TECHNICAL WHITE PAPER:
July 2026

Migrating vSphere-Based Kubernetes Workloads to VKS Using Zero-Copy FCD Adoption

A Professional Services Guide for Fast-Path Migration from vSphere-based Kubernetes Distributions (OpenShift, Rancher, vanilla) to VKS on VCF 9

Contents

Executive Summary

This paper is one of four in the VKS migration series. Use this paper if your source already runs on vSphere CSI and you want the fastest, zero-copy path. For a source that is VKS itself, see Migrating Workloads Between VKS Clusters Using Cross-vCenter vMotion for the equivalent vMotion-based approach. For a source not running vSphere CSI, see Migrating Non-vSphere Kubernetes Workloads to VKS Using Filesystem Replication. For the fully validated, benchmarked Velero+S3 reference methodology, see Migrating Kubernetes Workloads to VKS Using Velero and S3-Compatible Storage.

Broadcom, through its portfolio of VMware products, delivers vSphere Kubernetes Service (VKS) as the enterprise-grade Kubernetes runtime on VMware Cloud Foundation 9. As organizations modernize from legacy Kubernetes distributions -- including Tanzu Kubernetes Grid, Red Hat OpenShift, SUSE Rancher, and "vanilla" Kubernetes running on vSphere and utilizing the vSphere CSI driver for persistent volumes -- migrating to VKS delivers a consistent, auditable operational plane managed through the Supervisor control plane.

This white paper outlines a structured, four-phase migration framework: Discover, Analyze, Blueprint, and Execute. While Velero is the de-facto open-source standard for Kubernetes disaster recovery, designed to safely back up, restore, and migrate cluster resources and persistent volumes, data needs to be actively copied across clusters, which relies on both network bandwidth and compute resources. Here, we combine Velero for stateless manifest and metadata migration and use the CnsRegisterVolume API to seamlessly adopt persistent volumes. This architecture effectively bypasses traditional data-copy overhead, reducing migration windows for vSphere CSI-backed storage from hours to minutes.

NOTE

This methodology applies to Kubernetes workloads running on vSphere with vSphere CSI storage. Non-vSphere storage (AWS EBS, NFS, Ceph) requires the data-copy fallback described in Migrating Kubernetes Workloads to VKS Using Velero and S3-Compatible Storage and Migrating Non-vSphere Kubernetes Workloads to VKS Using Filesystem Replication.

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.

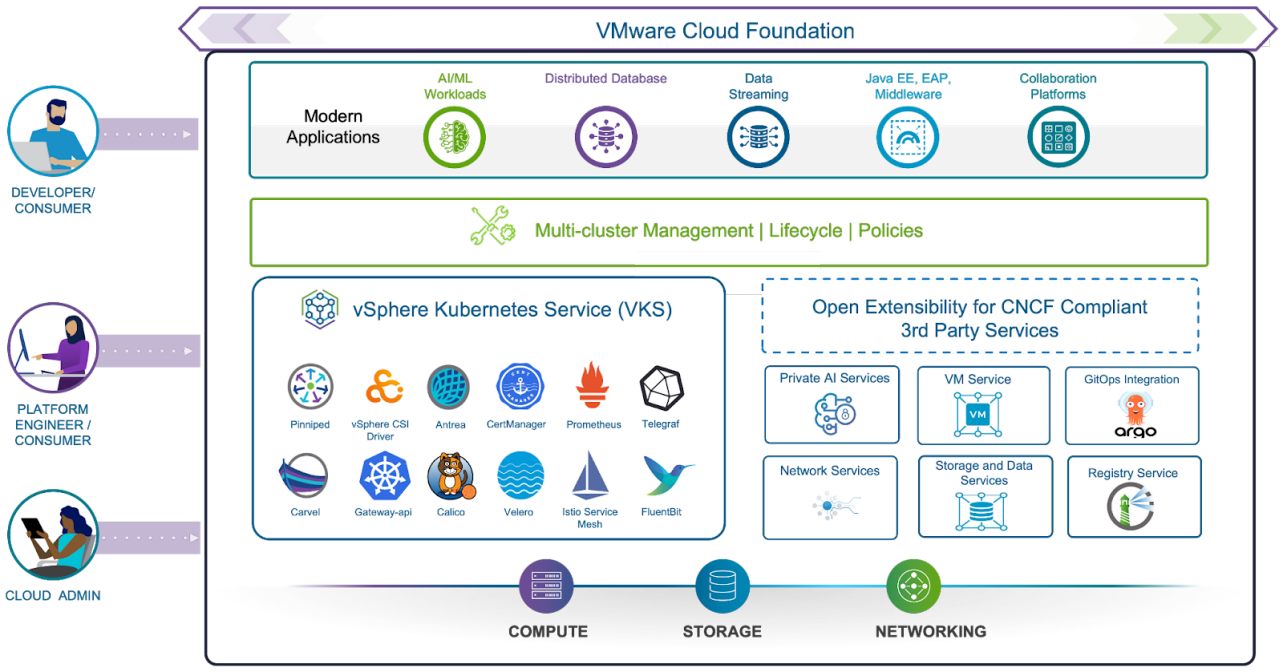
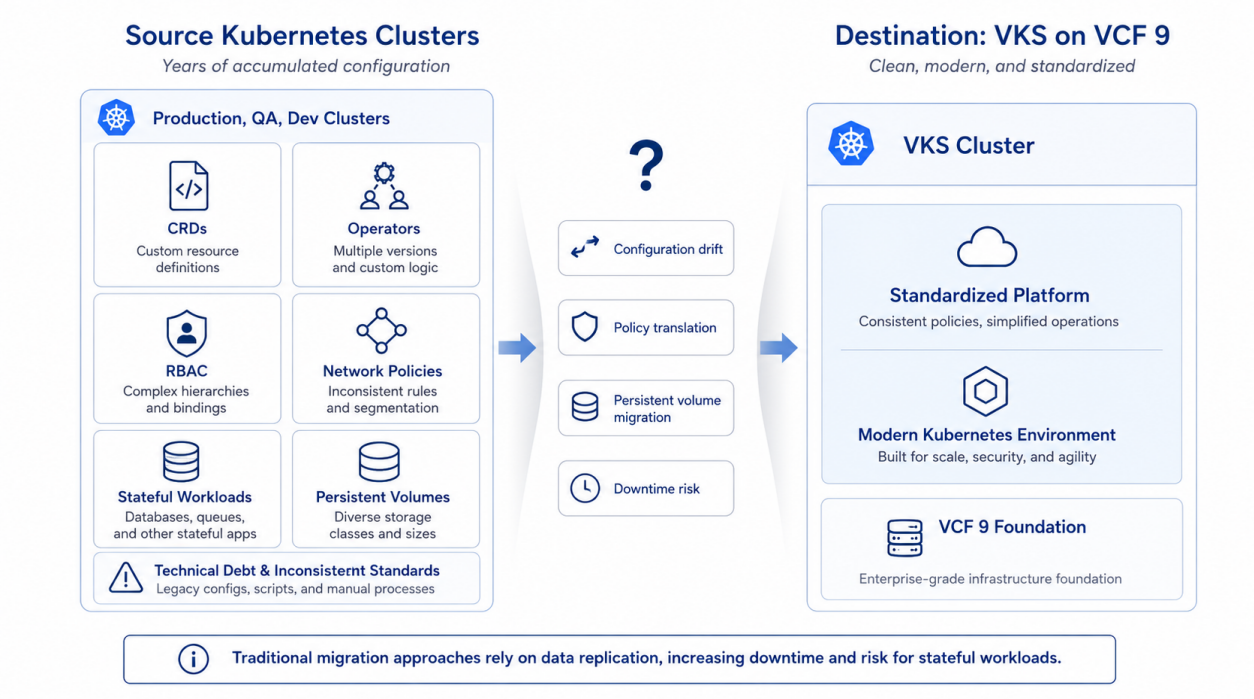


