TECHNICAL WHITE PAPER:
July 2026

Migrating Non-vSphere Kubernetes Sources to VKS Using Filesystem Replication

A Professional Services Guide for Storage-Agnostic Migration from Any Non-vSphere K8s Distribution 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 runs on any non-vSphere storage (NFS, Ceph, cloud block storage, or any other non-vSphere CSI driver) — no data-copy mechanism in this series avoids moving bytes in that case, and this paper's rsync-based approach is the most storage-agnostic way to do it. For a source that is VKS itself, see Migrating Workloads Between VKS Clusters Using Cross-vCenter vMotion. For a source already running on vSphere CSI, see Migrating vSphere-Based Kubernetes Workloads to VKS Using Zero-Copy FCD Adoption for a zero-copy alternative. For the fully validated, benchmarked reference methodology (works for any source, but always copies data via S3), see Migrating Kubernetes Workloads to VKS Using Velero and S3-Compatible Storage.

This white paper presents a structured framework for migrating workloads from heterogeneous Kubernetes environments to VMware vSphere Kubernetes Service (VKS) on VMware Cloud Foundation 9.

The methodology separates migration into two paths. Velero transfers Kubernetes manifests and metadata while excluding storage objects. pv-migrate copies filesystem data from source PVCs to pre-provisioned VKS volumes using rsync over SSH. Because the transfer operates at the filesystem level, over the network, the clusters do not need to share a storage platform, CSI driver or Kubernetes distribution.

Controlled quiescence, validation gates and retention of the source PVC provide a clear cutover and rollback boundary. The result is a repeatable approach for migrating stateless and stateful workloads to VKS, with migration time determined primarily by dataset size, file count and available network bandwidth.

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.

