TECHNICAL WHITE PAPER:
August 2026

Migrating Kubernetes Workloads to VKS Using Velero and S3-Compatible Storage

A Professional Service Guide for Migrating Application Objects and Data from any Kubernetes source — with a Worked Example: OpenShift on Bare Metal to VKS Using OADP and MinIO

Table of Contents

List of Tables

Executive Summary

This paper is one of four in the VKS migration series. Use this paper if you want the validated, benchmarked Velero+S3 methodology -- it works for any source but is the slowest of the four since it always performs a full data copy. For a source that is VKS itself, see Migrating Workloads Between VKS Clusters Using Cross-vCenter vMotion. For a source not running vSphere CSI, see Migrating Non-vSphere Kubernetes Workloads to VKS Using Filesystem Replication for a lighter-weight rsync-based alternative. For a source already on vSphere CSI, see Migrating vSphere-Based Kubernetes Workloads to VKS Using Zero-Copy FCD Adoption for a zero-copy path that avoids the data copy entirely.

Moving Kubernetes workloads to vSphere Kubernetes Service (VKS) on VMware Cloud Foundation 9 presents two problems, whatever the source platform. The first is getting the data across: persistent volumes must physically move from the source storage backend to VKS-provisioned storage. The second is object fidelity: source platforms carry distribution-specific API objects that have no equivalent on VKS, so a plain backup and restore will not produce a working application on the target. This paper documents a general-purpose, storage-agnostic method: it applies to any Kubernetes source and requires only that both clusters reach a common S3-compatible store, but -- unlike the zero-copy path in Paper 4 -- it always performs a full data copy through that store.

This paper is organized into two parts.

Part 1 describes a platform-agnostic methodology built on Velero and any S3-compatible object store: the CSI snapshot Data Mover carries persistent-volume contents through the store onto freshly provisioned target volumes, while every namespace-scoped object is classified before migration -- it moves unchanged, is converted to its VKS equivalent, or is excluded. The classification scripts are tested against recorded expected verdicts, so a wrong verdict surfaces as a tooling defect before cutover, not as a broken application after it. Velero may run on the source either as the upstream distribution or as a vendor-curated packaging supplied with the source Kubernetes distribution -- such as OpenShift API for Data Protection (OADP) on Red Hat OpenShift; the VKS target always runs native Velero.

Part 2 applies the methodology end-to-end in a worked example: migrating workloads from Red Hat OpenShift Container Platform on bare metal to VKS, using OADP as the source-side Velero and MinIO as the shared backup store. Two test scenarios validate the object path -- OBJ-XFORM-01 (Route to Ingress, DeploymentConfig to Deployment, SCC to Pod Security Standards, with deliberate failure traps the tooling must catch) and OBJ-CRD-02 (operator-backed CustomResources requiring install-CRD-first ordering). On the data plane, validation moved roughly 5.9 TiB of user data across five scenarios -- up to ten PVCs in parallel and 2.02 TiB in a single batch -- with zero data-integrity errors.

NOTE

The example manifests, transform overlays, expected-verdict files, and scanner scripts referenced throughout this paper accompany the validated-solutions repository: https://github.com/vmware/vks-validated-solutions/tree/main/VKS-Migrations

Part 1: Migration Methodology

Part 1 is platform-agnostic. Nothing in it depends on the source distribution, the specific S3-compatible store, or how Velero is packaged; those specifics are the subject of the worked example in Part 2.

Introduction

vSphere Kubernetes Service

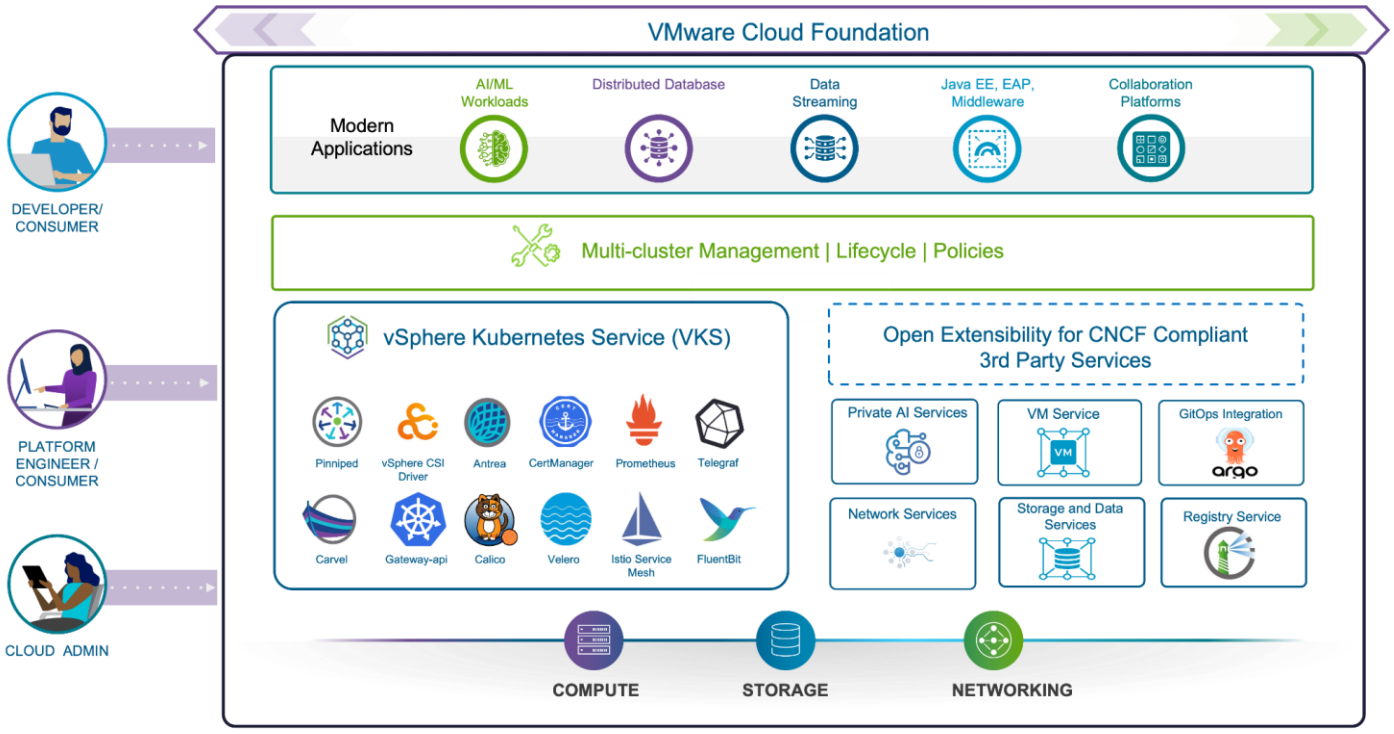
vSphere Kubernetes Service (VKS) is the Kubernetes runtime built directly into VMware Cloud Foundation (VCF). With CNCF certified Kubernetes, VKS enables platform engineers to deploy and manage Kubernetes clusters while leveraging a comprehensive set of cloud services in VCF. Cloud admins benefit from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

Benefits of using VKS

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

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

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



The Two Migration Problems

The data-plane problem. Source and target are different clusters on different storage backends with no shared storage, so an in-cluster snapshot alone is not portable. Persistent-volume contents must be physically carried from the source backend to VKS-provisioned storage.

The object-fidelity problem. When a source namespace is restored onto VKS unchanged, its distribution-specific objects either fail to apply, because their APIs are absent on the target, or apply but never reconcile, because the controllers that own them do not exist on VKS. An ingress-type object with no controller to serve it, a deployment variant with no controller to roll it out, a RoleBinding that references a source-only ClusterRole, a build artifact pointing at an internal registry that was never migrated: each is a silent failure, where the restore reports success while the workload is subtly broken.

The migration therefore cannot be a single undifferentiated backup and restore. Every object must be assigned a migration fate before any batch is executed, and the tooling that assigns those fates must itself be trustworthy, because a misclassification is precisely the kind of error that passes review and surfaces only in production.

Solution Architecture

Velero and the S3-Compatible Backup Store

Velero is the single engine for both planes. On the source cluster it backs up namespace-scoped objects and through the CSI snapshot Data Mover — the contents of every persistent volume, into a shared S3-compatible BackupStorageLocation (BSL). On the target, native Velero restores the objects, re-provisions the volumes through the vSphere CSI driver, and downloads the data. The architecture requires only an S3-compatible endpoint; the store must be pre-created, resolvable, and reachable under the same URL from both clusters, with its TLS chain trusted by both. Store-specific configuration and failure modes are catalogued for MinIO in the Part 2 appendix.