Figure 1: VMware Cloud Foundation with vSphere Kubernetes Service

The Migration Challenge

Customers operating Kubernetes workloads on vSphere face a complex migration challenge when transitioning to VKS on VCF 9. Source clusters contain years of accumulated configuration: custom CRDs, operator frameworks, RBAC hierarchies, Network Policy definitions, and stateful workloads with substantial persistent volume datasets. Traditional migration approaches rely on data replication, introducing unacceptable downtime windows and data integrity risks for production databases and message queues.



Key Pain Points

Below we summarize the typical issues encountered with migrating Kubernetes workloads:

- **Stateful workload risk:** PVCs backed by large datasets create multi-hour migration windows when using data-copy approaches.
- **Platform-specific resources:** OpenShift Routes, Rancher Projects, and TKGi NSX-T bindings have no direct VKS equivalent and require manual translation.
- **RBAC and secrets complexity:** Multi-namespace RBAC bindings and Vault/External Secrets integrations require precise remapping.
- **Compliance exposure:** PCI-DSS, HIPAA, and GDPR-tagged workloads require compliance officer approval before any wave assignment.
- **Rollback confidence:** Without pre-migration snapshots and validation gates, rollback procedures are undocumented and high-risk.
- **Helm release metadata collision:** Migrating namespaces without exclusion filters preserves Helm secrets (*sh.helm.release.v1*) on the destination. This legacy state prevents *helm upgrade* commands from successfully reconciling target resources, leading to failed adoption or inadvertent deployment rollbacks.