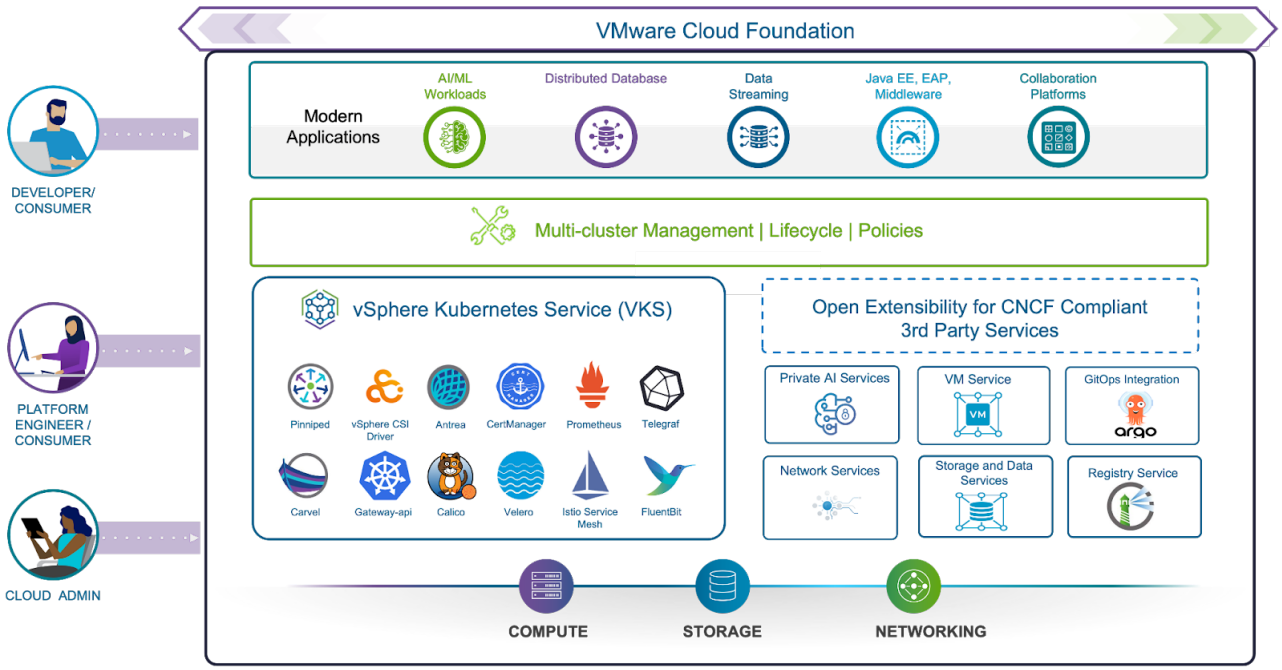
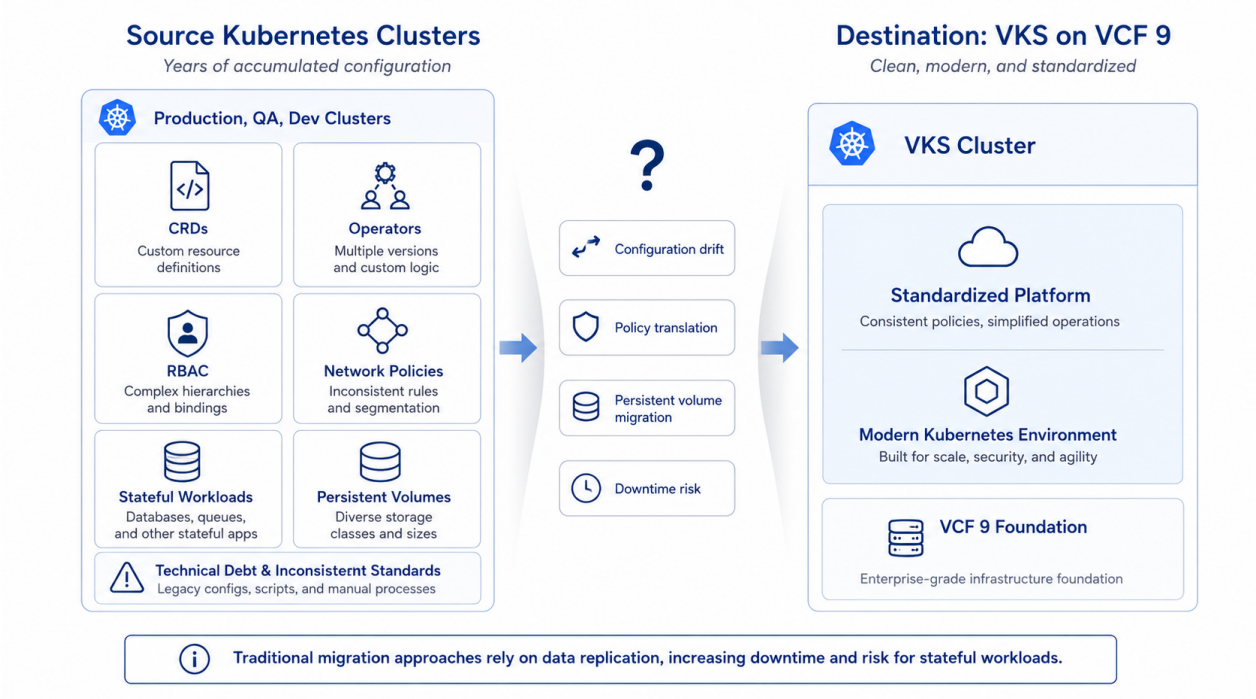


Figure 1: VMware Cloud Foundation with vSphere Kubernetes Service

The Migration Challenge

Customers operating Kubernetes workloads on Bare-Metal 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

