portworx
by Everpure

Demystifying Kubernetes

for VMware Admins

Contents

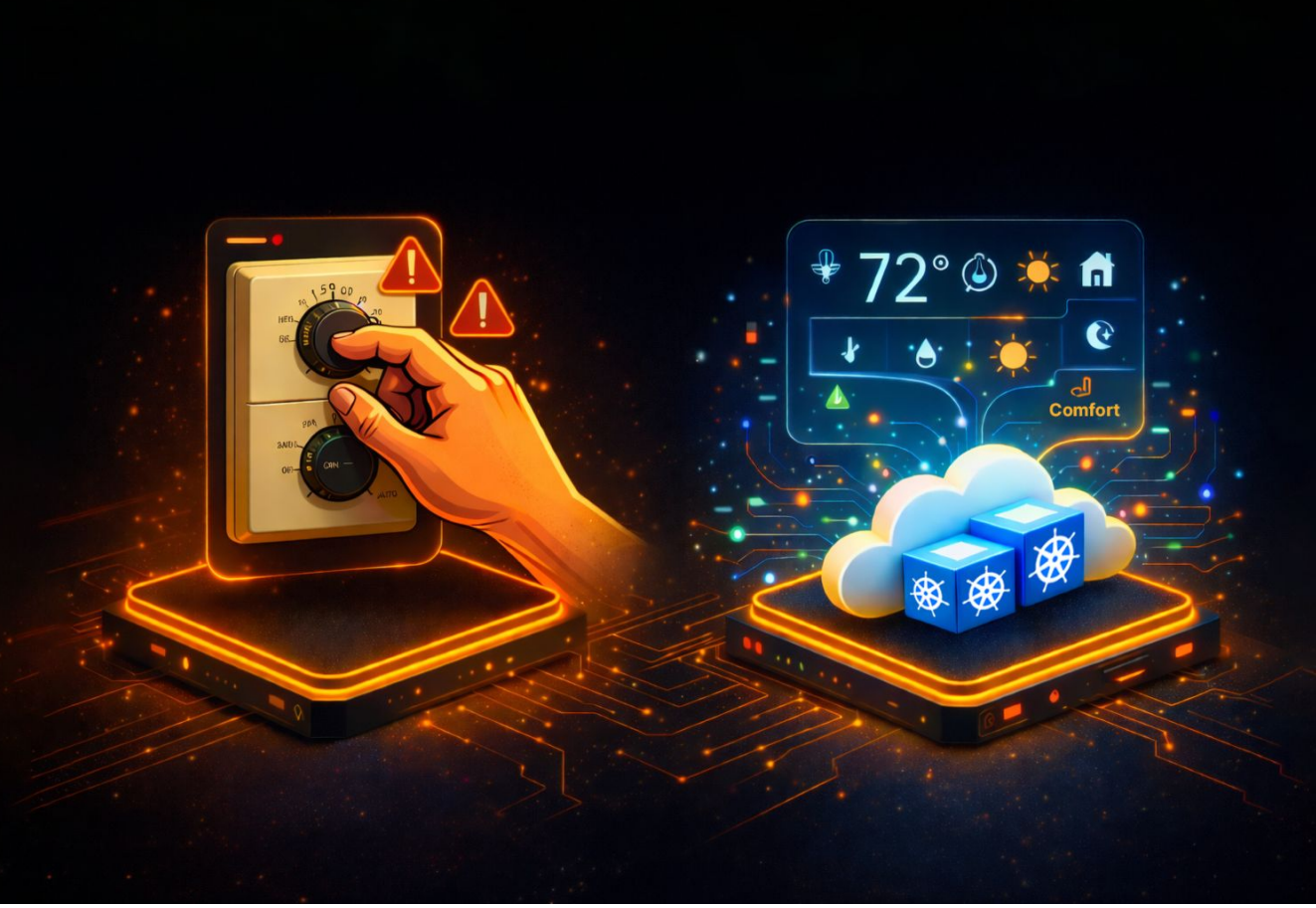
01

Chapter One

ClickOps to GitOps

The Paradigm Shift

For VMware admins, the challenge isn't starting over—it's learning how their existing expertise translates to Kubernetes and cloud-native operations.