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

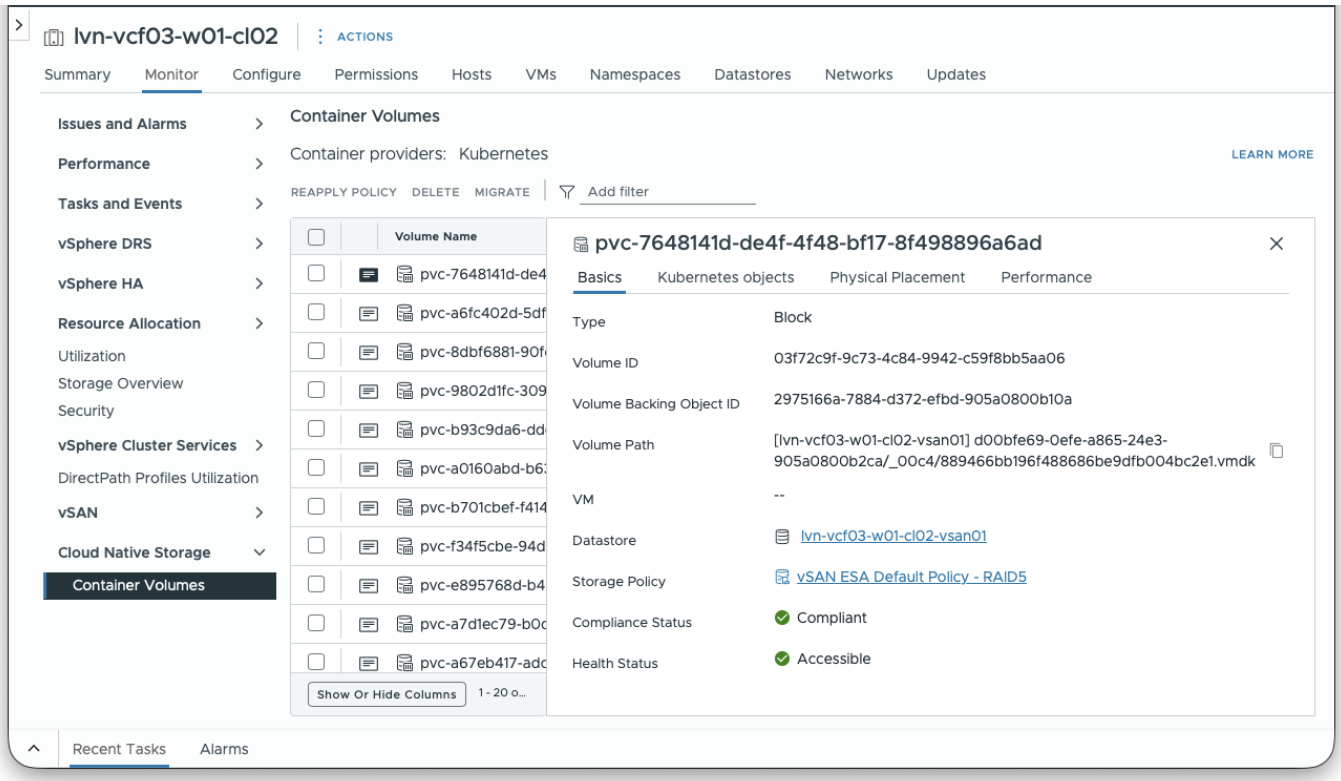
Solution Architecture

vSphere FCD, CSI Driver and Persistent Storage

For Kubernetes distributions running on vSphere, each Kubernetes node runs on a single Virtual Machine. Kubernetes workloads requesting persistent storage via the vSphere Container Storage Interface (CSI) driver will have their volumes provisioned via a virtual disk mounted on the worker node. The vSphere CSI driver bridges the requests from the Kubernetes layer to vSphere; thus, a request for a Persistent Volume will end in a vSphere task to provision storage from a given datastore (ultimately governed by the Kubernetes Storage Class, via a vSphere Storage Policy) which will create a vSphere First Class Disk of the requested specifications.

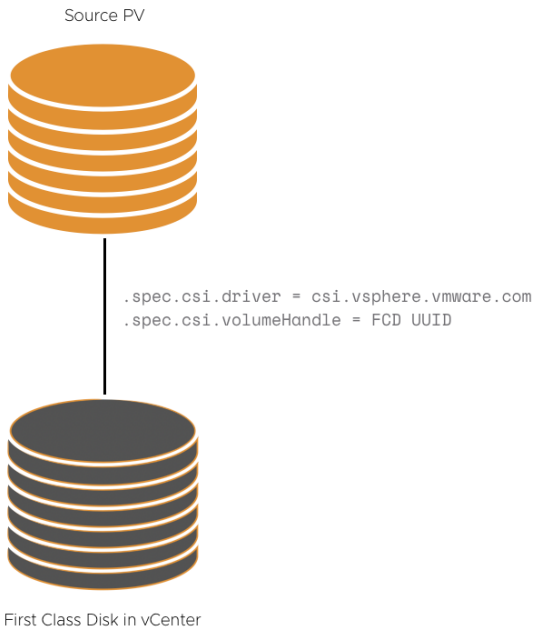
A First Class Disk, or FCD, is a vSphere-managed virtual disk with an identity and lifecycle independent of any particular Virtual Machine and unlike a conventional Virtual Machine disk, each FCD is a standalone storage object rather than as a disk permanently owned by a Virtual Machine.

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).



Direct vSphere CSI Clusters

For non-VKS Kubernetes clusters (upstream, OpenShift, etc.) the identity chain is relatively simple. Here, the CSI volumeHandle directly identifies the underlying FCD.

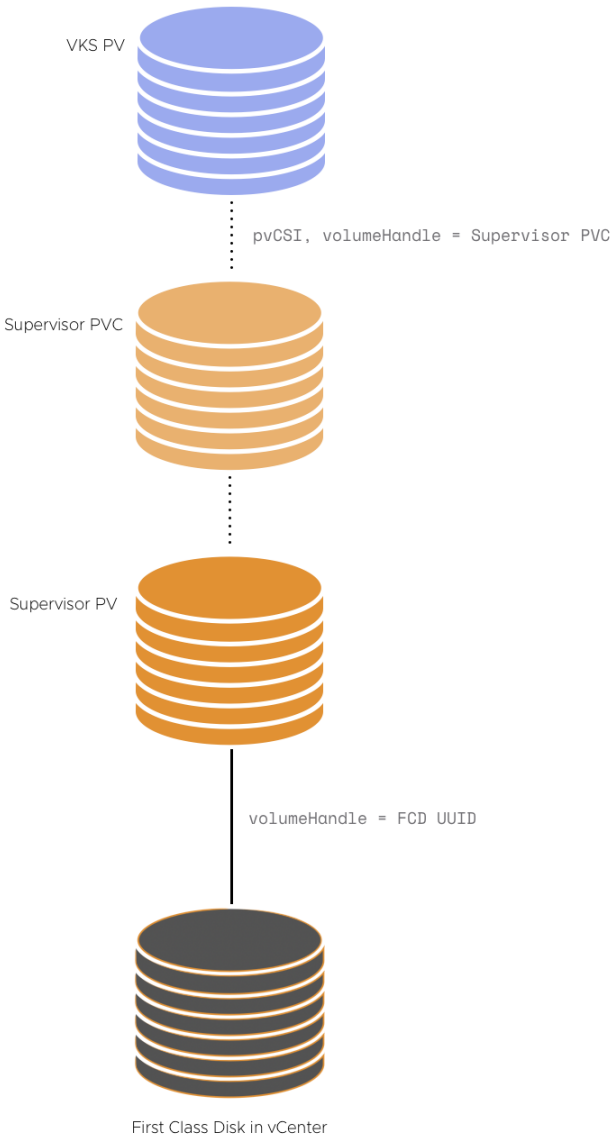


The FCD UUID can therefore be discovered directly from the source PV, for example:

```
PV_NAME=$(
  kubectl -n "$NAMESPACE" get pvc "$PVC_NAME" \
    -o jsonpath='{.spec.volumeName}'
)
FCD_UUID=$(
  kubectl get pv "$PV_NAME" \
    -o jsonpath='{.spec.csi.volumeHandle}'
)
```

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 PVC 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.