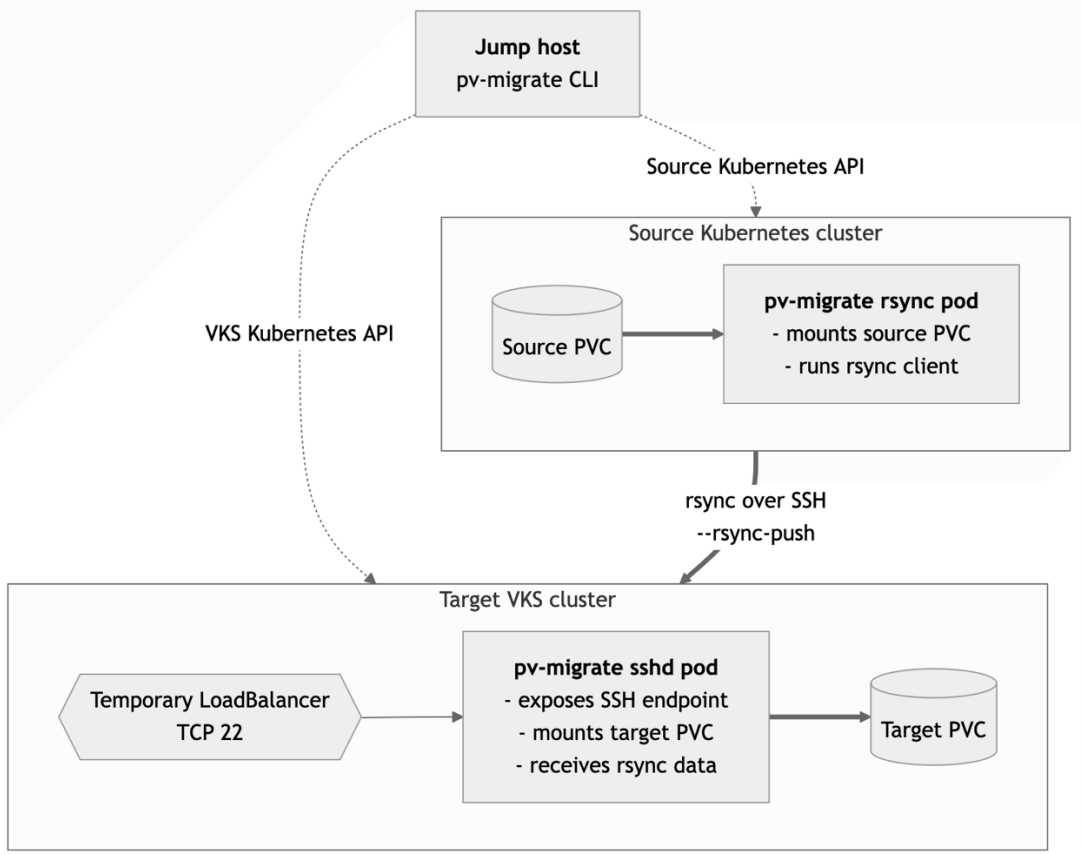
Solution Architecture

Persistent Storage Migration using rsync

Rsync is a fast, POSIX-compliant file synchronization utility that leverages a delta-transfer algorithm to minimize network bandwidth consumption by transmitting only the differential changes between source and destination datasets. Here, we leverage pv-migrate, which utilizes rsync over an SSH tunnel to perform a filesystem-level copy between an existing source PVC and a pre-provisioned VKS PVC.

The pv-migrate CLI runs from a jump host with API access to both clusters, but application data does not pass through the jump host.

Using the rsync-push method, `pv-migrate` creates an rsync Job in the source cluster and a temporary SSHD deployment and loadBalancer Service on VKS. The source Job mounts the source PVC and initiates an encrypted SSH connection to the VKS loadBalancer address. The SSHD Deployment writes the received data directly to the mounted destination PVC.



The first execution copies the complete filesystem. Subsequent executions compare the source and destination and transfer only changed data, allowing an optional pre-copy followed by a final synchronisation after the application has been quiesced. SSH keys and mover resources are generated for the migration and cleaned up after completion.

Note that this is an rsync session transported over SSH, rather than an SSH port-forward tunnel. The SSH connection provides encryption and key-based authentication, while rsync supplies file comparison, metadata preservation and incremental transfer; see https://github.com/utkuozdemir/pv-migrate/blob/main/docs/migrate.md

The source storage implementation is not relevant. Just as long as the source PVC is a filesystem-based volume that the migration Job can mount and read. Note that root-owned data may require an approved source-side security context. The final transfer must occur after the application has stopped writing.

