RBAC objects and 145 Secrets catalogued
```

Tier-4: vSphere FCD Cross-Reference

Here we interrogate the Supervisor PVC/PV to confirm the FCD mappings

```
## TIER 4 — vSphere FCD Cross-Reference

alias sup='kubectl --kubeconfig=${SUPERVISOR_KUBECONFIG}'

sup get pvc -A -o json |
```

```
  jq -r '.items[] | [.metadata.namespace,.metadata.name,.spec.volumeName] | @tsv' |
  while IFS=$'\t' read -r ns pvc pv; do
    fcd=$(sup get pv "$pv" -o jsonpath='{.spec.csi.volumeHandle}')
    printf '{"namespace":"%s","pvc":"%s","pv":"%s","fcd":"%s"}\n' \
    "$ns" "$pvc" "$pv" "$fcd"
  done > /tmp/vks-discovery/fcd_mappings_supervisor.jsonl

govc disk.ls -json -q metadataKey.eq=cns.k8s.pvc.namespace |
  jq --slurpfile mappings /tmp/vks-discovery/fcd_mappings_supervisor.jsonl '
    ($mappings | map({(.fcd): .}) | add // {}) as $volumes |
    [
      .objects[]?
      | (.config.id.id // .config.id // empty) as $id
      | select($volumes[$id] != null)
      | $volumes[$id] + {confirmedInVCenter:true}
    ]
  ' > /tmp/vks-discovery/fcd_map.json

✓ Tier 4: 89 of 103 PVCs resolved to vSphere FCDs and confirmed in vCenter
```

> ⚠ **WARNING**
>
> FCD UUID Integrity:
>
> After any FCD vMotion, run the Tier-4 inspection against the destination vCenter and verify FCD UUIDs match the catalogue before proceeding to the *execute* phase.

Phase 2: Analyze

PVC Migration Decision Framework

Below is a comparison between migration within the vCenter boundary and outside of it.

Decision Criterion	Zero-Copy Path	Data-Copy Path
First Class Disk (FCD) in vCenter Database	Yes → Retain Supervisor objects; create new VKS PV	No → Velero + vMotion + CnsRegisterVolume
Compliance tag	Manual sign-off, then zero-copy	Manual sign-off + audit log
Time for 100 GB volume	< 2 minutes	45–90 minutes

IMPORTANT

Regulatory Workloads: Namespaces tagged `pci-scope=true`, `hipaa=true`, or `gdpr=true` MUST receive written compliance officer approval before wave assignment.

Phase 3: Blueprint

Beyond the technical migration process, each migration wave should be governed by a blueprint plan that defines the source inventory, target architecture, wave sequencing, compliance approvals, rollback procedures and validation criteria. This ensures that execution is not treated as a purely technical cutover, but as a controlled change with clear ownership, approval gates and measurable success conditions.

SDF Section	Content	Owner	Gate
Background	Scope, timeline, risk summary	Project Manager	Gate 1: Sponsor
Source Inventory	Full discovery output + risk scores	Solution Architect	Gate 1: Sponsor

Target Architecture	VKS cluster spec, NSX-T, storage classes	Platform Architect	Gate 2: Platform
Wave Plan	Namespace groupings, sequence, windows	Migration Lead	Gate 2: Platform
Compliance Approvals	PCI/HIPAA/GDPR sign-offs	Compliance Officer	Gate 3: Compliance
Rollback Procedures	Snapshot IDs, restore commands, RTOs	Operations Lead	Gate 3: Compliance
Validation Criteria	Health check scripts, success thresholds	QA Lead	Gate 3: Compliance

NOTE
No execution wave may begin until all three gates have recorded written approval in the SDF.

Phase 4: Execute

Execution is performed in controlled migration waves.
Each wave:

1. Quiesces the selected source workloads.
2. Protects the backing FCD through the source Supervisor.
3. Removes the source VKS storage objects.
4. Attaches the FCD to a temporary helper VM.
5. Moves the helper VM and FCD to the destination vCenter using cross-vCenter vMotion.
6. Detaches and registers the migrated FCD with the destination Supervisor.
7. Reconstructs the destination VKS PV and PVC.
8. Restores the application manifests and activates the workloads.

The migration is split into two independent paths:

- **The Kubernetes object path:** Velero migrates namespace-scoped application manifests and metadata.
- **The storage path:** cross-vCenter vMotion transfers the existing FCD to the destination vCenter. CnsRegisterVolume then reconstructs the CNS and Supervisor representation of the volume.

Scope and assumptions

The example below supports the following baseline case:

- Source and destination are VKS clusters managed by different vCenters and Supervisors
- The source volume is a vSphere CSI First Class Disk (FCD)
- The volume is `ReadWriteOnce` and filesystem-based
- The destination Supervisor supports `CnsRegisterVolume`
- A powered-off helper VM can be migrated between the two vCenters
- The destination datastore is accessible to the destination Supervisor
- The required destination storage policy is assigned to the FCD and exposed to the destination Supervisor namespace
- Velero is installed on both source and destination clusters and both installations can access the same backup store
- The application can tolerate an outage covering detach, cross-vCenter storage migration, registration, reconstruction and validation
- Preservation of the FCD UUID across vCenters has been validated with the exact vCenter, ESXi and VCF/VKS versions in use

The destination FCD UUID must always be discovered after vMotion. **It must not be assumed to match the source value without verification.**

For simplicity, the example omits retries, persistent migration state, concurrency controls, automated rollback and finalizer recovery.

Further examples can be referenced from the GitHub repository
https://github.com/vmware/vks-validated-solutions/tree/main/VKS-Migrations/VKS-to-VKS/examples

Stage 0: Environment Setup

Four Kubernetes API contexts are required:

- Source VKS cluster
- Source Supervisor
- Destination Supervisor
- Destination VKS cluster

Access to both vCenters is also required for helper-VM operations and cross-vCenter vMotion.