Vendor-Curated Velero Distributions

Velero is an upstream open-source project, and several Kubernetes vendors curate and support their own packaging of it for their distribution — on Red Hat OpenShift, for example, it ships as the OpenShift API for Data Protection (OADP) operator. This gives two ways to run the engine on the source cluster:

- **Approach A — vendor-curated (operator-managed):** the distribution's operator packages Velero, the CSI integration, and the Data Mover behind a single custom resource, keeps the components version-matched, and upgrades them as a unit under vendor support.
- **Approach B — upstream Velero:** velero install places the same engine on the source directly; plugins, BSL, and node agent are assembled and versioned manually. This is the only option on distributions with no curated operator.

The engine, the backup format, and the CLI are identical in both; what differs is packaging, lifecycle, and support. Backup artifacts are interchangeable between approaches, and the classification and transform layer is engine-agnostic throughout. In every case the VKS target runs native upstream Velero, so both approaches converge on the same restore path. A measured comparison of the two approaches on OpenShift is provided in Part 2.

The Data Plane: CSI Snapshot Data Mover

At backup time each PVC is snapshotted through the CSI driver, and the snapshot contents are read by the node agent's embedded Kopia engine and uploaded to the S3 BSL. Kopia is the open-source backup engine embedded in Velero's node agent: it splits file data into content-addressed blocks, deduplicates, compresses, and encrypts them, stores identical blocks only once, and on later backups uploads only the blocks that changed. At restore time the VKS cluster provisions a fresh volume through the vSphere CSI driver and the target node agent downloads the data into it. For volumes whose driver lacks CSI snapshot support, Velero's file-system backup provides the same movement at the file level.

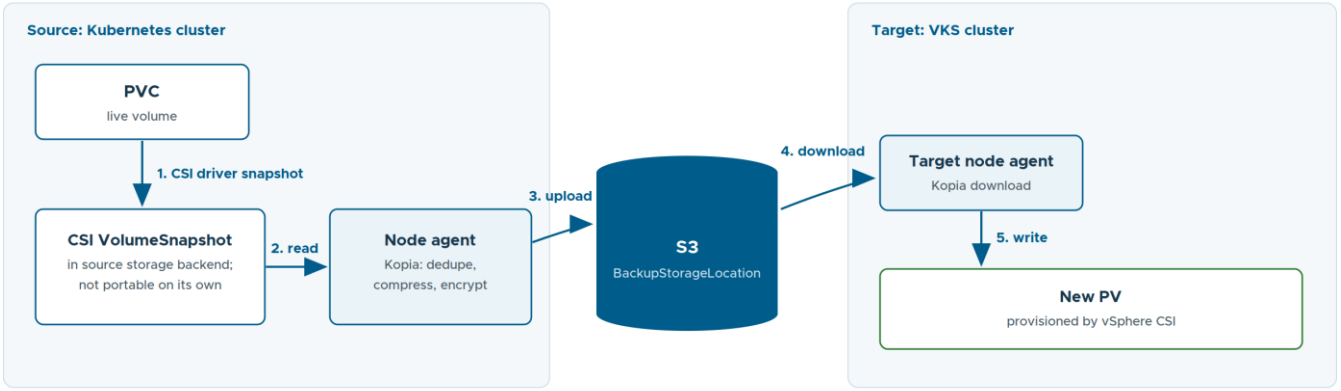


Figure 1: CSI snapshot Data Mover flow — snapshot, Kopia upload, S3, download, re-provisioned PV.

The Object Plane: Classification Model

Every namespace-scoped object is assigned exactly one of three verdicts. The verdict determines how the object is treated during the batch and what evidence is required to accept it.

Table 1. The Object Plane: Classification Model

Verdict	Meaning	Typical objects	Handling
MIGRATE_AS_IS	Portable object with a direct VKS equivalent and no platform coupling.	Deployments, Services, ConfigMaps, Secrets, ServiceAccounts, PVCs, NetworkPolicies.	Backed up and restored unchanged via Velero.
REVIEW_REQUIRED	Applies on VKS but carries semantics needing confirmation or conversion.	Distribution-specific ingress and deployment variants; security-policy bindings; RoleBindings referencing distribution-only roles.	Converted via the target-side transform overlay; orphan-checked postflight.
BLOCK_MIGRATION	Cannot land correctly without a prerequisite.	Operator-backed CRDs/CRs absent on the target; in-cluster build artifacts with no runtime meaning on VKS.	Prerequisite installed first, or object excluded entirely.

These verdicts are produced by two report-only scripts — a preflight object scanner run against the source and a postflight object scanner run against the target after restore. Both classify and record but never mutate cluster state. The transform overlay — the set of target-side conversion manifests that re-express distribution-specific objects as VKS-native constructs — is the third deliverable; all three are published in the accompanying repository and exercised in Part 2.

⛔ IMPORTANT

Script verdicts are validated, not trusted. Each scenario records the expected classification for every seeded object before the run. Any divergence between those expected verdicts and the script output is logged as a defect against the scanner — a tooling defect in a dedicated backlog, not a migration failure — keeping tooling accuracy measurable and separate from migration outcome.

Objects and data run through the same engine and the same store, in the same batch: the classification model governs the objects; the Data Mover carries the state.

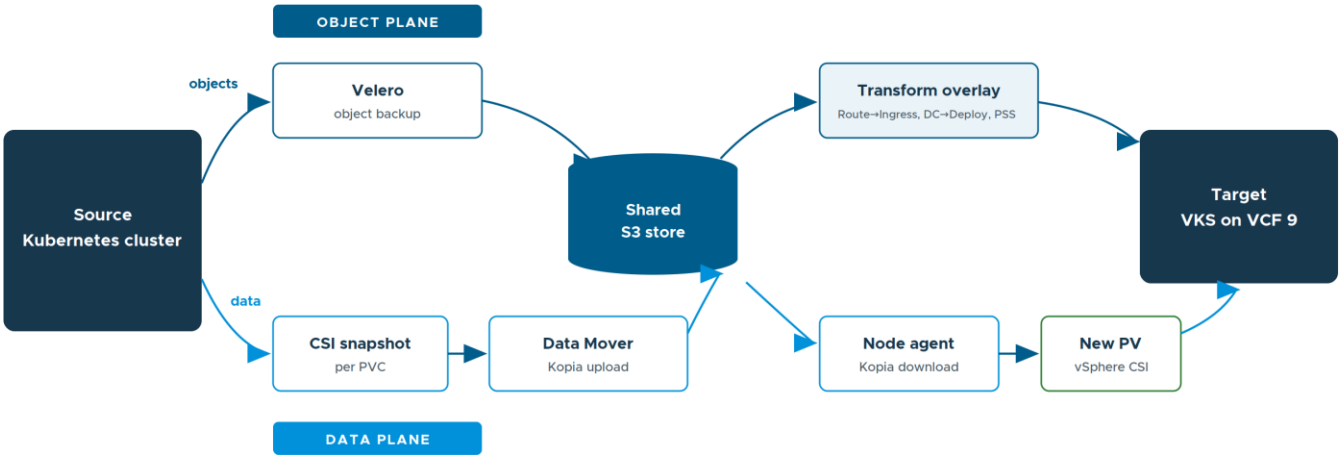


Figure 2: Combined migration architecture — object plane and data plane share one engine, one S3 store, one batch.

The Five-Phase Methodology

The object-fidelity methodology overlays classification and transformation gates onto a five-phase framework. Each planning phase produces a gate artifact that the next phase consumes.