Migration Architecture Overview

A successful workload migration must transfer both persistent application data and Kubernetes configuration. The architecture therefore separates the migration into two paths:

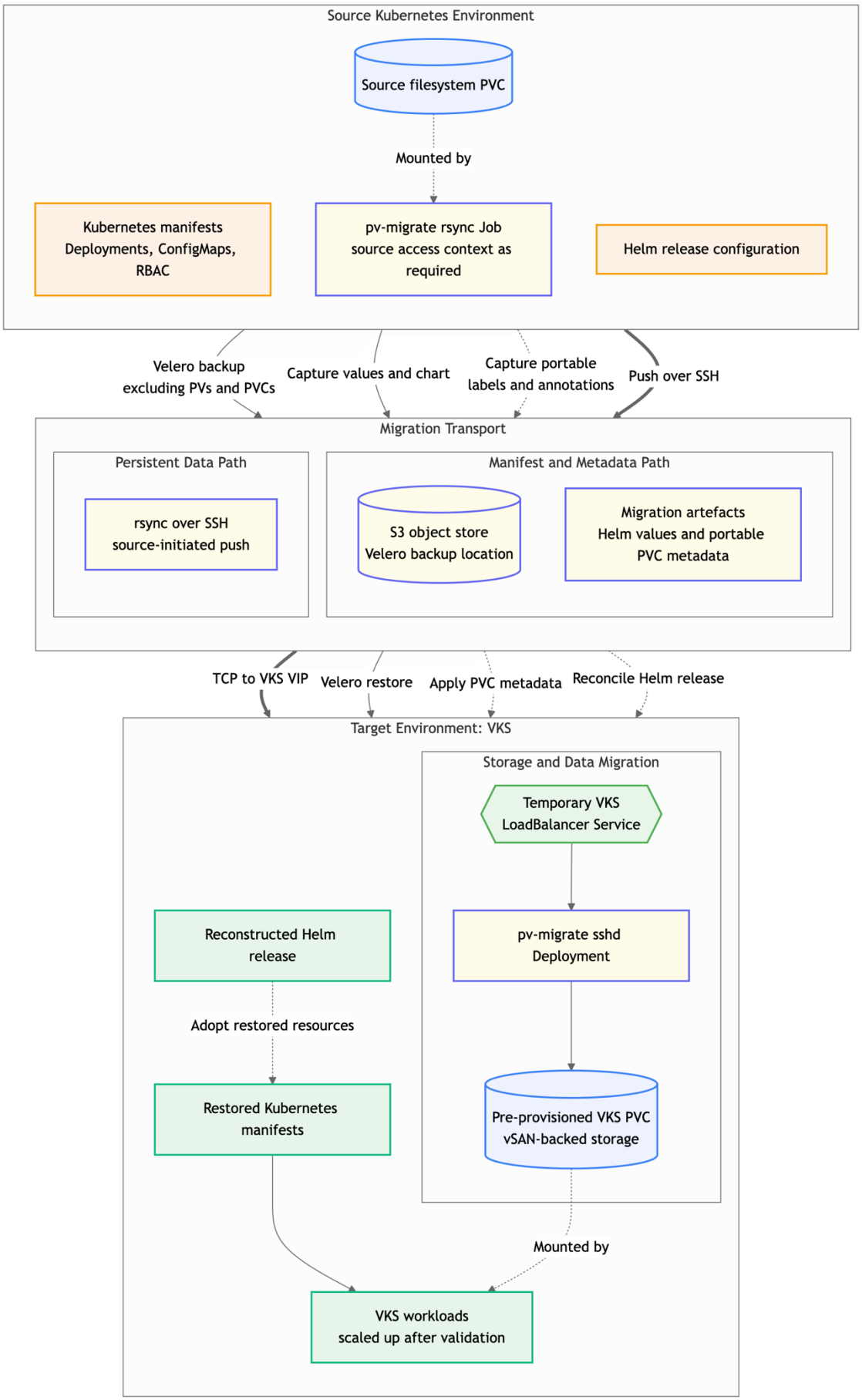
- pv-migrate copies filesystem data directly from the source PVC to a pre-created VKS PVC
- Velero captures namespace-scoped Kubernetes resources to an S3-compatible endpoint