```
## STAGE 0 — Environment and Tool Setup

export SRC_VKS_KUBECONFIG="$HOME/.kube/source-vks-config"
export SRC_SUP_KUBECONFIG="$HOME/.kube/source-supervisor-config"
export DST_SUP_KUBECONFIG="$HOME/.kube/destination-supervisor-config"
export DST_VKS_KUBECONFIG="$HOME/.kube/destination-vks-config"

export SRC_VKS_NS="prod-payments"
export SRC_SUP_NS="source-vsphere-namespace"
export DST_SUP_NS="migration-target"
export DST_VKS_NS="prod-payments"

export PVC_NAME="payments-db"
export DEPLOYMENT_NAME="payments-api"
export STATEFULSET_NAME="postgres-payments"

export BACKUP_NAME="wave1-prod-payments"
export RESTORE_NAME="wave1-prod-payments"

export DST_SUP_PVC="payments-db-adopted"
export REGISTER_NAME="register-payments-db"
export DST_VKS_PV="payments-db-cross-vcenter"

export MIGRATION_DIR="./${BACKUP_NAME}-artifacts"
mkdir -p "$MIGRATION_DIR"

src_vks() { kubectl --kubeconfig "$SRC_VKS_KUBECONFIG" "$@"; }
src_sup() { kubectl --kubeconfig "$SRC_SUP_KUBECONFIG" "$@"; }
dst_sup() { kubectl --kubeconfig "$DST_SUP_KUBECONFIG" "$@"; }
dst_vks() { kubectl --kubeconfig "$DST_VKS_KUBECONFIG" "$@"; }

src_vks cluster-info
src_sup cluster-info
dst_sup cluster-info
dst_vks cluster-info

✓ Source Kubernetes, destination VKS and destination Supervisor APIs reachable
```

Stage 1: Validate the Velero Backup Store

Velero must already be installed on the source and destination clusters. Both installations must have access to the same S3 endpoint used for the migration:

```
## STAGE 1 — Validate Velero BackupStorageLocation

src -n velero get backupstoragelocations.velero.io
vks -n velero get backupstoragelocations.velero.io

velero backup-location get --kubeconfig "$SRC_KUBECONFIG"
velero backup-location get --kubeconfig "$VKS_KUBECONFIG"

✓ Velero BackupStorageLocation reports available on both clusters
```

Before starting the migration wave, also confirm:

- Cross-vCenter vMotion compatibility checks pass
- The destination datastore has sufficient capacity
- The storage policy is available on the destination Supervisor
- The source and destination vCenters have been tested for FCD UUID preservation
- No operator or GitOps controller will recreate the source PVC during migration

Stage 2: Discover and record the source storage chain

Resolve the source PVC to its PV and record the underlying FCD UUID