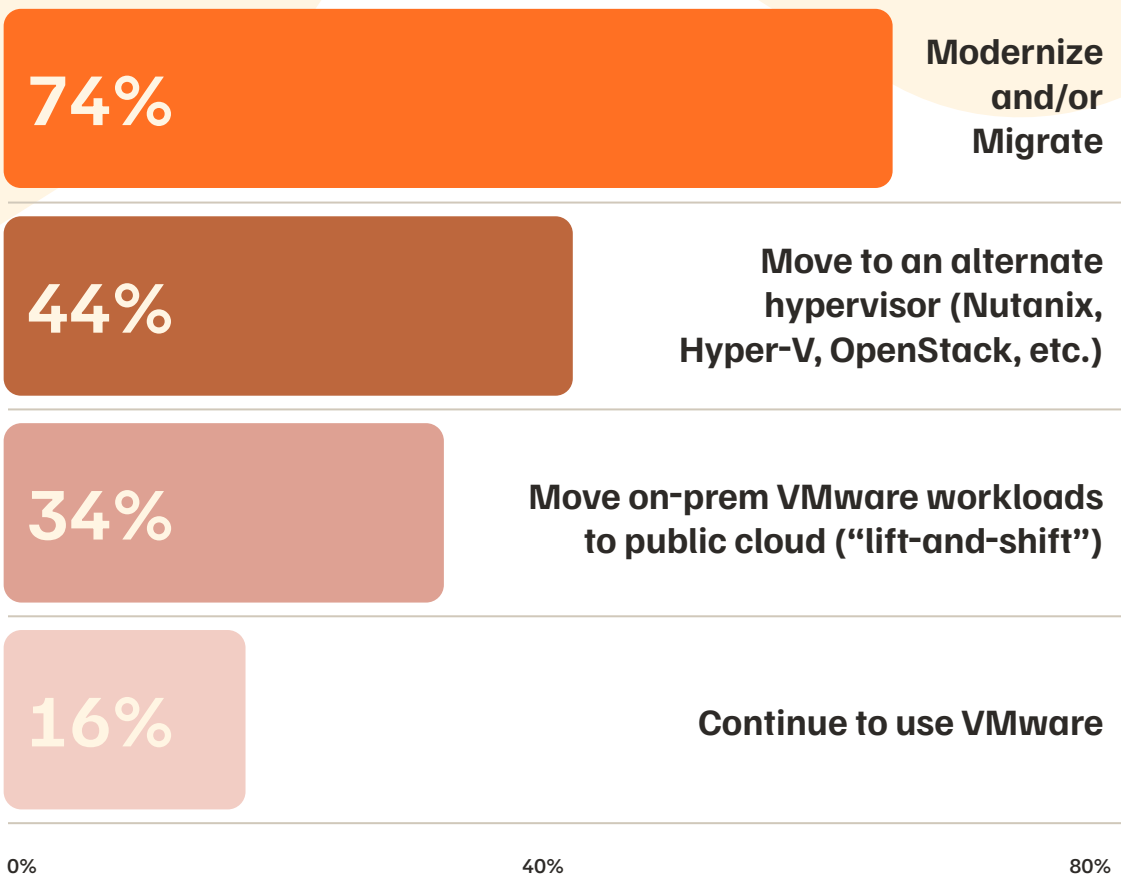


The VMware era shaped a generation of infrastructure professionals. You learned to manage data centers through vCenter's interface. You clicked through wizards, configured VMs, and built reliable infrastructure that powered enterprises for decades. That expertise has proven value.

But the ground is shifting beneath your feet.

Broadcom's acquisition of VMware changed the economics overnight. License costs increased dramatically. Support models shifted. Organizations that ran VMware for 15 years are now questioning their infrastructure strategy for the first time.

What has your company done, or has plans to do, with VMware workloads?
Choose all that apply.



Source: Voice of Kubernetes Report 2026

You need to learn new tools and new workflows—**not new fundamentals.”**

This isn't about whether Broadcom is right or wrong. This is about recognizing a forcing function. The acquisition accelerated a transition that was already underway. Kubernetes adoption has grown steadily for years. The Broadcom catalyst compressed years of gradual change into months of accelerated decisions.

Your organization faces a choice. Pay significantly more for the same VMware capabilities. Or invest that budget into building cloud-native infrastructure skills. Many organizations are choosing option two.

Now you need to make it happen!