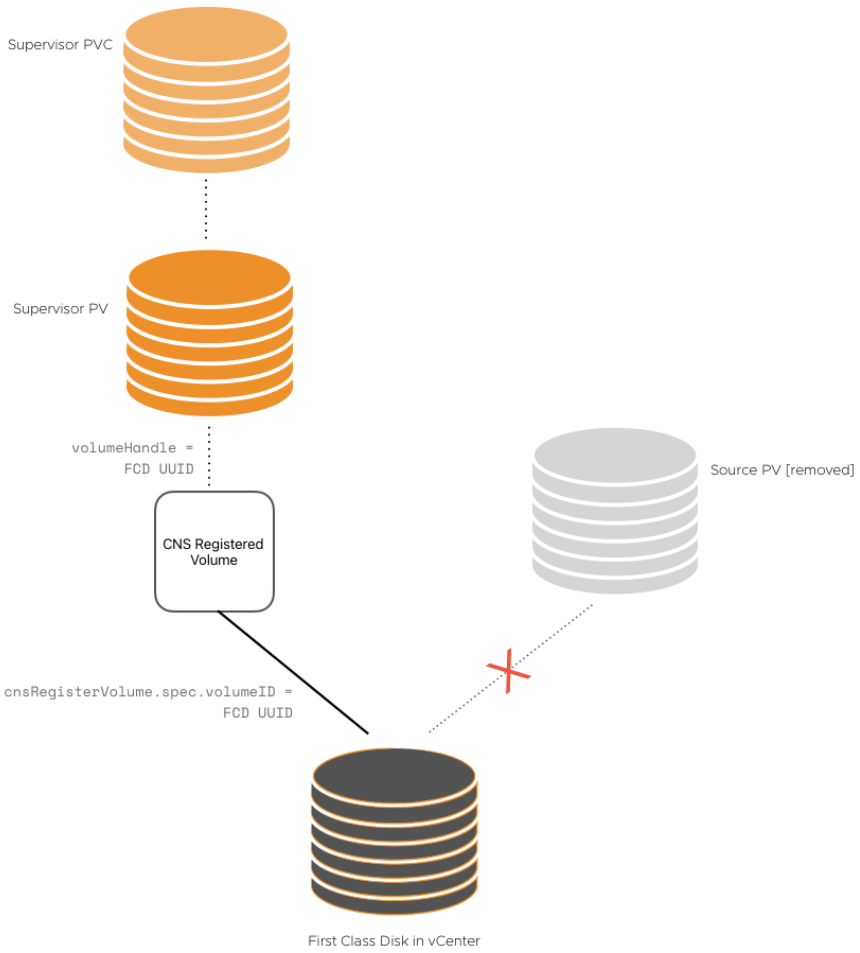
For more details visit https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vcf-consumption/latest/managing-vsphere-kuberenetes-service-clusters-and-workloads/deploying-workloads-on-tkg-service-clusters/storage-concepts-for-tkg-service-clusters.html

Registering an Existing FCD on the Supervisor

Now, if an FCD from a direct CSI cluster already exists as an entity in vCenter, we can adopt the volume by altering the ownership chain. We do this by leveraging `CnsRegisterVolume` Custom Resource (CR) which reverses the normal provisioning direction.

A simplified registration object is:

```
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
```



Here the `volumeID` is the existing source FCD UUID (from the direct CSI provisioner). The `CnsRegisterVolume` object creates the corresponding Supervisor PV and PVC (the Supervisor PV `volumeHandle` matches the `volumeID` of the registered volume/FCD UUID, and it addresses the FCD directly). Note this CR must be created in the Supervisor namespace associated with the destination VKS cluster.

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

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

Thus, the volume has been adopted by VKS without any data copy operation.

Although the FCD remains in place, it must not be actively used by both the original cluster and VKS at the same time (if the volume is still mounted on a Kubernetes node, vCenter will throw a `FailedAttachedVolume` error detailing the specifics). Thus, the Kubernetes workloads on the source cluster must be scaled to zero and the PVC removed (the CSI driver will unmount the volume from the node at this point) before the volume can be used by VKS.

Before releasing the source claim, it is important to change the PV reclaim policy to Retain. This separates the deletion of the Kubernetes claim from the deletion of the FCD.