Table 2. The Five-Phase Methodology

Phase	Objective	Primary tooling	Gate artifact
1: Discovery	Object inventory, CRD catalog, platform-API detection	kubectl, preflight scanner	Discovery Report
2: Analyze	Classification, transform-overlay authoring, expected-verdict definition	Preflight scanner, overlay manifests	Verdict Matrix + Overlay
3: Blueprint	Batch plan, prerequisite ordering, approvals	Batch planner	Signed Batch Plan
4: Execute	Velero backup, transform, restore to VKS	Velero, kubectl	Migration Report
5: Monitor	Postflight scan, orphan detection, acceptance gate	Postflight scanner, kubectl	Postflight Report

Phases 1 to 3 are planning phases. **Discovery** inventories every namespace-scoped object, catalogs CRDs and their served versions, and detects distribution-specific API groups. **Analyze** assigns each object its verdict, authors the transform overlay for the REVIEW_REQUIRED set, and records the expected-verdict table that the scanners will be measured against. **Blueprint** orders the batches, sequences prerequisites (operators and CRDs before their CRs, webhook backends before the objects they admit), and captures approvals. The outputs are the gate artifacts above; Execute and Monitor consume them.

Phase 4 — Execute, in Five Stages

Execution runs in controlled batches. Each batch proceeds through five stages; the full command-level walkthrough is in Part 2.

Stage 0 — Environment and engine setup. Velero installed on the source (Approach A or B) with CSI support and the node agent enabled; native Velero on the target with its node agent; the shared BSL verified Available from both clusters; timeouts raised for large batches.

Stage 1 — Preflight object scan (report-only). The scanner classifies every object in the source namespace; output is compared to the expected verdicts and divergences logged as scanner defects.

Stage 2 — Backup (objects and data). Velero backs up the namespace with snapshot data movement enabled; BLOCK_MIGRATION build artifacts are excluded; the stage completes only when every DataUpload reports Completed.

Stage 3 — Prepare the target namespace. The destination namespace is created, its security-enforcement label applied, and the transform overlay bound to the target environment and applied — before the restore, so admission enforcement is active when workload pods arrive.

Stage 4 — Restore onto VKS. Velero restores with namespace mapping and a storage-class map, excluding the source originals of transformed objects; the stage completes only when every DataDownload reports Completed.

Phase 5 — Monitor and Accept

The postflight scanner confirms transforms applied, excluded objects absent, operator-backed CRs reconciled, and no orphaned references remaining. A batch is accepted only when the postflight report matches expectations across every object; the acceptance decision is recorded in an evidence package alongside the preflight report, postflight report, verdict comparison, and defect log.

Operational Principles

Five principles distilled from validation govern every batch. The Part 2 appendix expands each into concrete symptoms and resolutions.

- **Verify the store from both sides.** A BSL that shows Available on the source proves nothing about the target. The Stage 0 acceptance criterion is BSL Available on the source and on the VKS cluster, against the same endpoint URL.
- **The data CRs are the source of truth.** Backup and restore describe summarize object results; persistent-volume contents are tracked separately in DataUpload and DataDownload custom resources. A batch is not done — and must not be accepted — until every one of them reports Completed. Gate every stage on the data CRs, never on the Backup/Restore phase alone.
- **PartiallyFailed is the normal terminal state.** A migration restore into VKS will almost always report PartiallyFailed — excluded platform-specific resources, skipped existing objects, and benign warnings all land there. The restore phase is not an acceptance signal in either direction; only the postflight scan and the acceptance checklist decide a batch.
- **Raise timeouts and TTLs before large batches.** Default prepare and resource timeouts (30 minutes) fail 1 TiB volumes under parallel load; both were raised to 4 h in validation. Set the backup TTL explicitly to cover the full migration window, including rollback reserves — the 30-day default silently garbage-collects backups mid-project.
- **The source is never mutated — until cutover.** Backups snapshot volumes without detaching them, scans are report-only, and the migration writes only to the target, so rolling back a batch means deleting the restored namespace and volumes on VKS while the source remains authoritative. The moment of commitment is cutover; after the data download completes, the volume exists on both clusters, so quiesce the source (or run a delta batch) before the final cutover and never serve traffic from both sides at once.

Part 2: Worked Example — OpenShift on Bare Metal to VKS with OADP and MinIO

Part 2 applies the methodology to a concrete pairing: Red Hat OpenShift Container Platform (OCP) on bare metal as the source, VKS on VCF 9 as the target, OADP as the vendor-curated Velero on the source, and MinIO as the shared S3-compatible store. Teams migrating from another distribution replace only this part.

Table 3. Example OpenShift-to-VKS Migration Environment

Component	Value
Source platform	OpenShift Container Platform on bare metal (four-worker cluster)
Source storage	CephFS on OpenShift Data Foundation (ODF), 3× replication, 23 TiB raw
Source Velero	OADP 1.5.5 operator (Velero 1.18), CSI plugin + Data Mover enabled
Target platform	VKS guest cluster on VMware Cloud Foundation 9
Target storage	vSphere CSI driver on vSAN ESA
Target Velero	Native upstream Velero 1.18, node agent enabled
Backup store	MinIO, 8 TiB volume, shared BackupStorageLocation
Network	10 GbE between clusters and store

On OCP, Approach A is the OpenShift API for Data Protection (OADP) — the Red Hat-supported operator that installs and manages Velero on the source cluster through a single DataProtectionApplication (DPA) custom resource declaring the Velero deployment, the BSL, the enabled plugins, and the node agent that carries the CSI snapshot Data Mover. Wherever this paper refers to the operator's custom resource — client limits, timeouts, image-backup settings — those values belong in the DPA. Approach B, upstream Velero installed directly, remains available on OCP and is shown as a Stage 0 variant.

OpenShift Object Mappings

The distribution-specific objects on an OCP source are Routes, DeploymentConfigs, SecurityContextConstraints (SCCs), ImageStreams, BuildConfigs, and OLM-managed CustomResourceDefinitions. Source-side API interaction uses the oc CLI, interchangeable with kubectl for every command in this paper. The classification of a representative production namespace set, discovered during Phase 1:

Table 4. OpenShift Object Mapping and Migration Actions

Object category	Verdict	Migration action	Risk
Deployments / Services / ConfigMaps	MIGRATE_AS_IS	Velero backup → restore, unchanged	Low
Secrets / ServiceAccounts	MIGRATE_AS_IS	Restored; legacy service-account-token Secrets excluded (target mints tokens)	Medium
PVCs and persistent-volume contents	MIGRATE_AS_IS	CSI snapshot → Data Mover → re-provision + download	High

Object category	Verdict	Migration action	Risk
Routes (edge / passthrough / reencrypt)	REVIEW_REQUIRED	Transform → Ingress + TLS Secret	Medium
DeploymentConfigs	REVIEW_REQUIRED	Transform → Deployment	Medium
SCC anyuid bindings	REVIEW_REQUIRED	Transform → PSS baseline namespace label	High
ImageStreams / BuildConfigs	BLOCK_MIGRATION	Exclude; source images externally	Medium
Operator CRDs / CRs (absent on target)	BLOCK_MIGRATION	Install CRD + operator first, then restore CR	High
RoleBindings → source-only ClusterRoles	REVIEW_REQUIRED	Postflight orphan detection	High

Transform Mappings

Three transforms are applied on the target side through the transform overlay. The overlay carries two open parameters bound to the target environment at execution time: the ingress class and the apps domain.

- **Route → Ingress:** each Route becomes an Ingress with an equivalent host rule and a TLS Secret. The overlay supplies the target ingress class (<target-ingress-class>) and rewrites the host to the target apps domain (<target-apps-domain>). A Route can alternatively be translated to a Gateway API HTTPRoute — Contour on VKS reconciles Gateway API resources, and Gateway API is the successor to the Ingress API. This example standardizes on Ingress because it is the validated path; the same classification and overlay pattern applies unchanged to an HTTPRoute target.
- **DeploymentConfig → Deployment:** the DeploymentConfig is re-expressed as a standard Deployment. Image-change and config-change trigger behavior is reconstructed explicitly, since the DeploymentConfig controller does not exist on VKS.
- **SCC anyuid → PSS baseline label:** OpenShift assigns each namespace a random, non-root UID range by default, which breaks container images that expect a fixed — often root — user; such workloads are granted the anyuid SCC on the source. Kubernetes on VKS has no per-namespace UID assignment, so the same images run without special dispensation, and the anyuid dependency is re-expressed as a Pod Security Standards enforcement level applied as a namespace label rather than a cluster-scoped SCC binding. The correct level for an anyuid workload is baseline — baseline permits running as root while still blocking privileged escalation, whereas restricted requires runAsNonRoot and would reject the very workloads that needed anyuid. Enforce restricted only after the images have been remediated to run as non-root.