This is a forcing function for VM admins like you. The real question is how you move forward without starting over. You already understand infrastructure. You know networking, storage, compute, and security. Those concepts translate directly to Kubernetes. The mental models you built over years of VMware work remain valuable. You need to learn new tools and new workflows, not new fundamentals.

At Portworx, we have built this 10-part series to demystify Kubernetes from a VM admin’s perspective. This series will map VMware concepts to their Kubernetes equivalents. Each part addresses a domain you already understand and shows how it translates to the cloud-native world. This blog series will guide you through that transition.

Let's start with the basics in the blog - from ClickOps to GitOps.

The ClickOps Model and Its Limits

Let's start with the most significant shift: how you operate infrastructure. VMware built its success on making complex operations accessible through graphical interfaces. vCenter provides a single pane of glass for your entire infrastructure. You point, click, configure, and deploy.

This model works. Millions of VMs run in production because administrators used vCenter's interface to create them. The GUI provides immediate feedback. You see your changes reflected in real time. Error messages appear in dialog boxes. Logs populate in dedicated windows.

Consider how you deploy a new VM today:

- Log into vCenter
- Right-click a cluster
- Select "New Virtual Machine"
- Walk through a wizard
- Select a template
- Configure CPU, memory, and disk
- Choose a network
- Click through screens until you reach "Finish"
- The VM appears in your inventory

You watch its creation progress. It powers on. You move to the next task.

This workflow is called ClickOps.
Operations driven by clicking through graphical interfaces.

ClickOps delivers immediate satisfaction.

You see results instantly. The learning curve feels manageable because the interface guides you. VMware invested heavily in UX design to make infrastructure management approachable.

Advanced VMware professionals recognized the limits of manual operations years ago. Many adopted Infrastructure as Code tools to automate provisioning. Terraform modules deploy VMs consistently. Ansible playbooks configure guest operating systems. PowerCLI scripts batch repetitive tasks. vRealize Automation provides self-service catalogs backed by code. These approaches bring version control and repeatability to VMware environments. They represent a significant step toward declarative infrastructure. Yet IaC adoption within VMware shops remains uneven. Many administrators prefer the GUI for its immediacy and visual feedback. Some organizations lack the development culture required for IaC. Others invested in automation only for specific use cases while handling everything else manually. The result is a spectrum: some teams operate almost entirely through code, while others rarely leave vCenter's interface.

But ClickOps creates problems at scale.

Every click you make in vCenter is ephemeral. No record exists of which buttons you pressed or which options you selected. The VM exists, but the process that created it vanished the moment you clicked "Finish."

When a colleague asks how you configured that VM, you describe it from memory. When you need to create 50 similar VMs, you click through the wizard 50 times. Or you build automation after the fact to replicate what you did manually.

Consider what happens when something breaks at 2 AM. You log into vCenter. You click through menus searching for the problem. You find it, fix it, and go back to sleep. Tomorrow, your colleague asks what happened. You explain verbally. Maybe you write it up. Maybe you don't.

Now multiply this across a team of ten administrators over five years. Thousands of configuration decisions exist only in your current infrastructure and in the memories of the people who made them.

ClickOps also struggles with consistency. Human operators make human mistakes. The VM you create on Monday might differ slightly from the one you create on Friday. Those differences compound over time. Your infrastructure drifts from its intended state without anyone noticing.

You've felt this problem even if you haven't named it. You've inherited VMs configured by people who left the company years ago. You've spent hours figuring out why one server behaves differently from its neighbors. You've rebuilt environments from scratch because documenting the existing state would take longer than starting over.

These aren't VMware problems. These are ClickOps problems. The GUI made the infrastructure accessible. It also made infrastructure opaque.

The GitOps Model Kubernetes Demands

Kubernetes operates on a different philosophy. You declare the state you want. Kubernetes works continuously to make that state real.