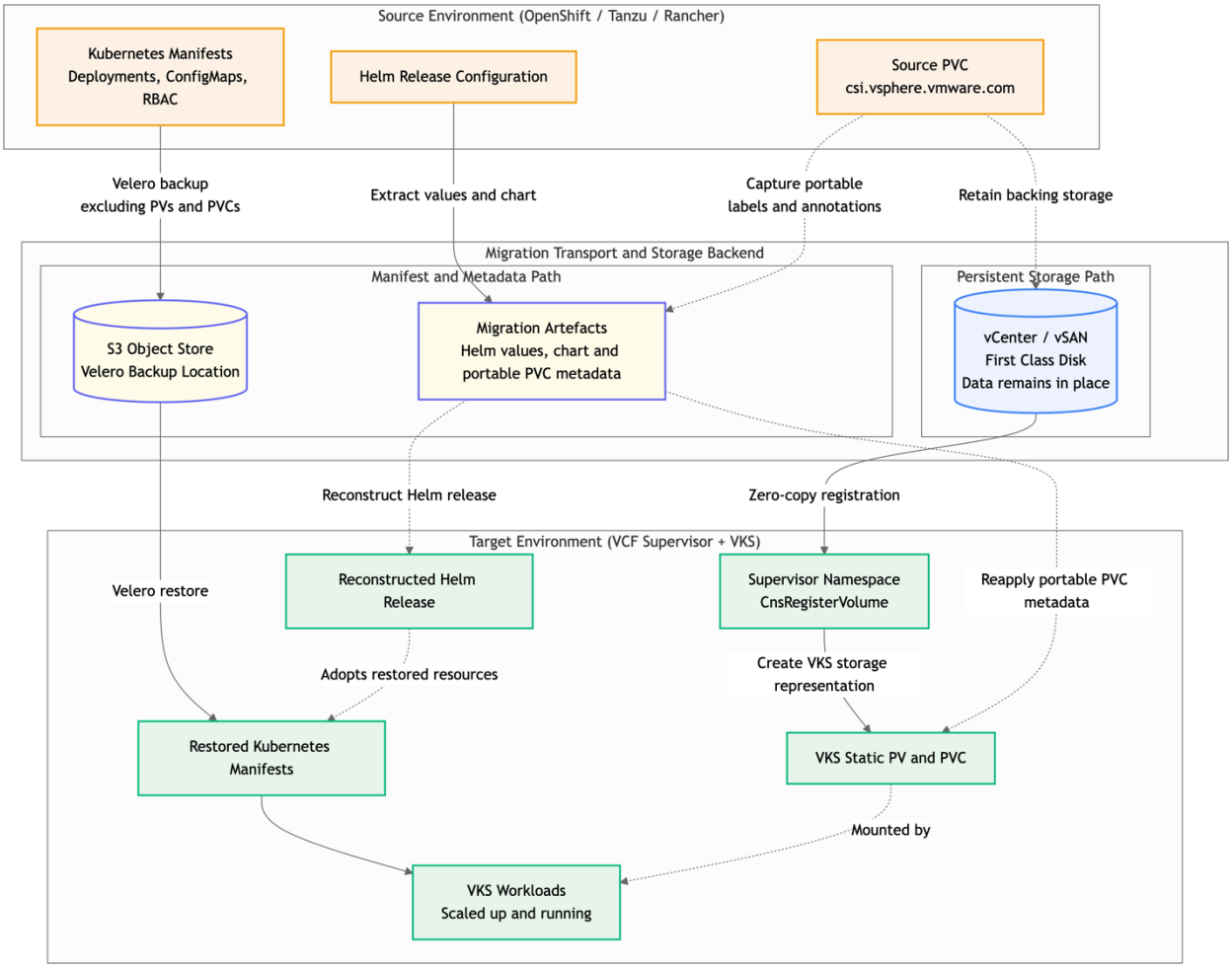
Migration Architecture Overview

Given that we are able to seamlessly register and adopt FCD volumes using the available Supervisor CR, we can weave this into a sound migration strategy. Furthermore, since the CNS driver supports volume snapshotting a rollback procedure can also be envisioned.

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 guest cluster on VCF 9. Stateless manifests and configurations are transported via Velero to an S3 object store backup location.

Meanwhile, the stateful vSphere FCDs remain securely in place on the datastore, bypassing the data migration network entirely. The `CnsRegisterVolume` API is then utilized on the VCF 9 Supervisor to adopt these existing FCDs directly into the new VKS cluster. By utilizing this zero-copy FCD adoption path, the migration timeline is restricted only by API synchronization speeds rather than network bandwidth limitations.

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 Cluster	Migration Tooling	Target: VKS on VCF 9
Direct vSphere CSI consumers, such as: Upstream / Tanzu / OpenShift Rancher / K8s	Velero + S3 CnsRegisterVolume FCD Registry	Supervisor NS VKS Guest Cluster vSphere CSI