```
## STAGE 2 — Discover and record the source storage chain

export SRC_VKS_PV=$(
  src_vks -n "$SRC_VKS_NS" get pvc "$PVC_NAME" \
    -o jsonpath='{.spec.volumeName}'
)

export SRC_SUP_PVC=$(
  src_vks get pv "$SRC_VKS_PV" \
    -o jsonpath='{.spec.csi.volumeHandle}'
)

export SRC_SUP_PV=$(
  src_sup -n "$SRC_SUP_NS" get pvc "$SRC_SUP_PVC" \
    -o jsonpath='{.spec.volumeName}'
)

export SRC_FCD_UUID=$(
  src_sup get pv "$SRC_SUP_PV" \
    -o jsonpath='{.spec.csi.volumeHandle}'
)

export CAPACITY=$(
  src_vks get pv "$SRC_VKS_PV" \
    -o jsonpath='{.spec.capacity.storage}'
)

export FS_TYPE=$(
  src_vks get pv "$SRC_VKS_PV" \
    -o jsonpath='{.spec.csi.fsType}'
)
export FS_TYPE="${FS_TYPE:-ext4}"

printf '%-24s %s\n' \
  "Source VKS PV:" "$SRC_VKS_PV" \
  "Source Supervisor PVC:" "$SRC_SUP_PVC" \
  "Source Supervisor PV:" "$SRC_SUP_PV" \
  "Source FCD UUID:" "$SRC_FCD_UUID" \
  "Capacity:" "$CAPACITY" \
  "Filesystem:" "$FS_TYPE"
```

Record the source objects before making any changes:

```
src_vks -n "$SRC_VKS_NS" get pvc "$PVC_NAME" -o yaml \
  > "$MIGRATION_DIR/source-vks-pvc.yaml"

src_vks get pv "$SRC_VKS_PV" -o yaml \
  > "$MIGRATION_DIR/source-vks-pv.yaml"

src_sup -n "$SRC_SUP_NS" get pvc "$SRC_SUP_PVC" -o yaml \
  > "$MIGRATION_DIR/source-supervisor-pvc.yaml"

src_sup get pv "$SRC_SUP_PV" -o yaml \
  > "$MIGRATION_DIR/source-supervisor-pv.yaml"
```

Capture app-owned PVC metadata before the source PVC is deleted:

```
src_vks -n "$SRC_VKS_NS" get pvc "$PVC_NAME" -o json |
  jq '{
    metadata: {
      labels: (.metadata.labels // {}),
      annotations: (
        (.metadata.annotations // {}) |
        with_entries(
          select(
            (.key | startswith("pv.kubernetes.io/") | not) and
            (.key | startswith("volume.kubernetes.io/") | not) and
            (.key | startswith("volume.beta.kubernetes.io/") | not) and
            (.key != "kubectl.kubernetes.io/last-applied-configuration")
          )
        )
      )
    }
  }' > "$MIGRATION_DIR/portable-pvc-metadata.json"
```

Capture the desired source replica counts

```
export SOURCE_DEPLOY_REPLICAS=$(
  src_vks -n "$SRC_VKS_NS" get deployment "$DEPLOYMENT_NAME" \
    -o jsonpath='{.spec.replicas}'
)

export SOURCE_STS_REPLICAS=$(
  src_vks -n "$SRC_VKS_NS" get statefulset "$STATEFULSET_NAME" \
    -o jsonpath='{.spec.replicas}'
)

✓ VKS PV -> Supervisor PVC -> Supervisor PV -> FCD relationship recorded
```

Stage 3: Quiesce and snapshot the source volume

Here, a Supervisor `VolumeSnapshot` will prevent the deletion of the Supervisor PV. This retention behaviour must first be validated for the deployed VKS/VCF version.

Quiesce the workloads:

```
## STAGE 3 — Quiesce and protect the source volume

src_vks -n "$SRC_VKS_NS" scale \
  "deployment/$DEPLOYMENT_NAME" \
  "statefulset/$STATEFULSET_NAME" \
  --replicas=0

src_vks -n "$SRC_VKS_NS" rollout status \
  "deployment/$DEPLOYMENT_NAME" \
  --timeout=300s

src_vks -n "$SRC_VKS_NS" rollout status \
  "statefulset/$STATEFULSET_NAME" \
  --timeout=300s
```

Create the Supervisor snapshot after quiescing:

```
export SRC_SNAPSHOT="${SRC_SUP_PVC}-migration-snapshot"
export SRC_SNAPSHOT_CLASS="<source-supervisor-snapshot-class>"

cat <<EOF | src_sup apply -f -
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
```