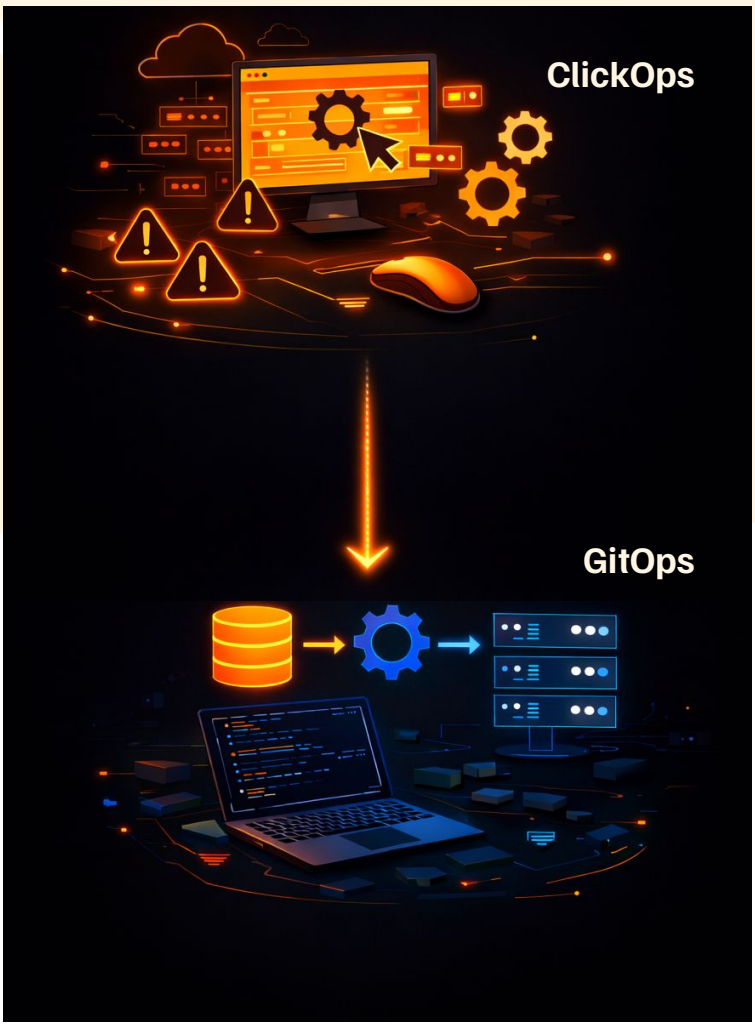
This is declarative infrastructure. You write a YAML file describing your desired configuration. You commit that file to a Git repository. An automated process applies that configuration to your cluster. Kubernetes compares the desired state to the current state and reconciles any differences.

Consider deploying an application in Kubernetes:

You write a manifest describing your deployment. Three replicas. Two gigabytes of memory each. A specific container image. Environment variables. Health checks.

You then commit this manifest to Git. A GitOps tool like Argo CD or Flux detects the change. It applies the manifest to your cluster. Kubernetes creates the pods. The application runs.

The next day, when your colleague asks how you configured that application. You point them to the Git repository. Every detail is documented. Every change is tracked. Every decision has a timestamp and an author.



When you need to create 50 similar applications, you write a template. Or you copy and modify the manifest. The process scales because the configuration exists as code.

When something breaks at 2 AM, you check Git to see what changed recently. You revert the problematic commit. The GitOps tool detects the reversion and rolls back automatically. Tomorrow, the incident review examines the Git history. The timeline is complete and accurate.

This is GitOps. Operations driven by Git repositories as the source of truth.

The Rise of Cloud Native

GitOps is one component of a larger movement. Cloud native computing has grown from a niche approach to the dominant paradigm for modern infrastructure.

The Cloud Native Computing Foundation now hosts over 180 projects. Kubernetes sits at the center, but the ecosystem extends far beyond container orchestration. Service meshes handle traffic management. Observability platforms collect metrics, logs, and traces. Policy engines enforce security and compliance. Each project solves a specific problem. Together, they form a complete infrastructure stack.

This ecosystem delivers advantages that traditional virtualization struggles to match.

Portability becomes real. A Kubernetes manifest runs on any conformant cluster. Your application deploys the same way on AWS, Azure, Google Cloud, or your own data center. Vendor lock-in decreases because the API remains consistent across providers. Organizations run workloads where economics and requirements dictate, not where infrastructure constraints force them.

Scalability becomes automatic. Kubernetes scales pods based on CPU, memory, or custom metrics. You define thresholds. The platform responds to demand without manual intervention. Applications handle traffic spikes without operators watching dashboards and clicking buttons. Scale-down happens automatically when demand subsides, reducing costs during quiet periods.

Resilience becomes built-in. Kubernetes restarts failed containers. It reschedules workloads when nodes fail. Health checks detect problems before users notice. Self-healing behavior runs continuously without operator involvement. The platform assumes failure will occur and handles it automatically.