Execute Walkthrough (Phase 4)

Each batch scans the source, backs up namespace objects and data with Velero, prepares the target namespace, restores onto VKS, and scans the target. The walkthrough uses Approach A (OADP on the source, the validated recommendation) and a single namespace; the Approach B variant differs only in Stage 0 and is shown there.

Stage 0: Environment and Engine Setup

Two Kubernetes API endpoints are used: the source OCP cluster and the destination VKS guest cluster. OADP must be ready on the source with the CSI plugin and node agent enabled, and native Velero on the target with its node agent, both reaching the shared BSL — configured per the MinIO appendix (path-style, region, publicUrl, CA trust, checksum) and verified Available from both clusters.

```
## STAGE 0 --- Environment and OADP Setup
export SRC_KUBECONFIG="$HOME/.kube/ocp-config"
export VKS_KUBECONFIG="$HOME/.kube/vks-config"
export SRC_NS="xform-demo"  DST_NS="xform-demo"
export BACKUP_NAME="batch1-xform-demo"
export RESTORE_NAME="batch1-xform-demo-restore"
src() { oc --kubeconfig "$SRC_KUBECONFIG" "$@"; }
vks() { kubectl --kubeconfig "$VKS_KUBECONFIG" "$@"; }

# Confirm OADP DPA ready with CSI + Data Mover enabled
src -n openshift-adp get dataprotectionapplication
src -n openshift-adp get backupstoragelocation
src -n openshift-adp get pods -l name=node-agent
vks -n velero get pods -l name=node-agent

✓ Source OADP and target Velero reachable; BSL Available on both
```

Before any large or concurrent batch, raise the Velero timeouts from their defaults in the DPA — the 30-minute defaults fail 1 TiB volumes under parallel load — and set the client limits and image-backup behavior in the same place:

```
## DPA excerpt --- timeouts, client limits, image backup
apiVersion: oadp.openshift.io/v1alpha1
kind: DataProtectionApplication
spec:
  configuration:
    velero:
      client-qps: 300              # defaults are tuned for small clusters
      client-burst: 500
      resourceTimeout: 4h          # 30-min default fails 1 TiB volumes
    nodeAgent:
      enable: true
      uploaderType: kopia
      dataMoverPrepareTimeout: 4h  # validated for 1 TiB PVCs at 10-way
  backupImages: false              # no registry backup; data-only BSL
```

Approach B replaces only the source-side install: upstream Velero is installed directly on the source with the same capabilities the operator would have declared. Velero 1.14 and later ship CSI support in the core server, so no separate CSI plugin is installed. Every subsequent stage is identical, and the backup artifacts are interchangeable between approaches.

```
## STAGE 0 (Approach B) --- Upstream Velero on the OCP source
# CSI support is in Velero core since 1.14; only the object-store
# plugin is needed.
velero install \
  --kubeconfig "$SRC_KUBECONFIG" \
  --provider aws \
  --plugins velero/velero-plugin-for-aws \
  --bucket "$BACKUP_BUCKET" \
  --backup-location-config \
region=minio,s3ForcePathStyle=true,s3Url="$S3_URL" \
  --use-node-agent \
  --secret-file ./s3-credentials

# Verify engine, node agent, and BSL exactly as in Approach A
src -n velero get pods,backupstoragelocation

✓ Same engine, manual packaging; artifacts interchangeable
```

Stage 1: Preflight Object Scan (Report-Only)

Run the preflight scanner against the source namespace. It classifies every object and writes a report; it does not mutate the cluster. Compare the output to the scenario's expected verdicts and record any divergence in the defect log.

```
## STAGE 1 --- Preflight Object Classification
./preflight-objects.sh \
  --kubeconfig "$SRC_KUBECONFIG" \
  --namespace "$SRC_NS" \
  --report-only \
  --out ./reports/preflight-xform-demo.json

# Compare against the expected verdicts
jq -s '.[0] as $expected | .[1] as $actual
  | $expected | to_entries[]
  | select($actual[.key] != .value)' \
  expected/xform-demo.json reports/preflight-xform-demo.json

✓ 14 objects classified; divergences (if any) logged as defects
```

Stage 2: Backup (Objects and Data)

Back up the namespace with snapshot data movement enabled: each PVC is CSI-snapshotted and its contents uploaded to the store by the node agent. Only the build-time objects classified BLOCK_MIGRATION are excluded. The TTL is set explicitly to cover the migration window. Wait for every DataUpload to complete before declaring the backup done — the Backup phase alone does not cover the data.

```
## STAGE 2 --- Backup (objects + data)
velero backup create "$BACKUP_NAME" \
  --kubeconfig "$SRC_KUBECONFIG" \
  --include-namespaces "$SRC_NS" \
  --snapshot-move-data \
  --ttl 720h \
  --exclude-resources \
imagestreams.image.openshift.io,\
buildconfigs.build.openshift.io

# Data is done only when every DataUpload completes
src -n openshift-adp get dataupload \
  -l velero.io/backup-name="$BACKUP_NAME"

✓ Objects backed up; PV contents uploaded; builds excluded
```

Stage 3: Prepare the Target Namespace

Create the destination namespace, apply the Pod Security Standards enforcement label, and apply the transform overlay — in that order, and before the restore. Creating the namespace first means admission enforcement is already active when workload pods arrive in Stage 4, and the overlay's Ingress and Deployment are in place before traffic or reconciliation depends on them.

```
## STAGE 3 --- Prepare the target namespace
export INGRESS_CLASS="nginx"          # <target-ingress-class>
export APPS_DOMAIN="apps.vks.corp"    # <target-apps-domain>

# Create the namespace and enforce PSS baseline
# (anyuid workloads run as root; restricted would reject them)
vks create namespace "$DST_NS"
vks label namespace "$DST_NS" \
  pod-security.kubernetes.io/enforce=baseline --overwrite

# Apply the target-side conversions (Ingress, Deployment)
envsubst < xform-demo-transform-overlay.yaml | vks apply -f -
```

```
✓ Namespace ready; PSS enforced; overlay applied
```

Stage 4: Restore onto VKS (Objects and Data)

Restore onto the VKS guest cluster with namespace mapping and a storage-class map that redirects source storage classes to the target vSphere CSI class. Because Stage 3 pre-populated the namespace with the overlay, the restore runs with --existing-resource-policy=update; the source originals of transformed objects are excluded. Velero re-provisions each PVC, the node agent downloads its data, and the MIGRATE_AS_IS objects are restored around the transforms.

```
## STAGE 4 --- Restore onto VKS (objects + data)
# Map OCP storage classes to the VKS CSI class
vks -n velero create configmap change-storage-class \
  --from-literal thin-csi=vks-default \
  --from-literal ocs-storagecluster-ceph-rbd=vks-default
vks -n velero label configmap change-storage-class \
  velero.io/plugin-config= \
  velero.io/change-storage-class=RestoreItemAction

velero restore create "$RESTORE_NAME" \
  --kubeconfig "$VKS_KUBECONFIG" \
  --from-backup "$BACKUP_NAME" \
  --namespace-mappings "${SRC_NS}:${DST_NS}" \
  --existing-resource-policy=update \
  --exclude-resources \
routes.route.openshift.io,\
deploymentconfigs.apps.openshift.io

# Data is done only when every DataDownload completes
vks -n velero get datadownload \
  -l velero.io/restore-name="$RESTORE_NAME"

✓ Objects restored; volumes re-provisioned; data downloaded
```

Phase 5: Monitor and Accept

Run the postflight scanner against the target namespace. It confirms that transforms applied, that excluded objects are absent, that operator-backed CRs reconciled, and — critically — that no orphaned references remain.

```
## PHASE 5 --- Postflight Validation Gate
./postflight-objects.sh \
  --kubeconfig "$VKS_KUBECONFIG" \
  --namespace "$DST_NS" \
  --report-only \
  --detect-orphans \
  --out ./reports/postflight-xform-demo.json

# Assert transforms present and excludes absent
vks -n "$DST_NS" get ingress,deploy
vks -n "$DST_NS" get imagestreams 2>/dev/null | wc -l

# Assert every PVC Bound and its data landed
vks -n "$DST_NS" get pvc
vks -n velero get datadownload -o wide

✓ Transforms confirmed; traps flagged; excludes absent
```

A batch is accepted only when the postflight report matches expectations across every object. The acceptance decision is recorded in the evidence package alongside the preflight report, the postflight report, the verdict comparison, and the defect log.