Storage objects are excluded from Velero because the destination PVC is created separately and populated by pv-migrate:

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

Velero supports filtering backups by namespace and resource type, allowing the storage and Kubernetes object paths to remain independent. Velero resource-filtering documentation

Helm-managed applications require additional handling. The release name, chart version and supplied values must be captured before cutover, and the exact chart package must remain available. Scaling a workload to zero changes the live Kubernetes objects but does not necessarily change the desired state held by Helm, GitOps or an operator. These controllers must therefore remain suspended or use migration-specific replica overrides until the copied data has been validated.


Source Kubernetes Environment
Source filesystem PVC
Mounted by
Kubernetes manifests
Deployments, ConfigMaps,
RBAC
pv-migrate rsync Job
source access context as
required
Helm release configuration
Velero backup
excluding PVs and PVCs
Capture values and chart
Capture portable
labels and annotations
Push over SSH
Migration Transport
Persistent Data Path
rsync over SSH
source-initiated push
Manifest and Metadata Path
S3 object store
Velero backup location
Migration artefacts
Helm values and portable
PVC metadata
TCP to VKS VIP
Velero restore
Apply PVC metadata
Reconcile Helm release
Target Environment: VKS
Storage and Data Migration
Temporary VKS
LoadBalancer Service
Reconstructed Helm
release
pv-migrate sshd
Deployment
Adopt restored resources
Restored Kubernetes
manifests
Pre-provisioned VKS PVC
vSAN-backed storage
Mounted by
VKS workloads
scaled up after validation

The two migration paths converge on the VKS guest cluster:

Migration Concern	Migration Tooling	VKS Destination
Kubernetes API objects	Velero and S3-compatible storage	Velero backup → restore
Persistent filesystem data	pv-migrate using rsync over SSH	Pre-created VKS PVC
Helm chart and values	Helm repository/OCI registry and exported values	Reconciled Helm release
Application-owned PVC metadata	Exported JSON or YAML	Labels and annotations applied to the VKS PVC

Because PVs and PVCs are excluded from the Velero restore, portable application-owned labels and annotations must be captured from the source claim and applied to the destination PVC. Kubernetes-generated provisioning, binding and lifecycle metadata must not be copied.

Load Balancer and Network Requirements

- The LoadBalancer Service is created on the **destination** (VKS)
- The configured VKS load-balancer implementation allocates a temporary service address
- The source workload network must be able to route to the allocated VKS address
- Firewalls and NetworkPolicies must permit TCP 22 from the source migration pod to the VKS service address
- No inbound service or LoadBalancer is required on the source cluster
- The jump host requires API connectivity to both clusters, but it is not in the bulk-data path

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	pv-migrate	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 uses four phases - Discover, Analyze, Blueprint and Execute - followed by post-migration validation and rollback controls. Persistent data and Kubernetes resources follow separate migration paths before being brought together on the destination VKS cluster.

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, pv-migrate, Helm	Migration Report
Post migration Monitoring	Real-time validation, auto rollback triggers	Prometheus, kubectl, alert rules	Post-migration Report

Tooling

The VKS Migration Methodology uses four phases - Discover, Analyze, Blueprint and Execute - followed by post-migration validation and rollback controls. Persistent data and Kubernetes resources follow separate migration paths before being brought together on the destination VKS cluster.

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.

pv-migrate	Copies filesystem data between PVCs. It creates temporary rsync and SSHD workloads, generates an ephemeral SSH key pair and transfers data directly from the source cluster to a pre-created VKS PVC.
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)	Provides additional vSphere inventory and storage validation where the source environment is also running on vSphere. It is not required for the generic data-copy path.