Resource efficiency improves. Containers share operating system kernels. They start in seconds rather than minutes. Bin-packing algorithms place workloads optimally across nodes. Organizations run more applications on fewer servers. The density gains compound as container adoption increases.

Developer velocity increases. Teams ship changes faster when deployments are automated and rollbacks are simple. The feedback loop from code commit to production shrinks from days to minutes. Developers take ownership of their applications because the deployment process is accessible and repeatable.

These advantages explain why cloud native adoption accelerates year over year. Organizations that adopt Kubernetes report faster deployment cycles, improved uptime, and lower infrastructure costs. The benefits are measurable and documented across industries.

VMware served enterprises well for two decades. Cloud native infrastructure serves them better for the workloads and economics of today.

The Statefulness Question

You have a valid concern. Data persistence is the elephant in the room for any VMware professional evaluating Kubernetes.

VMware excels at protecting stateful workloads. vMotion migrates running VMs without downtime. High Availability restarts failed VMs on healthy hosts. vSAN replicates data across the cluster. These capabilities took years to mature. You trust them because you've seen them work in production for over a decade.

Kubernetes was originally designed for stateless applications. The early philosophy assumed containers were ephemeral. Pods came and went. Data lived elsewhere. This worked for web frontends and microservices. It failed for databases, message queues, and legacy applications that expected persistent local storage.

That limitation no longer exists.

The Container Storage Interface (CSI) standardized how Kubernetes interacts with storage systems. Every major storage vendor now ships CSI drivers. NetApp, Everpure, Dell EMC, and dozens of others provide enterprise-grade integration. Your existing SAN or NAS works with Kubernetes through CSI. The storage you trust already supports the platform you're learning.

Portworx represents the enterprise-grade answer to Kubernetes storage. It delivers software-defined storage built specifically for containers. Portworx replicates data across nodes, provides snapshots and backup, and enables disaster recovery across clusters. For VMware professionals, the capabilities feel familiar: high availability, data locality, and storage policies that follow your workloads. Portworx runs on any infrastructure, from bare metal to public cloud. It turns Kubernetes into a platform you can trust with your most demanding stateful applications.

Operators changed how Kubernetes manages complex applications. An Operator encodes operational knowledge in software. The PostgreSQL Operator handles database clustering, failover, and backup. The Kafka Operator manages broker scaling and partition rebalancing. These aren't scripts or runbooks. They're control loops that continuously reconcile desired state with actual state. The same pattern Kubernetes uses for pods now applies to stateful applications.

StatefulSets provide stable network identities and persistent storage for pods. When a database pod restarts, it reconnects to its existing storage and rejoins the cluster with its original hostname. The abstraction handles the complexity that vMotion handles in VMware environments.

GitOps manages configuration. CSI, Operators, and StatefulSets manage data. The combination delivers the data protection guarantees you expect from VMware. Chapter 5 of this series covers storage in depth. You'll see exactly how vSAN concepts translate to Kubernetes persistent volumes and how data services like snapshots and replication work in cloud native environments.

Your concern is valid. The answer is that Kubernetes has grown up and matured.

Why This Shift Matters for Your Career and Organization

The infrastructure industry is moving toward declarative, version-controlled operations. This isn't speculation. Job postings increasingly list Kubernetes as a requirement. Cloud providers offer managed Kubernetes services. Enterprise software vendors ship Kubernetes operators.

Your VMware skills remain relevant. Understanding compute clusters, distributed storage, and software-defined networking translates directly. The concepts match, even when the tools differ.

But you need to add new capabilities. You need to learn YAML manifests and kubectl commands. You need to understand Git workflows and CI/CD pipelines. You need to think in terms of desired state rather than manual steps.

This is an expansion, not a replacement. The administrator who understands both VMware and Kubernetes has more career options than one who knows only VMware. The market values people who bridge these worlds.

Your organization needs people who guide this transition. You understand the current state. You know which workloads run where and why. That institutional knowledge is irreplaceable during migration. Position yourself as the person who leads this change rather than the person who resists it.

Moving from ClickOps to GitOps requires cultural change. Teams accustomed to GUI-driven workflows must adopt new habits. The transition affects more than technology.