Table 5. Postflight Acceptance Criteria and Validation Checks

Check	Pass condition
Migrate-as-is objects	All present and reconciled on the target
Route transform	Ingress present with correct class and host; Route absent
DeploymentConfig transform	Deployment present with completed rollout
SCC transform	PSS baseline namespace label present and enforced
Excluded objects	ImageStream and BuildConfig absent
Persistent-volume contents	All PVCs Bound; every DataDownload Completed; application-level data check passes
Orphan trap (Trap 1)	RoleBinding flagged as orphaned reference
Enforcement trap (Trap 2)	Privileged trap Pod rejected at admission
Operator CRs (OBJ-CRD-02)	CRD served, CR reconciled to Ready
Verdict comparison	Divergences, if any, logged as tooling defects

⚠ WARNING

Two live copies after restore. Once the data download completes, the volume exists on both clusters. Any writes accepted on the source after the backup snapshot are not on the target — so quiesce or scale down the source workload before the final batch, or run a delta batch immediately before cutover, and never serve traffic from both sides at once. Until cutover, rollback stays inexpensive: delete the restored namespace and its volumes on VKS; the source, never mutated, remains authoritative.

Validation Scenario: Object Transformation (OBJ-XFORM-01)

This scenario runs a controlled, end-to-end test of the object-transformation path on a small, disposable workload before the method is trusted on production namespaces. A fixed set of objects with known migration fates is seeded, migrated through Stages 0–4, and checked against its recorded expected verdicts. It validates two things at once: that the transformation path produces standard Kubernetes objects the VKS cluster serves directly, and that the preflight and postflight scanners classify those objects correctly.

Scenario Design

A dedicated namespace, xform-demo, is seeded with fourteen objects spanning the three migration fates plus two deliberate failure traps, chosen so that every fate and every transform mapping is exercised at least once.

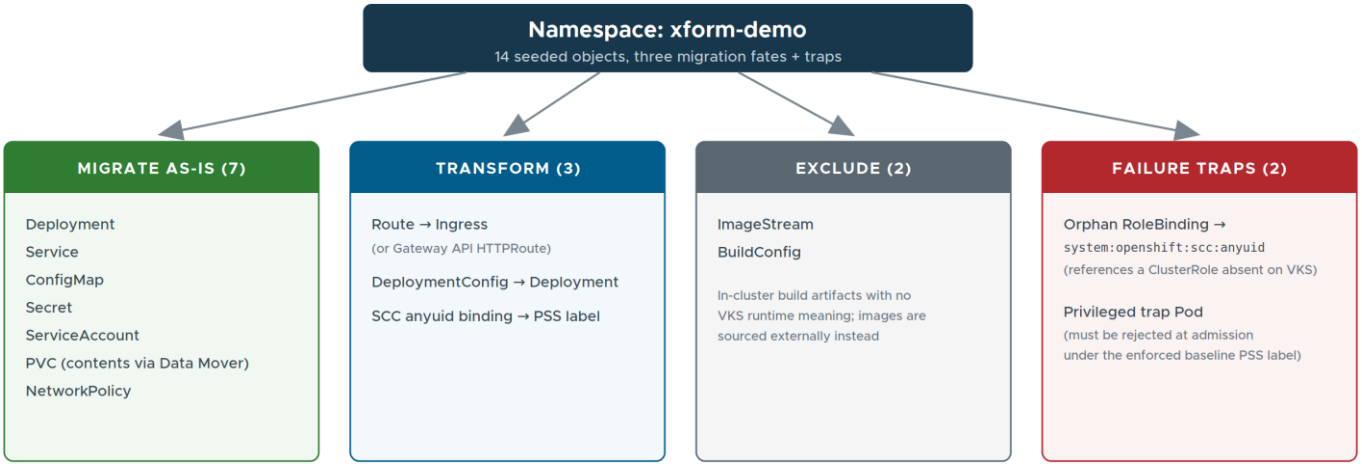


Figure 3: OBJ-XFORM-01 scenario layout — fourteen seeded objects across three migration fates plus two failure traps.

Table 6. Validation Namespace Object Inventory by Migration Fate

Fate	Seeded objects	Count
Migrate as-is	Deployment, Service, ConfigMap, Secret, ServiceAccount, PVC, NetworkPolicy	7
Transform	Route → Ingress, DeploymentConfig → Deployment, SCC anyuid binding → PSS baseline label	3
Exclude	ImageStream, BuildConfig	2
Trap objects	RoleBinding → system:openshift:scc:anyuid (orphan trap); privileged trap Pod (enforcement trap)	2
Total	—	14

Deliberate Failure Traps

A scenario that only ever passes proves nothing about the tooling's ability to catch failure. OBJ-XFORM-01 embeds two traps that must be caught for the scenario to be considered valid.

⚠ WARNING

Trap 1 — Orphan reference. A RoleBinding references the source-specific ClusterRole system:openshift:scc:anyuid, which is absent on VKS. Postflight orphan detection must flag this binding as referencing a non-existent subject. If it does not, the scanner has a gap.

Trap 2 — Inert enforcement. A Pod requesting privileged mode is seeded deliberately. Under the enforced baseline PSS label it must be rejected at admission. If it runs, the label was either not applied or not enforced — the transform passed silently with a weakened security posture, and the postflight scanner must flag it.

Expected Verdicts

Each scenario's answer key is a recorded table of the expected verdict for every seeded object. The preflight scanner is run against xform-demo on the source and its output compared to that table; after restore, the postflight scanner is run against the target. Any divergence is logged as a scanner defect.

Table 7. Expected Verdict Matrix for Validation Objects

Object	Preflight verdict	Postflight expectation
Deployment / Service / ConfigMap	MIGRATE_AS_IS	Present, unchanged, reconciled
Secret / ServiceAccount / NetworkPolicy	MIGRATE_AS_IS	Present, unchanged
PVC (with contents)	MIGRATE_AS_IS	Re-provisioned, Bound; data downloaded and verified
Route	REVIEW_REQUIRED	Replaced by Ingress + TLS Secret
DeploymentConfig	REVIEW_REQUIRED	Replaced by Deployment, rollout complete
SCC anyuid binding	REVIEW_REQUIRED	PSS baseline namespace label present; enforced
ImageStream / BuildConfig	BLOCK_MIGRATION	Absent from target (excluded)
RoleBinding → anyuid ClusterRole	REVIEW_REQUIRED	Flagged as orphaned reference (Trap 1)
Privileged trap Pod	MIGRATE_AS_IS	Rejected at admission by baseline PSS (Trap 2)

Acceptance Checklist

The scenario is accepted only when every checklist item passes. Items that fail because the scanner missed a verdict are recorded in the defect log as tooling gaps — they do not, by themselves, fail the migration path, because the migration path and the tooling are evaluated independently.

1. All seven MIGRATE_AS_IS objects present and reconciled on the target.
2. Route replaced by a functioning Ingress with the correct class and host.
3. DeploymentConfig replaced by a Deployment with a completed rollout.
4. SCC anyuid dependency expressed as an enforced baseline PSS namespace label.
5. ImageStream and BuildConfig absent from the target namespace.

6. Orphaned RoleBinding flagged by postflight (Trap 1 caught).
7. Privileged trap Pod rejected at admission (Trap 2 caught).
8. Preflight verdicts match the expected verdicts for all fourteen objects.
9. Any divergence recorded in the scanner defect log.

Validation Scenario: CRD and Operator Migration (OBJ-CRD-02)

OBJ-CRD-02 extends the same pattern to the hardest object class: an operator-backed CustomResource whose CustomResourceDefinition and controlling operator are absent on the target. A CR restored without its CRD is either rejected outright or restored as an inert object that nothing reconciles. The scenario seeds a CRD/CR pair backed by an operator that exists on the source but not on the target, and exercises four behaviors in sequence: BLOCK_MIGRATION classification, install-CRD-first ordering, reconcile-check validation, and CRD version-skew handling.

BLOCK_MIGRATION Classification

The preflight scanner must classify the custom resource as BLOCK_MIGRATION because its CRD is not served by the target API. A CR that classifies as MIGRATE_AS_IS or REVIEW_REQUIRED when its CRD is absent is a scanner defect — the single most consequential class of miss, because it would allow an unreconcilable object to be restored unnoticed.