Tool and container-image versions should be pinned and approved before execution. The pv-migrate mover images must be accessible from both clusters and should be mirrored into an approved internal registry where external image access is restricted. Velero must be installed on both clusters, and both installations must be able to access the selected S3 endpoint.

The migration identity requires sufficient namespace-scoped permissions to create and monitor temporary Jobs, Deployments, Services, Secrets and ServiceAccounts. Additional source-side permissions may be required where the migration pod must read root-owned filesystem data. The VKS LoadBalancer address must also be reachable from the source workload network on TCP 22.

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
```

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.

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 prepares the destination storage on VKS, quiesces the selected source workloads, copies the persistent filesystem data with pv-migrate, restores the Kubernetes manifests with Velero and then activates the workloads on VKS.

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

The migration uses two independent paths:

- **Kubernetes object path:** Velero transports namespace-scoped application manifests and metadata while excluding PVs, PVCs and storage snapshots.
- **Persistent data path:** pv-migrate mounts the source and destination PVCs in temporary mover workloads and transfers their filesystem contents using rsync over SSH.

Note that the source and destination volumes are independent storage objects. They do not need to share a vCenter, CNS domain, storage array, CSI driver or Kubernetes distribution. The original source PVC remains in place to support rollback until the migration wave has been accepted.

Scope and assumptions

This simplified example supports the following baseline case:

- A migration jump host has authenticated API access to both clusters
- The source migration pod can establish a TCP connection to the temporary VKS LoadBalancer address
- Velero is installed on both clusters and can access the same BackupStorageLocation
- The application supports a controlled quiescence period
- No destination workload writes to the target PVC while the copy is in progress
- Source and destination workloads are never simultaneously promoted as active writers

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

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

export SRC_CONTEXT="source-context"
export VKS_CONTEXT="migration-vks:migration-vks"

export SRC_NS="prod-payments"
export DST_NS="prod-payments"
export PVC_NAME="payments-db"
export WORKLOAD_NAME="payments-api"

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

export TARGET_STORAGE_CLASS="<vks-storage-class>"
export TARGET_CAPACITY="1Ti"

src() {
  kubectl \
    --kubeconfig "$SRC_KUBECONFIG" \
    --context "$SRC_CONTEXT" "$@"
}
```

```
vks() {
  kubectl \
    --kubeconfig "$VKS_KUBECONFIG" \
    --context "$VKS_CONTEXT" "$@"
}

src cluster-info
vks cluster-info
pv-migrate version

✓ Source Kubernetes, and destination VKS 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: Capture Source State and Metadata

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 3: Create the Destination VKS 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

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

vks -n "$DST_NS" wait \
  --for=jsonpath='{.status.phase}'=Bound \
  "pvc/$PVC_NAME" \
  --timeout=300s
```

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 created
```

Stage 4: Quiesce and Copy the Data

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
```

Wait until all source pods using the claim have terminated and the filesystem is no longer being modified.

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

✓ Workload quiesced, snapshot ready, manifest backup completed
```

Run the data copy:

```
pv-migrate \
  --source-kubeconfig "$SRC_KUBECONFIG" \
  --source-context "$SRC_CONTEXT" \
  --source-namespace "$SRC_NS" \
  --source "$PVC_NAME" \
  --dest-kubeconfig "$VKS_KUBECONFIG" \
  --dest-context "$VKS_CONTEXT" \
  --dest-namespace "$DST_NS" \
  --dest "$PVC_NAME" \
  --rsync-push \
  --strategies loadbalancer \
  --dest-delete-extraneous-files \
  --helm-timeout 10m

✓ Workload quiesced, manifest backup complete and data migrated
```

Stage 5: Apply PVC Metadata

Apply the previously captured portable labels and annotations to the destination PVC:

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

✓ PVC metadata applied
```