Change management becomes explicit. In ClickOps, changes happen immediately when someone clicks a button. In GitOps, changes go through pull requests. Colleagues review proposed modifications before they apply. This slows down individual changes but reduces incidents caused by hasty modifications.

Documentation becomes automatic. The Git repository serves as a living record of your infrastructure. New team members read the manifests to understand the environment. Auditors examine the commit history for compliance evidence. The documentation stays current because the documentation is the configuration.

Troubleshooting becomes traceable. When problems occur, you correlate symptoms with recent commits. The sequence of changes is recorded with precision. "What changed?" becomes an answerable question.

Rollbacks become routine. Reverting to a previous configuration means reverting to a previous commit. The process is identical whether you're rolling back a minor tweak or a major deployment. Recovery time decreases because recovery is automated.

These organizational benefits compound over time. The initial learning curve is real. Teams invest weeks or months developing GitOps proficiency. But that investment pays dividends for years to come as operational overhead decreases and consistency improves.

The Journey Ahead

The shift from ClickOps to GitOps represents the largest operational change in enterprise infrastructure since virtualization itself. That transition happened over a decade. This one is happening faster.

You have skills that transfer. You have knowledge that matters. The question is whether you'll add Kubernetes to your capabilities or watch colleagues do it instead.

Start small. Provision a managed Kubernetes cluster on your preferred cloud provider. AWS offers EKS. Azure provides AKS. Google Cloud runs GKE. Each provider offers free tiers or credits for learning. Deploy a simple application. Read the YAML manifest and understand what each line does. Commit the manifest to a Git repository.

The first step requires minimal investment. Most cloud providers let you run small clusters at low cost or no cost during trial periods. You begin learning this week.

What's next in this series?

Chapter 2 of this series introduces KubeVirt as your bridge between worlds. You'll see how your VM expertise applies directly to running virtual machines in Kubernetes. The skills you built over years of VMware work become assets rather than baggage.

The Broadcom acquisition forced a conversation many organizations had avoided for years. That conversation is now unavoidable. Participate in it with knowledge rather than uncertainty.

02

Chapter Two

KubeVirt

The Bridge Between Worlds

KubeVirt lets organizations adopt Kubernetes
without leaving their virtual machines behind.

You spent years building a reliable VM infrastructure. Your applications run on VMs. Your teams understand VMs. Your monitoring, backup, and security tools integrate with VMs. Nobody expects you to abandon all of that overnight.

Adopting Kubernetes doesn't mean abandoning virtualization entirely. You don't need to containerize every workload before progressing. There's a transition path that connects the VM environment you're familiar with to the container ecosystem you're exploring.

That bridge is KubeVirt.

KubeVirt extends Kubernetes to run virtual machines as first-class workloads alongside containers. Your existing VMs run inside the same platform that hosts your new containerized applications. One control plane. One set of tools. One operational model. Two workload types.

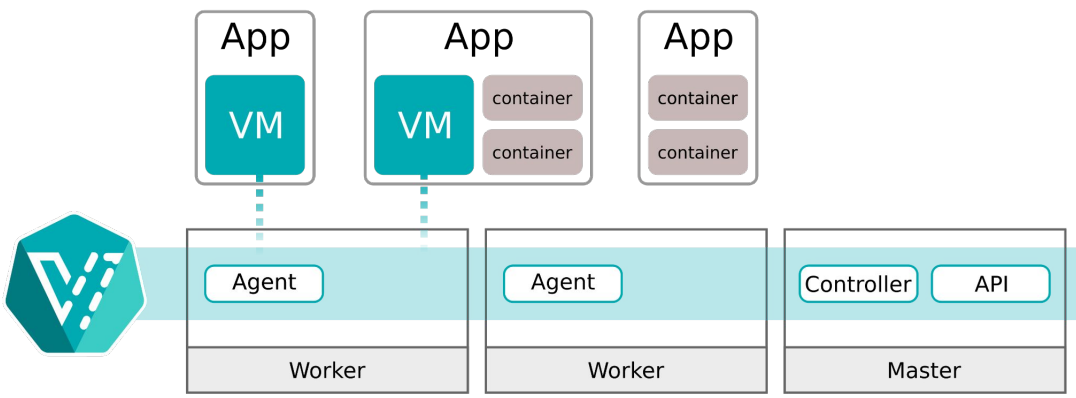
This matters because real migration takes years. You have legacy applications that nobody wants to refactor. You have commercial software that only ships as VM images. You have workloads where containers make no sense today and might never make sense. KubeVirt lets you move forward without leaving these workloads behind.