Install-CRD-First Ordering

The batch must install the CRD and its operator on the target before the custom resource is restored. Restoring the CR first — before its CRD is served — is the failure this ordering exists to prevent. The same principle generalizes to admission webhooks: order restores so webhook backends and operators are Available before the objects they admit.

```
## OBJ-CRD-02 --- Install-CRD-First Ordering
# 1. Install the CRD on the target
vks apply -f crd/widget-crd.yaml

# 2. Deploy the operator and wait for availability
vks apply -f operator/widget-operator.yaml
vks -n widget-system rollout status \
  deployment/widget-operator --timeout=300s

# 3. Restore only the CR, now that its CRD is served
velero restore create objcrd02-cr \
  --kubeconfig "$VKS_KUBECONFIG" \
  --from-backup objcrd02-src \
  --include-resources widgets.example.com

✓ CRD served, operator Available, CR restored in order
```

Reconcile-Check Validation

Installing the CRD and restoring the CR is necessary but not sufficient. The postflight scanner must confirm that the operator actually reconciled the restored CR — that the resource reached its Ready condition and that the operator produced its expected managed objects. A CR that is present but never reconciled is a failure a naive object-count check would miss.

```
## OBJ-CRD-02 --- Reconcile Check
vks -n "$DST_NS" wait \
  --for=condition=Ready \
  widget/objcrd02-sample \
  --timeout=300s

# Confirm operator-managed children exist
vks -n "$DST_NS" get deploy,svc \
```

```
  -l app.kubernetes.io/managed-by=widget-operator

✓ CR reconciled; operator-managed objects present
```

CRD Version-Skew Handling

When the source and target CRD schemas differ — a served version added, deprecated, or removed — the restored CR may be expressed in a version the target no longer serves, or may require conversion. The scenario validates that version skew is detected and resolved before the CR is accepted: the target CRD must serve a version compatible with the stored object, and any conversion webhook the operator relies on must be present.

> ⛔ IMPORTANT
>
> Version skew is a blocking condition. If the target CRD does not serve a version compatible with the backed-up CR, the object must remain BLOCK_MIGRATION until the schema is aligned. Restoring a CR against an incompatible CRD version risks silent data loss on fields the target schema drops.

Validated Results and Capacity Planning

Every scenario in this section was validated end-to-end — backup, restore, and byte-exact verification of the restored contents through mounted verify pods — in the environment described above. Across the five scenarios, 6,020 GiB (~5.9 TiB) of user data was migrated with zero data-integrity errors.

> NOTE
>
> Units. All capacities use binary units (1 TiB = 1,024 GiB) and all throughputs are MiB/s, derived as user data ÷ wall clock.

Validation Summary

Table 8. Validation Test Results and Data Migration Performance

Scenario	Workload (user data)	Backup	Restore	Result
Single small PVC	1 × 450 GiB	24 min	~70 min	Pass, 0 errors
Single large PVC	1 × 800 GiB	54 min	100 min	Pass, 0 errors
4-way parallel	4 × 450 GiB (1.76 TiB)	86 min	~70 min	Pass, 0 errors
1 TiB cross-filesystem	1 × 1 TiB PVC, 900 GiB filled	54 min	100 min	Pass, 0 errors
10-way mixed (Test C)	1×1 TiB + 4×200 GiB + 5×100 GiB PVCs, 2,070 GiB filled (~2.02 TiB)	160 min	177 min	Pass, 0 errors

Observed Throughput

Each backup runs in two phases: a CephFS clone of the source snapshot (5–15 minutes depending on size), then the Kopia upload from the clone to the BSL for the rest of the wall clock. Restores skip the clone phase and stream directly from the store into the newly provisioned target volume.

Table 9. Observed Backup Throughput Across Validation Workloads

Backup workload	Duration	Aggregate throughput
1 × 450 GiB single PVC	24 min	~320 MiB/s
1 × 800 GiB single PVC	54 min	~253 MiB/s
1 × 1 TiB (900 GiB filled)	54 min	~284 MiB/s
4 × 450 GiB parallel (1.76 TiB)	86 min	~357 MiB/s aggregate
10-way mixed (2,070 GiB, Test C)	160 min	~221 MiB/s aggregate

Table 10. Observed Restore Throughput Across Validation Workloads

Restore workload	Duration	Aggregate throughput
1 × 450 GiB single PVC	~70 min	~110 MiB/s
1 × 800 GiB single PVC	100 min	~137 MiB/s
1 × 1 TiB (900 GiB filled)	100 min	~154 MiB/s
4 × 450 GiB parallel	~70 min	~439 MiB/s aggregate
10-way mixed (Test C)	177 min	~200 MiB/s aggregate

Returns diminish past four to five concurrent streams on a four-worker cluster: Ceph cloner threads saturate even at max_concurrent_clones=16, node-agent load slots fill (loadConcurrency 10 per node), and MinIO disk writes contend under ten simultaneous Kopia streams. Ten-way parallelism moves a larger batch in one pass, but measured aggregate throughput fell below four-way on both legs: ~221 versus ~357 MiB/s on backup, and ~200 versus ~439 MiB/s on restore. On the backup leg the drop tracks the contention described above. On the restore leg the target cluster showed no saturated resource; the observed contention was at the shared MinIO store serving ten concurrent Kopia streams. For batches that must move in one pass, plan restores at the ~200 MiB/s aggregate observed at ten-way rather than the four-way peak.

For planning on comparable hardware (four-worker Ceph source, 10 GbE network), expect an aggregate 220–360 MiB/s per backup batch and 200–440 MiB/s per restore batch depending on parallelism. Compute the migration window as total user data divided by that rate, plus roughly 20% overhead for Backup CR coordination and restore-side PVC provisioning.

Capacity Planning

Cross-filesystem migration introduces an overhead asymmetry that determines how full source volumes can safely be. CephFS reports the full nominal PVC capacity as usable, because its metadata lives in a separate Ceph pool. The vSphere CSI target formats ext4, which pre-allocates inode tables at format time (about 1.7% of capacity — 17 GiB on a 1 TiB volume) and reserves a further 5% of blocks for root. Effective usable capacity on the target is therefore roughly 93% of nominal. Fill source PVCs no higher than 90% of nominal capacity, and size target PVCs larger than the source when migrating pathologically full volumes.

Concurrent backups clone each source PVC on CephFS before upload, so peak raw usage during the backup window is roughly (source data + clones) × replication factor. In Test C, 2,070 GiB of user data at 3× replication peaked near 12.4 TiB raw on the 23 TiB cluster — about 54% utilization, safely below the ODF nearfull threshold of 0.75. Watch per-pool MAX AVAIL rather than cluster-wide raw free space, and keep any concurrent batch at or below roughly 70% of MAX AVAIL.

Size the backup store at 1.2× the largest expected migration batch or more. When MinIO fills, Kopia fails with “Storage backend has reached its minimum free drive threshold” and each DataUpload retries ten times before failing; recovery means deleting old backups or expanding the MinIO volume.

Test C Reference Configuration

Test C is the validated reference for customer-realistic migration batches and the template for sizing and planning.

Table 11. Test C Reference Configuration for Migration Validation and Sizing

Tier	Count	PVC size	fio fill	Total user data
Large	1	1 TiB	900 GiB	900 GiB
Medium	4	200 GiB	180 GiB	720 GiB

Tier	Count	PVC size	fio fill	Total user data
Small	5	100 GiB	90 GiB	450 GiB
Total	10	—	—	2,070 GiB (~2.02 TiB)

Table 12. Test C End-to-End Migration Performance Summary

Phase	Wall clock	Aggregate throughput	Errors
Backup (10 parallel)	160 min	~221 MiB/s	0
Restore (10 parallel)	177 min	~200 MiB/s	0
End-to-end	337 min (5 h 37 m)	—	0

Two prerequisites made the ten-way run clean: raising the Velero timeouts (resourceTimeout and dataMoverPrepareTimeout, both to 4 h) before launch, and headroom on both stores — Ceph peaked at ~54% and MinIO consumed about 2 TiB of 8 TiB. All ten volumes verified byte-exact on the target, DataDownload helper pods spread across all three target workers, and every Restore CR completed without manual intervention.

Approach A vs. Approach B on OpenShift

Both approaches were exercised during validation. The table records the comparison as deployed in this environment.