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 (zero-copy)	Medium
PVCs — other storage	14	Velero + Restic/Kopia data copy	High
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 (§6)	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 zero-copy, 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 / oc	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 analysed. 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. If the source cluster exposes the OpenShift Routes API, Routes are collected as a separate platform-specific artefact.

```
## TIER 2 — Network Topology

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

if kubectl api-resources \
     --api-group=route.openshift.io \
     -o name |
   grep -qx 'routes.route.openshift.io'; then
     kubectl get routes.route.openshift.io \
      --all-namespaces \
      -o json \
      > /tmp/vks-discovery/ocp_routes.json

else
  echo "[SKIP] OpenShift Routes API not present"

fi

✓ 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

For a Kubernetes source cluster using the vSphere CSI driver directly, including OpenShift on vSphere, the source PV volumeHandle identifies the underlying FCD.

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

kubectl get pvc -A -o json |
  jq -r '.items[] | [.metadata.namespace,.metadata.name,.spec.volumeName] | @tsv' |
  while IFS=$'\t' read -r ns pvc pv; do
    fcd=$(kubectl 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.jsonl

govc disk.ls -json -q metadataKey.eq=cns.k8s.pvc.namespace |
  jq --slurpfile mappings /tmp/vks-discovery/fcd_mappings.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**
>
> UUID Integrity: Cross-datastore Storage vMotion can alter FCD UUIDs.
>
> Run `govc disk.ls -json` after any storage migration and verify UUIDs match the catalogue before proceeding to the Execute phase.

Phase 2: Analyze

Platform-Specific Resource Translation

In an ideal world, both source and destination Kubernetes clusters will have exactly the same routes, security contexts, image locations and storage classes, etc. In practice, many of these will need to be translated. In this Phase, it is essential that the differences between the clusters be mapped out, along with an action on how this is to be translated. The table below shows some examples that may need consideration, but this will vary significantly in your environment.

Source Resource	Platform	VKS Equivalent	Action
Route (edge/passthrough)	OpenShift	Ingress + TLS Secret	Auto-translate via converter
Project (namespace wrapper)	OpenShift	Namespace + RBAC	Namespace + RoleBinding copy
SecurityContextConstraints	OpenShift	PodSecurityAdmission	Manual policy mapping
Rancher Project	Rancher	Namespace + VM-Class	Namespace label remapping
GlobalNetworkPolicy	Rancher	NetworkPolicy	Auto-translate via Calico
TKGi NSX-T Binding	Tanzu TKGi	VKS NSX-T config	Supervisor network update
PodSecurityPolicy (depr.)	Any pre-1.25	PodSecurityAdmission label	Per-NS enforcement mode
StorageClass vsphere-volume	Legacy vSphere	csi.vsphere.volume	SC manifest update

See Appendix A for an example of resource translation for network ingress.

PVC Migration Decision Framework

Below is a comparison between the zero-copy (using the `CnsRegisterVolume` CR) and the 'traditional' data-copy path and associated method decision. For example, the zero-copy method can only be used when the vSphere CSI driver is used by the source Kubernetes cluster.

Decision Criterion	Zero-Copy Path	Data-Copy Path
Storage driver	vSphere CSI (csi.vsphere.vmware.com)	Any other CSI driver
First Class Disk (FCD) in vCenter Database	Yes → CnsRegisterVolume	No → Velero + Restic
Cross-datastore move	No	Yes (UUID risk)
Compliance tag	Manual sign-off, then zero-copy	Manual sign-off + audit log
Time for 100 GB volume	< 2 minutes	45–90 minutes
Tooling	CnsRegisterVolume CR	velero restore + restic

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 quiesces the selected source workloads, preserves the backing FCDs, registers them with the destination Supervisor, restores the Kubernetes manifests and then activates the workloads on VKS.

The example below uses a single namespace and PVC. Repeat the same sequence for each approved migration wave.

The migration is split into two independent paths:

- **The Kubernetes object path:** Velero migrates namespace-scoped application manifests and metadata.
- **The storage path:** the existing vSphere First Class Disk is detached from the source workload, registered with the VKS Supervisor using `CnsRegisterVolume`, and then presented to the VKS guest cluster through a statically bound PV and PVC.

No application data is copied between clusters. The existing FCD remains on the vSphere storage platform; only its Kubernetes and CNS associations are reconstructed.

Scope and assumptions

This simplified example supports the following baseline case:

- The source volume is an existing vSphere FCD using the vSphere CSI driver
- The source and destination use the same vCenter/CNS storage domain, or the destination Supervisor can otherwise access the FCD
- The volume is ReadWriteOnce and is Filesystem-based
- The destination is a VKS guest cluster
- A suitable vSphere Namespace already exists on the Supervisor
- The required storage policy is assigned to that Supervisor namespace
- Velero is installed on both source and destination clusters, and both installations can access the same S3 endpoint (BackupStorageLocation)
- The workload can tolerate a short, controlled outage
- Only one cluster will mount and write to the FCD at any point in time

For simplicity, this example deliberately omits features such as persistent state files, retries, ownership labels, concurrency checks, rollback automation and finalizer recovery.

Stage 0: Environment Setup

Three Kubernetes API endpoints are used:

- The source Kubernetes cluster
- The destination VKS cluster
- The Supervisor cluster associated with the destination VKS cluster

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

export SRC_KUBECONFIG="$HOME/.kube/source-config"
export VKS_KUBECONFIG="$HOME/.kube/vks-config"
export SUP_KUBECONFIG="$HOME/.kube/supervisor-config"

export SRC_NS="prod-payments"
export DST_NS="prod-payments"
export SUP_NS="migration-target"
export PVC_NAME="payments-db"
export WORKLOAD_NAME="payments-api"
export BACKUP_NAME="wave1-prod-payments"
export RESTORE_NAME="wave1-prod-payments"

src() { kubectl --kubeconfig "$SRC_KUBECONFIG" "$@"; }
```

```
vks() { kubectl --kubeconfig "$VKS_KUBECONFIG" "$@"; }
sup() { kubectl --kubeconfig "$SUP_KUBECONFIG" "$@"; }

src cluster-info
vks cluster-info
sup 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 BackupStorageLocation 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
```

Stage 2: Quiesce and Protect the Source Workload

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

```
## STAGE 2 — Quiesce and Protect Source Storage

export SOURCE_PV=$(
  src -n "$SRC_NS" get pvc "$PVC_NAME" \
    -o jsonpath='{.spec.volumeName}'
)
export FCD_UUID=$(
  src get pv "$SOURCE_PV" \
    -o jsonpath='{.spec.csi.volumeHandle}'
)
export CAPACITY=$(
  src get pv "$SOURCE_PV" \
    -o jsonpath='{.spec.capacity.storage}'
)
export FS_TYPE=$(
  src get pv "$SOURCE_PV" \
    -o jsonpath='{.spec.csi.fsType}'
)

printf 'Source PV: %s\nFCD UUID: %s\n' \
  "$SOURCE_PV" "$FCD_UUID"
```

Protect the FCD from automatic deletion:

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

Quiesce the application (note: A StatefulSet can also be scaled to zero. Operators, DaemonSets, Jobs and application-managed replicas require workload-specific quiescence):

```
src -n "$SRC_NS" scale \
```

```
  "deployment/$WORKLOAD_NAME" \
  --replicas=0
```