Why KubeVirt Exists

Kubernetes was designed for containers. The original architects assumed applications would be stateless, ephemeral, and containerized. Pods run containers. Containers package applications. The model works for microservices, web frontends, and modern applications built from the ground up.

Enterprises operate differently. Most organizations run thousands of VM workloads accumulated over 15 years of VMware deployments. These include database servers, Microsoft Windows applications, legacy middleware, and commercial software packages. These workloads remain the backbone of enterprise IT. They generate revenue. They support business operations. Telling a CFO that migration requires rewriting every application is not a conversation anyone is going to have successfully.

The industry recognized this gap. Red Hat, Intel, and other vendors collaborated to create KubeVirt under the Cloud Native Computing Foundation. The project launched in 2017 and reached general availability in subsequent years. Production deployments now run at scale across multiple industries.



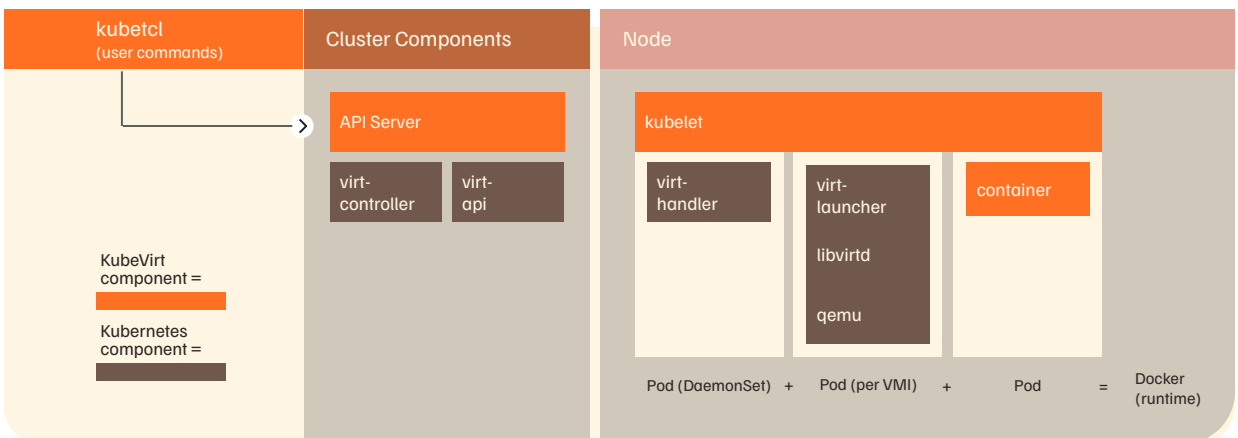
KubeVirt solves a practical problem. Organizations want Kubernetes benefits without abandoning their existing VM investments. They want one platform to manage instead of two. They want operational consistency across workload types. They want a transition path that respects their current reality while enabling their future goals.

How KubeVirt Works

KubeVirt installs as an add-on to any standard Kubernetes cluster. It introduces new Custom Resource Definitions (CRD) that teach Kubernetes how to understand and manage virtual machines. The VirtualMachine resource in KubeVirt defines a VM specification. The VirtualMachineInstance represents a running VM.

Under the hood, KubeVirt uses the same virtualization technology you trust. QEMU provides the emulation layer. KVM delivers hardware-accelerated virtualization on Linux hosts. These technologies power OpenStack, Proxmox, and countless enterprise virtualization platforms. The underlying hypervisor is mature and production-proven.

Each VM runs inside a Kubernetes pod. The pod hosts a container that runs the VM process. From the Kubernetes scheduler's perspective, this pod consumes CPU, memory, and storage like any other workload. The scheduler places VMs on nodes using the same logic it uses for containers. Resource requests, node affinity, and anti-affinity rules all apply.



This architecture delivers a critical benefit. Your VMs become Kubernetes-native resources. You manage them with kubectl or through the available dashboard UI in commercial platforms such as Red Hat OpenShift Virtualization. You define them in YAML manifests. You store those manifests in Git. The declarative model from Chapter 1 applies directly to your virtual machines.