Table 13. Comparison of OADP and Upstream Velero on OpenShift

Aspect	Approach A — OADP (operator-managed)	Approach B — Upstream Velero
Underlying engine	Upstream Velero	Upstream Velero (same binary, same backup format)
Install / config	One DPA custom resource declares Velero, plugins, BSL, node agent	velero install; components assembled and versioned manually
CSI + Data Mover	Packaged and version-matched by the operator	Functional; plugin/agent compatibility managed by you
Lifecycle / upgrades	OLM-managed, upgraded as a unit	Manual; each component tracked separately
Support model	Red Hat supported on OCP	Community / CNCF
Measured backup — 1 × 450 GiB	24 min	~90 min (as configured)
Measured backup — 4 × 450 GiB	86 min (parallel)	~7 h (serial, as configured)
Measured backup — 10-way (2,070 GiB)	160 min (parallel)	Not practical serially
Target side	Native Velero on VKS	Native Velero on VKS (identical)

NOTE

Read the timing rows carefully. The measured gap primarily reflects how each install was configured — the OADP deployment ran DataUploads in parallel out of the box, while the manual install as assembled processed the same volumes serially. It is not an engine difference; both run the same binary, and a manually tuned upstream install can parallelize equally. The validation verdict favors Approach A on different grounds: it keeps the CSI integration, Data Mover, and node agent version-matched, survives upgrades as a unit, and removed an entire class of plugin-compatibility defects that manual assembly exposed. Approach B remains fully workable, produces identical backup artifacts, and is the only option on sources with no curated operator.

Appendix: Operational Pain Points and Triage (MinIO-Backed Example)

The pain points below were encountered or designed around during validation. The store-side entries are specific to MinIO and should be re-validated before being applied to another S3-compatible store such as NooBaa or AWS S3; the Velero and Data Mover entries apply to any store. Most of these failures are silent or misleading — a backup that reports success but cannot be read, a restore that completes while skipping every object, a signature error that points at credentials when the real fault is the addressing style.

MinIO Backup Store Pain Points

MinIO is S3-compatible, not S3-identical. Every one of the following differences has produced a failed or unreadable backup in real migrations.

Table 14. Common MinIO Backup Store Pain Points and Resolutions

Pain point	Symptom	Resolution
Path-style addressing	SignatureDoesNotMatch or DNS resolution failures on bucket access	Set s3ForcePathStyle: "true" in the BSL config. MinIO serves endpoint/bucket, not bucket.endpoint.
Region mismatch	Signature errors despite reachable endpoint and valid credentials	Set region consistently in the BSL (commonly minio or us-east-1); the SDK signs against it even when MinIO ignores it.
Missing publicUrl	Backups succeed but velero backup logs / describe --details cannot fetch objects	Set publicUrl in the BSL when the in-cluster s3Url is not resolvable from the operator workstation.
Bucket does not exist	BSL stuck Unavailable; backups fail immediately	Pre-create the bucket (mc mb). Velero never creates it.
Credential format	BSL Unavailable with access-denied despite correct keys	Use AWS INI format with the exact profile name the BSL references ([default] unless overridden).
Self-signed TLS	x509: certificate signed by unknown authority	Provide caCert (base64) in the BSL, or terminate TLS in front of MinIO. Avoid insecureSkipTLSVerify outside labs.
Checksum incompatibility	AWS SDK trailing-checksum errors against older MinIO releases	Set checksumAlgorithm: "" in the BSL config, or upgrade MinIO to a release that accepts streaming trailers.
Asymmetric reachability	Source backs up; target restore cannot list or fetch the backup	Both clusters must resolve and reach the same s3Url. Verify from the VKS target before the batch, not during it.
Wiped or re-created bucket	error to connect to repository: repository not initialized in the provided storage	Stale BackupRepository CRs point at the vanished Kopia repo. Delete them (oc delete backuprepositories.velero.io --all -n openshift-adp); they re-initialize on the next backup.

The following BSL shows a MinIO-correct configuration with every pain point addressed explicitly.

```
## MinIO-correct BackupStorageLocation
apiVersion: velero.io/v1
kind: BackupStorageLocation
metadata:
  name: default
  namespace: velero
spec:
  provider: aws
```

```
  objectStorage:
    bucket: vks-migration            # pre-created with mc mb
    caCert: <base64-ca-bundle>       # self-signed TLS chain
  config:
    region: minio                    # must match SDK signing
    s3ForcePathStyle: "true"         # MinIO is path-style
    s3Url: https://minio.corp:9000     # reachable from BOTH clusters
    publicUrl: https://minio.corp:9000 # for logs/describe
    checksumAlgorithm: ""            # older MinIO compatibility

✓ BSL Available on source AND target before any batch
```

Velero Engine Pain Points

Velero's defaults are tuned for disaster recovery into an empty cluster, not for migration into a live one. Each of the following behaviors must be explicitly handled by the batch plan.

Table 15. Common Velero Migration Engine Pain Points and Resolutions

Pain point	Symptom	Resolution
Existing resources skipped	Restore completes; objects in a populated namespace silently unchanged	Set --existing-resource-policy=update (this methodology restores into a namespace pre-populated only with the transform overlay, per Stage 3).
PartiallyFailed misread	Restores routinely end PartiallyFailed with warnings; teams read it as success or as failure	Run velero restore describe --details and triage warnings item by item; gate acceptance on the postflight scan, not the restore phase.
Service clusterIP retained	Restored Services carry the source spec.clusterIP; conflicts or CIDR mismatch on target	Strip clusterIP via restore modifiers, or tolerate reassignment; validate NodePort collisions before the batch.
Admission webhooks reject objects	Objects rejected at restore because their validating/mutating webhook backend is not yet running	Order restores so webhook backends and operators are Available first — the generalized form of install-CRD-first.
Stale ServiceAccount tokens	Restored service-account-token Secrets are invalid on the target	Exclude legacy token Secrets; let the target mint tokens. Workloads using projected tokens are unaffected.
Cluster-scoped gaps	Namespaced backup restores RoleBindings whose ClusterRoles were never captured	Inventory cluster-scoped dependencies in Discovery; restore them deliberately or flag orphans postflight (Trap 1).
Stuck finalizers	Restored objects wait forever on finalizers whose controllers do not exist on target	Classify controller-owned objects BLOCK_MIGRATION until the controller exists; strip foreign finalizers in the overlay.
Backup TTL expiry	Backups silently garbage-collected from the store mid-project (default TTL 30 days)	Set --ttl explicitly to cover the full migration window, including rollback reserves (Stage 2 uses 720h).

Data Mover Pain Points

The CSI snapshot Data Mover is what turns a manifest tool into a data-migration tool, and it has its own failure modes. Most present as a backup or restore that sits in progress indefinitely while the phase reports nothing wrong — which is why the data CRs, not the phase, are the source of truth.

Table 16. Common Data Mover Pain Points and Resolutions

Pain point	Symptom	Resolution
Missing VolumeSnapshotClass label	DataUpload stuck Pending; backup never progresses past snapshotting	The VolumeSnapshotClass must carry the velero.io/csi-volumesnapshot-class label (or be set in the DPA); without it the CSI snapshot is never taken.
No storage-class mapping on target	Restored PVCs Pending forever; provisioning fails against a source class name VKS does not have	Apply the change-storage-class ConfigMap (Stage 4) mapping every source class to the VKS vSphere CSI class before the restore.
Node agent resource limits	Node-agent pods OOMKilled or evicted mid-upload on large volumes; DataUpload restarts or fails	Raise node-agent memory/CPU via the DPA (nodeAgent resource allocations); size for the largest single volume, not the average.
Crash-consistent snapshots only	Databases restore but replay or discard in-flight writes; application-level corruption on busy volumes	CSI snapshots are crash-consistent. Add Velero pre/post hooks (fsfreeze, DB flush-and-lock) or quiesce the workload for app-consistent batches.
Concurrency ceiling	Batches with many PVCs serialize; total duration far exceeds largest-volume time	Raise node-agent load concurrency and spread stateful namespaces across batches; uploads parallelize per node, not per cluster.
Phase completes before data	Backup or Restore shows Completed while DataUpload / DataDownload still run or have failed	Gate every stage on the DataUpload/DataDownload CRs (as in Stages 2 and 4), never on the Backup/Restore phase alone.
Prepare timeout on large PVCs under load	DataUpload Failed: timeout on preparing data upload — the default dataMoverPrepareTimeout of 30 min is exceeded by a 900+ GiB CephFS clone under concurrent load	Set resourceTimeout: 4h and dataMoverPrepareTimeout: 4h in the DPA (Stage 0 excerpt); validated for 1 TiB PVCs at 10-way concurrency.
WaitForFirstConsumer target storage class	DataDownload stalls in Accepted for hours; the node agent cannot create its expose pod because restored PVCs carry a velero.io/dynamic-pv-restore selector	Map to the Immediate-binding variant of the target storage class in the change-storage-class ConfigMap; never a late-binding class.
Orphan CephFS subvolumes on the source	Failed backups leave clones invisible to Kubernetes; raw Ceph capacity shrinks (observed: ~10 TiB consumed, HEALTH_WARN backfillfull)	Diff the subvolume names referenced by live PVs against ceph fs subvolume ls in the rook toolbox; remove orphan snapshots first, then the subvolumes.
Wedged PVCs after a failed restore	kubectl delete pvc hangs; the vSphere CSI finalizer cannot complete cleanup, blocking re-restores onto the same claim name	Patch metadata.finalizers to null on the stuck PVCs, then force-delete, before re-running the restore.