Create a cutover snapshot:

```
cat <<EOF | src apply -f -
apiVersion: snapshot.storage.k8s.io/v1
kind: VolumeSnapshot
metadata:
  name: payments-db-cutover
  namespace: ${SRC_NS}
spec:
  volumeSnapshotClassName: <vSphere-snapshot-class>
  source:
    persistentVolumeClaimName: ${PVC_NAME}
EOF
```

Create a manifest-only Velero backup while the workload remains scaled down:

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

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

Once the backup is completed, release the source claim (note: the source PV should remain present with a Retain reclaim policy and the original FCD UUID):

```
src -n "$SRC_NS" delete pvc "$PVC_NAME"

✓ Workload quiesced, snapshot ready, manifest backup completed and PV claim removed
```

Stage 3: Register the FCD with the Destination Supervisor

Create a CnsRegisterVolume object in the Supervisor namespace associated with the destination VKS cluster:

```
## STAGE 3 — FCD Zero-Copy Adoption

export SUPERVISOR_PVC_NAME="payments-db-adopted"
export REGISTER_NAME="register-payments-db"

cat <<EOF | sup create -f -
apiVersion: cns.vmware.com/v1alpha1
kind: CnsRegisterVolume
metadata:
  name: ${REGISTER_NAME}
  namespace: ${SUP_NS}
spec:
  pvcName: ${SUPERVISOR_PVC_NAME}
  volumeID: ${FCD_UUID}
  accessMode: ReadWriteOnce
EOF
```

Verify that the resulting Supervisor PV references the original FCD:

```
export SUPERVISOR_PV=$(
```

```
  sup -n "$SUP_NS" get pvc "$SUPERVISOR_PVC_NAME" \
    -o jsonpath='{.spec.volumeName}'
)

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

test "$REGISTERED_FCD" = "$FCD_UUID"

✓ FCD registered and represented by a bound Supervisor PVC
```

No application data is copied during registration. The existing FCD remains on the vSphere storage platform and is associated with the destination Supervisor namespace.

Stage 4: Create the VKS PV and PVC

Create the destination namespace and a static VKS PV. In the VKS guest cluster, the PV volumeHandle is the Supervisor PVC name, not the underlying FCD UUID.

```
## STAGE 4 — Create VKS Storage Objects

export VKS_PV_NAME="payments-db-zero-copy"

vks create namespace "$DST_NS"

cat <<EOF | vks apply -f -
apiVersion: v1
kind: PersistentVolume
metadata:
  name: ${VKS_PV_NAME}
spec:
  capacity:
    storage: ${CAPACITY}
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  persistentVolumeReclaimPolicy: Retain
  storageClassName: ""
  claimRef:
    namespace: ${DST_NS}
    name: ${PVC_NAME}
  csi:
    driver: csi.vsphere.vmware.com
    volumeHandle: ${SUPERVISOR_PVC_NAME}
    fsType: ${FS_TYPE}
EOF
```

Create the matching VKS PVC:

```
cat <<EOF | vks apply -f -
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: ${PVC_NAME}
  namespace: ${DST_NS}
spec:
  accessModes:
    - ReadWriteOnce
  volumeMode: Filesystem
  storageClassName: ""
  volumeName: ${VKS_PV_NAME}
  resources:
    requests:
      storage: ${CAPACITY}
```

```
EOF

✓ VKS PV and PVC bound to the adopted Supervisor volume
```

Stage 5: Preserve Application Metadata on Reconstructed PVCs

The statically reconstructed destination PVC will bind and mount correctly without the source metadata. However, application controllers, operators, Helm charts, backup tools or policy engines may rely on specific labels or annotations to discover and manage the claim.

Application-owned PVC metadata should therefore be captured before the source claim is deleted and reapplied after the destination PVC has been created.

Capture portable PVC metadata (the filter removes common Kubernetes and CSI annotations associated with dynamic provisioning, node selection and source-cluster binding):

```
## Capture source PVC labels and application-owned annotations

src -n "$SRC_NS" get pvc "$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")
          )
        )
      )
    }
  }' > /tmp/portable-pvc-metadata.json
```

Apply the metadata to the destination PVC:

```
vks -n "$DST_NS" patch pvc "$PVC_NAME" \
  --type=merge \
  --patch-file /tmp/portable-pvc-metadata.json

✓ Application-owned PVC labels and annotations restored
```

For production use, an explicit allowlist of approved application annotation prefixes is preferable to copying all annotations except known Kubernetes system keys.

Stage 6: Restore the Namespace Manifests

Restore the namespace while continuing to exclude the storage objects. Note that because the source workload was backed up after being scaled to zero, the restored workload remains inactive until the migration is explicitly promoted.

```
## STAGE 5 – Velero Manifest Restore

velero restore create "$RESTORE_NAME" \
  --kubeconfig "$VKS_KUBECONFIG" \
  --from-backup "$BACKUP_NAME" \
  --namespace-mappings "${SRC_NS}:${DST_NS}" \
  --exclude-resources \
```

```
    persistentvolumes,\
    persistentvolumeclaims,\
    volumesnapshots.snapshot.storage.k8s.io,\
    volumesnapshotcontents.snapshot.storage.k8s.io

velero restore describe "$RESTORE_NAME" \
  --kubeconfig "$VKS_KUBECONFIG" \
  --details

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

Stage 6.5: Reconcile Helm State — Where Applicable

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

```
## STAGE 5.5 — 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 7: Activate the Workload on VKS

Scale the restored workloads to their approved destination replica counts:

```
vks -n "$DST_NS" scale \
  statefulset/postgres-payments \
  --replicas=3

vks -n "$DST_NS" scale \
  deployment/payments-api \
  --replicas=4

✓ StatefulSet postgres-payments: 3/3 Ready
✓ Deployment payments-api: 4/4 Ready
```