```
  name: ${SRC_SNAPSHOT}
  namespace: ${SRC_SUP_NS}
spec:
  volumeSnapshotClassName: ${SRC_SNAPSHOT_CLASS}
  source:
    persistentVolumeClaimName: ${SRC_SUP_PVC}
EOF

src_sup -n "$SRC_SUP_NS" wait \
  --for=jsonpath='{.status.readyToUse}'=true \
  "volumesnapshot/${SRC_SNAPSHOT}" \
  --timeout=5m
```

Create the manifest-only Velero backup:

```
velero backup create "$BACKUP_NAME" \
  --kubeconfig "$SRC_VKS_KUBECONFIG" \
  --include-namespaces "$SRC_VKS_NS" \
  --include-cluster-resources=false \
  --snapshot-volumes=false \
  --exclude-resources \
    pv,pvc,volumesnapshots,volumesnapshotcontents

velero backup describe "$BACKUP_NAME" \
  --kubeconfig "$SRC_VKS_KUBECONFIG" \
  --details
```

After the backup completes, release the source VKS objects:

```
src_vks -n "$SRC_VKS_NS" delete pvc "$PVC_NAME"
src_vks delete pv "$SRC_VKS_PV"
```

Confirm there is no reference to the source PV from VKS:

```
src_vks get volumeattachments.storage.k8s.io -o json |
  jq -e --arg pv "$SRC_VKS_PV" \
    '[.items[] |
      select(.spec.source.persistentVolumeName == $pv)] |
     length == 0'
```

Confirm that the Supervisor objects and FCD remain

```
test "$(
  src_sup -n "$SRC_SUP_NS" get pvc "$SRC_SUP_PVC" \
    -o jsonpath='{.spec.volumeName}'
)" = "$SRC_SUP_PV"

test "$(
  src_sup get pv "$SRC_SUP_PV" \
    -o jsonpath='{.spec.csi.volumeHandle}'
)" = "$SRC_FCD_UUID"


✓ Workloads stopped, backup complete, FCD retained and detached from VKS
```

Do not proceed if the Supervisor chain has unexpectedly disappeared or the FCD remains attached to a node.

Stage 4: vMotion the FCD to the destination vCenter

Here we create a temporary 'helper VM', which is only a transport envelope; it should **NOT** be powered on under any circumstances.

With govc configured for the source vCenter:

```
## STAGE 4 – vMotion the FCD to the destination vCenter

export HELPER_VM="${SRC_FCD_UUID}-migrator"
export GOVC_DATASTORE="<source-datastore>"
export GOVC_RESOURCE_POOL="<source-resource-pool>"

export SRC_FCD_PATH=$(
  govc disk.ls -json "$SRC_FCD_UUID" |
    jq -er '.objects[0].config.backing.filePath'
)

govc vm.create \
  -on=false \
  -net=none \
  "$HELPER_VM"

govc vm.disk.attach \
  -vm "$HELPER_VM" \
  -disk "$SRC_FCD_PATH" \
  -link=false
```

In the source vCenter UI, perform a cross-vCenter migration of the powered-off helper VM. Note: the relevant (source) storage policy should be applied to the destination datastore beforehand:

1. Select both destination compute and storage
2. Choose a datastore accessible to the destination Supervisor
3. Complete all compatibility checks
4. Migrate the helper VM and its attached FCD
5. Confirm the helper VM remains powered off

After the vMotion, configure govc for the destination vCenter and discover the migrated FCD.
If UUID preservation has already been validated, the original UUID should resolve:

```
export DST_FCD_PATH=$(
  govc disk.ls -json "$SRC_FCD_UUID" |
    jq -er '.objects[0].config.backing.filePath'
)

export DST_FCD_UUID="$SRC_FCD_UUID"

test -n "$DST_FCD_PATH"
test "$DST_FCD_UUID" = "$SRC_FCD_UUID"
```

If the original UUID does not resolve, stop and discover the actual destination identifier from the helper VM's disk backing. Do **not** submit the source identifier to `CnsRegisterVolume`.

Detach the FCD from the helper VM without deleting it, for example:

```
export DST_FCD_PATH=$(
govc vm.disk.detach \
  -vm "$HELPER_VM" \
  "$DST_FCD_PATH"