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
```

Post-Execution Monitoring 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	Fence the destination by scaling every VKS writer to zero	`vks -n "$DST_NS" scale deployment/payments-api statefulset/postgres-payments --replicas=0`	< 1 min
2	Confirm no VKS pod still consumes the destination PVC	`test "$(vks -n "$DST_NS" get pods -o json | jq --arg pvc "$PVC_NAME" '[.items[] | select(any(.spec.volumes[]?; .persistentVolumeClaim.claimName == $pvc))] | length')" -eq 0`	< 3 min
3	Confirm the VKS volume is detached	`export TARGET_PV=$( vks -n "$DST_NS" get pvc "$PVC_NAME" \ -o jsonpath='{.spec.volumeName}' )` `while [ "$( vks get volumeattachments.storage.k8s.io -o json | jq --arg pv "$TARGET_PV" \ '[.items[] | select(.spec.source.persistentVolumeName == $pv) ] | length' )" -ne 0 ]; do echo "Waiting for $TARGET_PV to detach..." sleep 5 done` `echo "No VolumeAttachment remains for $TARGET_PV"`	< 5 min
4	Protect the destination recovery copy	`vks patch pv "$TARGET_PV" \ --type=merge \ -p '{"spec":{"persistentVolumeReclaimPolicy":"Retain"}}'` `vks -n "$DST_NS" annotate pvc "$PVC_NAME" \ migration.vmware.com/rollback-hold="true" \ migration.vmware.com/rollback-time="$(date -u +%FT%TZ)" \ --overwrite`	< 1 min
5	Establish whether VKS accepted post-cutover writes	Compare an app specific transaction or write watermark with the cutover watermark. If they differ, stop the auto rollback	App specific

6	Confirm that the original source PVC remains bound	`src -n “$SRC_NS” scale deployment/payments-api –replicas=”$SOURCE_REPLICAS”` `src -n “$SRC_NS” scale statefulset/postgres-payments –replicas=”$SOURCE_STS_REPLICAS”`	< 1 min
7	Reactivate the source workloads using their recorded replica counts	`src -n "$SRC_NS" scale deployment/payments-api \` `  --replicas="$SOURCE_DEPLOY_REPLICAS"` `src -n "$SRC_NS" scale statefulset/postgres-payments \` `  --replicas="$SOURCE_STS_REPLICAS"` `src -n "$SRC_NS" rollout status deployment/payments-api \` `  --timeout=300s` `src -n "$SRC_NS" rollout status statefulset/postgres-payments \` `  --timeout=300s` `SOURCE_APP_HEALTH=$(` `  curl -fsS https://payments-source.example.com/health \|` `  jq -r '.status'` `)` `test "$SOURCE_APP_HEALTH" = "healthy"` `printf 'Deployment payments-api restored to %s replicas\n' \` `  "$SOURCE_DEPLOY_REPLICAS"` `printf 'StatefulSet postgres-payments restored to %s replicas\n' \` `  "$SOURCE_STS_REPLICAS"` `printf 'Source application health: %s\n' "$SOURCE_APP_HEALTH"`	< 5 min
8	Validate the source application and restore traffic	Check rollouts and application health, then invoke the approved DNS/LB/route swtich	App specific
9	Record the rollback and retain the VKS PVC	Annotate the target PVC and update the migration / incident record	< 1 min

⚠ WARNING

Split-Brain Prevention: The source and VKS workloads use separate copies of the data. Activating both as writers will not corrupt a shared disk, but it will create divergent application state. Only one environment may be promoted as the authoritative writer. If VKS has accepted writes, restarting the original source without reconciliation may lose those writes.

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 Kubernetes resources, package-manager state and persistent data to be handled as related but distinct concerns. This methodology uses Velero to transport namespace-scoped resources, pv-migrate to copy filesystem data between PVCs, and Helm to reconcile packaged applications on the destination. Because the data transfer operates through mounted filesystems, the source does not need to share the Kubernetes distribution, CSI driver or storage platform used by VKS.

The four-phase methodology replaces ad-hoc cutovers with controlled migration waves incorporating discovery, resource translation, workload quiescence, validation and rollback controls. Retaining the original source PVC until the VKS workload has been validated provides a clear rollback boundary, while explicit data-authority controls prevent divergent writes after promotion. The result is a repeatable and auditable approach for migrating stateful and stateless applications to VKS.

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