Network Throttling and Throughput Pain Points

Two very different traffic profiles share the path to the store. The object phase is thousands of small Kubernetes API calls and small S3 PUTs — request-rate bound, not bandwidth bound. The data phase is the opposite: the Data Mover streams full volume contents through the node agents into the store, so stateful batches are governed by upload and download throughput, link bandwidth, and store disk speed. Diagnosing a slow batch starts with knowing which phase is slow.

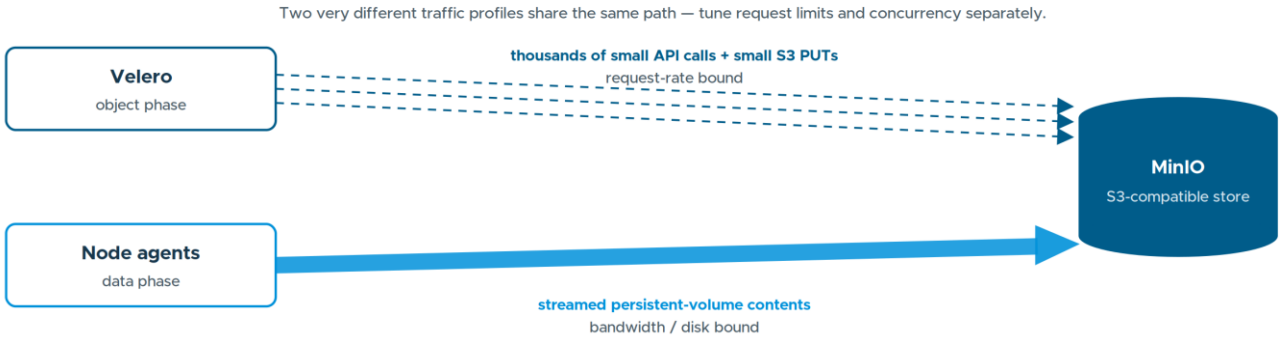


Figure 4: Object and data traffic profiles — request-rate-bound object phase and bandwidth-bound data phase share the same MinIO store.

Table 17. Scalability and Operational Pain Points in Velero-Based Migration Workflows

Pain point	Symptom	Resolution
Velero client-side API throttling	Backups of object-dense namespaces crawl; velero logs show client-side throttling / waited for … due to client-side throttling	Raise the Velero API client limits (client-qps / client-burst) via the DPA, or server flags on upstream Velero; defaults are tuned for small clusters.
API Priority and Fairness (APF)	429 Too Many Requests from the apiserver during discovery or backup on busy clusters	Schedule batches off-peak; if recurring, grant the Velero ServiceAccount a dedicated FlowSchema / priority level rather than raising cluster-wide limits.
MinIO request-rate limiting	503 SlowDown responses under concurrent request bursts; Velero retries inflate batch duration	Tune MINIO_API_REQUESTS_MAX and deadline settings; stagger concurrent namespace backups rather than launching all batches at once.
Proxy interception	S3 calls hang, stall, or fail TLS while direct cluster traffic is healthy	Add the MinIO endpoint to NO_PROXY on the Velero pods (the DPA supports proxy env); corporate proxies buffer and throttle streamed S3 traffic.
East-west firewall inspection	High per-request latency between cluster and MinIO inside VCF; many small PUTs amplify it	Exempt the backup path from NSX distributed-firewall deep inspection, or place MinIO on a network segment reached without inspection.
Discovery and resource timeouts	Backups fail on clusters with very large CRD counts; timeouts during resource enumeration	Keep resourceTimeout raised (4h per Stage 0); prune the backup scope with --include-resources rather than backing up every API group.
Phase misdiagnosis	A slow batch blamed on bandwidth when the object phase is throttled, or	Time the phases separately: object backup completes in minutes; DataUpload duration scales with volume size. Fix request rates for the former, bandwidth and store disk throughput for the latter.

Pain point	Symptom	Resolution
	on QPS when the data phase saturates a link	
Conflicting BackupRepository CRs from a prior upstream install	Operator install fails: updated validation is too restrictive: [].spec.resticIdentifier: Required value	The operator ships a stricter BackupRepository CRD; Kopia-only repositories lack resticIdentifier. Delete the existing BackupRepository CRs and the failed Subscription/CSV, then re-create the Subscription.

With an operator-managed install, the client limits are set in the DPA (Stage 0 excerpt); on upstream Velero, the same values are passed as server arguments. A timed dry-run batch establishes the baseline before tuning.

```
## Baseline and tune API throttling
# OADP: client-qps / client-burst live in the DPA (Stage 0)
# Upstream Velero equivalent:
velero server --client-qps=300 --client-burst=500

# Baseline a batch before and after tuning
time velero backup create baseline-"$SRC_NS" \
  --include-namespaces "$SRC_NS" --snapshot-volumes=false --wait

✓ Batch duration measured; throttling tuned, not guessed
```

NOTE

Throttle symptoms mimic store failures. A throttled backup and a broken BackupStorageLocation both present as a batch that hangs. Check the Velero pod logs for client-side throttling messages and MinIO for 503 SlowDown before re-running the store triage — the fixes are different.

Diagnosing a Failed or Partial Restore

The triage sequence below resolves the majority of restore incidents against a MinIO-backed BSL. It is run before any batch is retried, and its output is attached to the evidence package.

```
## Restore triage against a MinIO-backed BSL
# 1. Store reachable from the TARGET
velero backup-location get --kubeconfig "$VKS_KUBECONFIG"

# 2. Backup visible and not expired
velero backup get --kubeconfig "$VKS_KUBECONFIG"

# 3. Read the warnings, not just the phase ---
#    distinguish skipped-existing from failed
velero restore describe "$RESTORE_NAME" --details \
  --kubeconfig "$VKS_KUBECONFIG"
velero restore logs "$RESTORE_NAME" \
  --kubeconfig "$VKS_KUBECONFIG" | grep -Ei 'error|warn'

✓ Root cause classified: store, ordering, policy, or object
```

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 Kubernetes workloads to VKS on VCF 9 means moving the data and reworking the objects, and Velero handles both in the same batch: the CSI snapshot Data Mover carries each persistent volume's contents through the S3-compatible store onto freshly provisioned vSphere CSI volumes, while every namespace object is classified and handled according to its verdict. The methodology in Part 1 applies to any Kubernetes source Velero supports, whether Velero runs there as the upstream distribution or as a vendor-curated packaging; the VKS target runs native Velero in every case.

The worked example demonstrates the method against an OpenShift bare-metal source with OADP and MinIO: 6,020 GiB moved across five scenarios with zero data-integrity errors, at up to ten concurrent volumes per batch. Just as important, the test scenarios measure the tooling itself — each seeds known faults (an orphaned RoleBinding, a privileged pod, a CRD absent from the target) that the scanners must catch, and each records its expected verdicts in advance. A missed flag is a tooling defect, logged and fixed before the next batch; a caught one is evidence. The result is a migration whose outcome and whose tooling accuracy are both measured, both auditable, and both known before cutover rather than after it.

About the Author

Christian Rauber is a Product Marketing Engineer at Broadcom, specializing in VMware Cloud Foundation and Kubernetes platform technologies. He is a member of the Workloads group within the VCF division and focuses on validation and documentation of workloads on VCF and VKS.

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