✓ FCD present, policy-compliant and unattached in the destination vCenter
```

Important: do **not** delete the helper VM until the FCD has been registered and validated.

Stage 5: Reconstruct the destination Supervisor PV and PVC

Use the `CnsRegisterVolume` CR in the destination Supervisor namespace to create the Supervisor PV/PVC pair:

```
## Stage 5 — Reconstruct the destination Supervisor PV and PVC

cat <<EOF | dst_sup apply -f -
apiVersion: cns.vmware.com/v1alpha1
kind: CnsRegisterVolume
metadata:
  name: ${REGISTER_NAME}
  namespace: ${DST_SUP_NS}
spec:
  volumeID: "${DST_FCD_UUID}"
  accessMode: ReadWriteOnce
  pvcName: ${DST_SUP_PVC}
EOF
```

Resolve and verify the newly created destination Supervisor PV:

```
export DST_SUP_PV=$(
  dst_sup -n "$DST_SUP_NS" get pvc "$DST_SUP_PVC" \
    -o jsonpath='{.spec.volumeName}'
)

export DST_CAPACITY=$(
  dst_sup get pv "$DST_SUP_PV" \
    -o jsonpath='{.spec.capacity.storage}'
)

export REGISTERED_FCD=$(
  dst_sup get pv "$DST_SUP_PV" \
    -o jsonpath='{.spec.csi.volumeHandle}'
)

test "$REGISTERED_FCD" = "$DST_FCD_UUID"

✓ Destination Supervisor PVC and PV created
```

If registration reports a storage-class or policy mapping error, correct the destination policy assignment before proceeding.

Stage 6: Reconstruct the destination VKS PV and PVC

Create the destination namespace and static VKS PV. The `volumehandle` of the PV is the destination Supervisor PVC name:

```
## Stage 6 — Reconstruct the destination VKS PV and PVC

dst_vks create namespace "$DST_VKS_NS" \
  --dry-run=client \
  -o yaml |
  dst_vks apply -f -

cat <<EOF | dst_vks apply -f -
apiVersion: v1
kind: PersistentVolume
metadata:
  name: ${DST_VKS_PV}
spec:
  capacity:
    storage: ${DST_CAPACITY}
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  persistentVolumeReclaimPolicy: Retain
  storageClassName: ""
```

```
  claimRef:
    namespace: ${DST_VKS_NS}
    name: ${PVC_NAME}
  csi:
    driver: csi.vsphere.vmware.com
    fsType: ${FS_TYPE}
    volumeHandle: "${DST_SUP_PVC}"
    volumeAttributes:
      type: "vSphere CNS Block Volume"
---
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ${PVC_NAME}
  namespace: ${DST_VKS_NS}
spec:
  volumeMode: Filesystem
  accessModes:
    - ReadWriteOnce
  storageClassName: ""
  volumeName: ${DST_VKS_PV}
  resources:
    requests:
      storage: ${DST_CAPACITY}
EOF

✓ Namespace manifests restored; existing VKS PV and PVC retained
```

Verify the complete reconstructed chain:

```
test "$(
  dst_vks get pv "$DST_VKS_PV" \
    -o jsonpath='{.spec.csi.volumeHandle}'
)" = "$DST_SUP_PVC"

test "$(
  dst_sup get pv "$DST_SUP_PV" \
    -o jsonpath='{.spec.csi.volumeHandle}'
)" = "$DST_FCD_UUID"
```

Restore the portable PVC metadata captured from the source:

```
dst_vks -n "$DST_VKS_NS" patch pvc "$PVC_NAME" \
  --type=merge \
  --patch-file "$MIGRATION_DIR/portable-pvc-metadata.json"

✓ Destination VKS PVC bound
```

Stage 7: Restore application manifests

Restore the Velero backup, excluding all storage objects (because the backup was taken after the workloads were scaled to zero, the restored controllers should remain inactive until explicitly promoted):

```
## STAGE 7 — Restore application manifests

velero restore create "$RESTORE_NAME" \
  --kubeconfig "$DST_VKS_KUBECONFIG" \
  --from-backup "$BACKUP_NAME" \
  --namespace-mappings "${SRC_VKS_NS}:${DST_VKS_NS}" \
  --exclude-resources \
    persistentvolumes,\
persistentvolumeclaims,\
volumesnapshots.snapshot.storage.k8s.io,\
volumesnapshotcontents.snapshot.storage.k8s.io

velero restore describe "$RESTORE_NAME" \
```

```
  --kubeconfig "$DST_VKS_KUBECONFIG" \
  --details
```

Stage 7.5: Optional Helm Reconciliation

For Helm-managed applications, capture the source values and chart version before cutover:

```
## STAGE 7.5 — Optional Helm Reconciliation

helm get values payments-api \
  --namespace "$SRC_NS" \
  --kubeconfig "$SRC_KUBECONFIG" \
  --all \
  -o yaml \
  > supplied-values.yaml

helm get metadata payments-api \
  --namespace "$SRC_NS" \
  --kubeconfig "$SRC_KUBECONFIG"
```

Make the exact chart version available:

```
helm pull payments-chart \
  --version 2.4.1 \
  --destination ./helm-artifacts
```

If Velero restored the Helm release metadata, reconcile the existing release:

```
helm upgrade payments-api \
  ./helm-artifacts/payments-chart-2.4.1.tgz \
  --namespace "$DST_NS" \
  --values supplied-values.yaml \
  --kubeconfig "$VKS_KUBECONFIG"
```

If no Helm release metadata exists on the destination, reconstruct the release and deliberately adopt the restored resources:

```
helm upgrade --install payments-api \
  ./helm-artifacts/payments-chart-2.4.1.tgz \
  --namespace "$DST_NS" \
  --values supplied-values.yaml \
  --kubeconfig "$VKS_KUBECONFIG" \
  --take-ownership

✓ Helm release reconciled with the restored Kubernetes resources
```

Note: Do not delete Helm release Secrets indiscriminately. Determine whether the release metadata was restored, then use either the existing-release or release-reconstruction path.

Stage 8: Activate and validate the destination workloads

Activate the destination using the recorded source replica counts or separately approved destination values and validate the rollout:

```
dst_vks -n "$DST_VKS_NS" scale \
  "statefulset/$STATEFULSET_NAME" \
  --replicas="$SOURCE_STS_REPLICAS"

dst_vks -n "$DST_VKS_NS" scale \
  "deployment/$DEPLOYMENT_NAME" \
```

```
  --replicas="$SOURCE_DEPLOY_REPLICAS"

dst_vks -n "$DST_VKS_NS" rollout status \
  "statefulset/$STATEFULSET_NAME" \
  --timeout=300s

dst_vks -n "$DST_VKS_NS" rollout status \
  "deployment/$DEPLOYMENT_NAME" \
  --timeout=300s

test "$(
  dst_vks -n "$DST_VKS_NS" get pvc "$PVC_NAME" \
    -o jsonpath='{.status.phase}'
)" = "Bound"

test "$(
  dst_sup get pv "$DST_SUP_PV" \
    -o jsonpath='{.spec.csi.volumeHandle}'
)" = "$DST_FCD_UUID"

APP_HEALTH=$(
  curl -fsS https://payments.vks.corp.local/health |
    jq -r '.status'
)

test "$APP_HEALTH" = "healthy"