Phase 5: Monitor and Rollback

Post-Wave Validation

Validate the PVC binding, workload readiness and application health before accepting the migration wave:

```
## PHASE 5 — Post-Wave Validation Gate

# Confirm the migrated PVC is bound
vks -n "$DST_NS" wait \
  --for=jsonpath='{.status.phase}'=Bound \
  "pvc/$PVC_NAME" \
  --timeout=300s

# Confirm the restored controllers have completed rollout
vks -n "$DST_NS" rollout status \
  statefulset/postgres-payments \
  --timeout=300s
vks -n "$DST_NS" rollout status \
  deployment/payments-api \
  --timeout=300s

# Read the final Kubernetes state
PVC_PHASE=$(
  vks -n "$DST_NS" get pvc "$PVC_NAME" \
    -o jsonpath='{.status.phase}'
)
STS_DESIRED=$(
  vks -n "$DST_NS" get statefulset postgres-payments \
    -o jsonpath='{.spec.replicas}'
)
STS_READY=$(
  vks -n "$DST_NS" get statefulset postgres-payments \
    -o jsonpath='{.status.readyReplicas}'
)
DEPLOY_DESIRED=$(
  vks -n "$DST_NS" get deployment payments-api \
    -o jsonpath='{.spec.replicas}'
)
DEPLOY_READY=$(
  vks -n "$DST_NS" get deployment payments-api \
    -o jsonpath='{.status.readyReplicas}'
)
APP_HEALTH=$(
  curl -fsS https://payments.vks.corp.local/health |
    jq -r '.status'
)

# Enforce the validation gate
test "$PVC_PHASE" = "Bound"
test "${STS_READY:-0}" = "$STS_DESIRED"
test "${DEPLOY_READY:-0}" = "$DEPLOY_DESIRED"
test "$APP_HEALTH" = "healthy"
```

```
printf 'PVC %s: %s\n' "$PVC_NAME" "$PVC_PHASE"
printf 'StatefulSet postgres-payments: %s/%s Ready\n' "${STS_READY:-0}" "$STS_DESIRED"
printf 'Deployment payments-api: %s/%s Ready\n' "${DEPLOY_READY:-0}" "$DEPLOY_DESIRED"
printf 'Application health: %s\n' "$APP_HEALTH"
```

Expected output:

```
PVC payments-db: Bound
StatefulSet postgres-payments: 3/3 Ready
Deployment payments-api: 4/4 Ready
Application health: healthy

✓ Validation passed — wave1-prod-payments promoted to stable
```

Rollback Boundary

If validation fails, execute the following procedure:

Step	Description	Sample Action	Indicative RTO
1	Scale down destination VKS workloads to zero	`vks -n "$DST_NS" scale deployment/payments-api statefulset/postgres-payments -replicas=0`	< 1 min

2	Remove the VKS PVC and PV while preserving the FCD	`vks -n "$DST_NS" wait -for=delete pod -l app=payments-api -timeout=300s`
		
		`vks get volumeattachments -o wide`
			< 3 min

3	Confirm destination pods and volume attachments are removed	`vks -n "$DST_NS" delete pvc "$PVC_NAME"`
		
		`vks delete pv "$VKS_PV_NAME"`
			< 2 min

4	Release the destination Supervisor registration	`sup -n "$SUP_NS" delete cnsregistervolume "$REGISTER_NAME"` *Remove the Supervisor PVC using the approved retain/release procedure*	< 3 min

5	Recreate and bind the source PVC to the retained source PV	`src apply -f source-pvc.yaml`
		
		`src -n "$SRC_NS" wait -for=jsonpath='{.status.phase}'=Bound pvc/"$PVC_NAME" -timeout=300s`
			< 3 min

6	Scale the source workloads back up	`src -n "$SRC_NS" scale deployment/payments-api -replicas="$SOURCE_REPLICAS"`
		
		`src -n "$SRC_NS" scale statefulset/postgres-payments -replicas="$SOURCE_STS_REPLICAS"`
			< 3 min

7	Validate source workload and data	`src -n "$SRC_NS" rollout status deployment/payments-api -timeout=300s`
		
		`src -n "$SRC_NS" rollout status statefulset/postgres-payments -timeout=300s`
		
		`curl -fsS https://payments.example.com/health | jq .`
			< 3 min

8	Record and communicate the rollback	Update the migration record and raise the relevant incident or operational notification	< 1 min

⚠ WARNING

Split-Brain Prevention: Operations teams must explicitly ensure the VKS target workloads are fully scaled down (Step 1) before remounting the FCD to the source cluster. Reactivating the source OpenShift deployment while VKS is still running will result in immediate data corruption. Total rollback RTO is designed to be < 8 minutes from failure detection.

Note:

1. Do not use `govc disk.attach` to attach the FCD directly to a Kubernetes worker. Kubernetes and the vSphere CSI driver must control the attachment lifecycle
2. Before deleting the Supervisor PVC, confirm that the backing FCD will be retained. The exact release procedure depends on the Supervisor PV reclaim policy and the deployed VKS/VCF version
3. If rollback requires reverting to the cutover snapshot, all writes made after destination activation will be discarded

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

Migrating production Kubernetes workloads to VKS on VCF 9 requires decoupling stateless cluster configurations from stateful storage assets. By leveraging Velero for API-level manifest translation alongside the `CnsRegisterVolume` API for zero-copy First Class Disk (FCD) adoption, engineering teams can bypass the data gravity constraints that traditionally dictate multi-hour downtime windows.

The four-phase methodology detailed in this paper replaces manual, ad-hoc cutovers with a deterministic, state-driven reconciliation loop. With explicit compliance gates, platform-specific resource translation, and a sub-10-minute rollback SLA, platform administrators can execute complex cluster transitions with predictable outcomes, minimized risk, and strict auditability.

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