✓ Destination validated and migration wave promoted to stable
```

The promotion gate should also confirm:

- Expected data is present and current.
- Destination CNS reports the volume as healthy.
- The storage policy reports compliant.
- No source workload has restarted.
- Routing or DNS directs traffic only to the destination.

Stage 9: Source cleanup

Source-side cleanup must occur only after the migration has passed its acceptance window.

Because the FCD has left the source vCenter, source Supervisor PVC, PV and snapshot objects may become orphaned Kubernetes objects even though the disk is no longer present in source CNS.

Cleanup should therefore:

1. Confirm the FCD is healthy and registered in the destination vCenter
2. Confirm the helper VM no longer has the FCD attached
3. Delete the helper VM
4. Remove the source Supervisor snapshot
5. Remove the orphaned source Supervisor PVC/PV
6. Retain the Velero backup and migration records according to the rollback policy

Rollback Boundary

If validation fails, execute the following procedure:

Failure Boundary	Rollback Method	Expected RTO
Before source VKS storage release	Scale source workloads back up	Minutes
After source VKS storage release but before vMotion	Detach helper if necessary, recreate source VKS PV/PVC against the retained source Souervisor PVC, then reactivate source	Minutes
After cross-vCenter vMotion, but before destination activation	Reverse the helper-VM/FCD vMotion and reconstruct the source chain	Depends on vMotion duration
After destination activation and writes	Quiesce destination, move the current FCD back and reconstruct the source chain	Depends on vMotion duration
After source cleanup	Recovery from destination storage or backup	Recovery-dependant

Post-vMotion rollback procedure

If rollback is required after the FCD has been moved, perform the following:

1. Scale all destination workloads to zero
2. Confirm destination pods and `VolumeAttachment` objects have been removed
3. Either attach a snapshot to the Supervisor PVC or manually set to `Retain`
4. Remove the destination VKS PVC/PV
5. Release the destination Supervisor registration
6. Attach the retained FCD to the powered-off helper VM in the destination vCenter
7. Perform a reverse cross-vCenter vMotion to the source vCenter
8. Detach the FCD from the helper VM
9. Either validate the original source Supervisor objects or re-register the FCD in the source Supervisor using `CnsRegisterVolume`
10. Recreate the source VKS PV and PVC
11. Restore the captured PVC metadata
12. Scale the source workloads to their recorded replica counts
13. Validate the source application and data before restoring traffic

The rollback RTO must be measured during rehearsal for representative volume sizes. It is governed primarily by datastore placement, FCD size, network throughput and cross-vCenter vMotion duration.

If rollback uses a pre-cutover snapshot instead of returning the current FCD, all writes made after destination activation will be lost.

Technical Assistance

For technical inquiries or issues related to the migration methodologies outlined in this document, contact your assigned Broadcom Professional Services representative or designated account manager. They can provide guidance specific to your environment and escalate issues through the appropriate support channels as needed.

Conclusion

This method demonstrates that VKS-to-VKS migration does not require data to be extracted from one Kubernetes volume and copied into another. A conventional Velero-based backup and restore can be considerably slower because it stages volume data through an S3-compatible object store. pv-migrate offers a more direct alternative using rsync, but the approach demonstrated here takes advantage of vMotion's native integration with vSphere. By migrating the existing First Class Disk and reconstructing the Supervisor and VKS storage chain, the volume remains in its native form and no separate Kubernetes-level data mover is required.

Cross-vCenter migration nevertheless introduces important operational considerations. vMotion may still transfer the disk's storage blocks, while rollback after the move requires the FCD to be migrated back or recovered by another means. With version-specific testing, verified storage-policy mappings, reliable FCD identification, and rehearsed retention and rollback procedures, this approach provides a practical and repeatable path for migrating stateful VKS workloads across Supervisor and vCenter boundaries.

Appendix A: Example Resource Translation

Kubernetes workloads are often portable at the API level, but individual resources may still contain environment-specific values. During migration, these values should be translated deliberately rather than edited manually after the Velero restore. Below describes two examples where a configmap can be used on the destination Velero namespace to automatically translate values, without editing manifests

If the source and destination clusters use different container registries, restored workloads may continue to reference the source registry unless image references are rewritten during restore.

For example:

```
Source registry:      harbor.source.example.com
Destination registry: harbor.vks.example.com
```

Create a Velero image mapping ConfigMap on the destination cluster:

```
apiVersion: v1
kind: ConfigMap
metadata:
  name: change-image-name-config-modifier
  namespace: velero
  labels:
    velero.io/change-image-name: RestoreItemAction
    velero.io/plugin-config: ""
data:
  case1: harbor.source.example.com,harbor.vks.example.com
```

Apply it to the destination cluster:

```
kubectl --kubeconfig "$VKS_KUBECONFIG" apply \
  -f image-registry-mapping.yaml
```

Ingress resources commonly differ between source and destination environments. For example, a source cluster may use AKO, while the destination VKS cluster uses Contour.

Create a Velero resource-modifier rule:

```
version: v1
resourceModifierRules:
  - conditions:
      groupResource: ingresses.networking.k8s.io
      namespaces:
        - application-namespace
    patches:
      - operation: replace
        path: /spec/ingressClassName
        value: contour
```

Then create the ConfigMap in the destination Velero namespace:

```
kubectl --kubeconfig "$VKS_KUBECONFIG" \
  -n velero \
  create configmap ingress-modifier \
  --from-file=rules.yaml
```

Run the restore using the resource modifier:

```
velero restore create "$RESTORE_NAME" \
  --kubeconfig "$VKS_KUBECONFIG" \
  --from-backup "$BACKUP_NAME" \
  --namespace-mappings "${SOURCE_NS}:${DESTINATION_NS}" \
  --resource-modifier-configmap ingress-modifier
```

vmware®
by Broadcom

Item No: vmw-bc-wp-tech-uslet-word-2026; Jul-26