Consider a simple VM definition

```
apiVersion: kubevirt.io/v1
kind: VirtualMachine
metadata:
  name: legacy-app-server
spec:
  running: true
  template:
    spec:
      domain:
        resources:
          requests:
            memory: 4Gi
            cpu: 2
        devices:
          disks:
            - name: root-disk
              disk:
                bus: virtio
      volumes:
        - name: root-disk
          persistentVolumeClaim:
            claimName: legacy-app-disk
```

The manifest at left declares a VM with 4GB of memory and 2 CPU cores. It references a persistent volume claim for storage. Apply this manifest with kubectl and Kubernetes to create your VM. Delete the manifest, and Kubernetes removes the VM. **The lifecycle seamlessly integrates with GitOps workflows we discussed in Chapter 1.**

Running VMs and Containers Together

The unified control plane changes how you think about workload placement. VMs and containers share the same cluster. They share the same network. They share the same storage infrastructure. They even share the same nodes if you configure them that way.

Network communication happens naturally. A containerized frontend can connect to a VM-based database using standard Kubernetes Service resources. The VM gets an IP address from the pod network. Service discovery works identically. A container calls the database service name, and Kubernetes routes traffic to the VM.

Storage works through the same PersistentVolume system. Your VM disks live on persistent volumes backed by your storage infrastructure. Portworx, Ceph, NetApp, and other storage providers work through CSI drivers. The VM sees a block device. Kubernetes handles the provisioning, attachment, and replication. Storage policies apply to VM workloads exactly as they apply to container workloads.

This convergence eliminates infrastructure silos. You no longer need separate VMware clusters and Kubernetes clusters. You no longer need separate storage pools for VMs and containers. You no longer need separate network configurations. One infrastructure serves all workloads.

The operational benefits compound over time. Your monitoring tools integrate with a single platform rather than two. Your backup strategy covers one environment instead of two. Your team learns one set of tools instead of two. Operational complexity decreases as workloads consolidate.

When to Use VMs vs Containers

KubeVirt gives you a choice. That choice requires a framework for deciding which workload type fits each application.

Use containers when the application is designed for them. Microservices, stateless web applications, and modern 12-factor apps belong in containers. Applications that start fast, scale horizontally, and tolerate instance replacement thrive in container environments. If your development team builds and tests in containers, deploy to containers as well.

Use VMs when the workload requires them. Windows Server applications often need VM environments. Commercial software that ships as VM images or ISO files needs VMs. Legacy applications with specific kernel requirements or driver dependencies need VMs. Applications that assume long-running processes and persistent local state often fit better in VMs.

Use VMs for lift-and-shift migrations. When you move workloads from VMware to Kubernetes, starting with VMs minimizes risk. The application runs in a familiar environment. The migration validates your Kubernetes infrastructure. The VM proves that storage, networking, and compute work correctly. Containerization becomes a later optimization rather than an upfront requirement.

Use containers for new development. When teams build new applications, guide them toward containers. The container ecosystem offers better tooling for modern development workflows. Container images are smaller and start faster. CI/CD pipelines integrate more naturally. New applications should adopt new patterns.

The decision often comes down to the effort required for migration versus the operational benefit. Containerizing a legacy application requires development work. Running that application as a VM on KubeVirt requires minimal changes. If the application works and nobody plans to modify it, the VM approach makes sense. If the application needs active development, investing in containerization pays dividends over time.

Managing Both Workload Types

Your existing Kubernetes skills apply directly to KubeVirt VMs. The kubectl commands you learn for containers work for virtual machines with minor variations.

kubectl get virtualmachines lists your VMs. kubectl describe virtualmachine shows configuration details. kubectl delete virtualmachine removes a VM. kubectl logs retrieves console output. The CLI experience remains consistent.

Helm charts can deploy VMs alongside containers. A single chart defines your application with its containerized components and VM dependencies. One deployment command provisions everything. One upgrade command updates everything. The application becomes a cohesive unit regardless of underlying workload types.

RBAC policies control access uniformly. You define roles that grant permissions to create, modify, or delete VMs. You bind those roles to users or service accounts. The authorization model matches what you use for container workloads.

Monitoring integrates through standard Kubernetes mechanisms. KubeVirt exports metrics in Prometheus format. CPU utilization, memory consumption, disk I/O, and network traffic all appear in your existing dashboards. Alerts fire based on the same rules you define for containers.

For VMware professionals, this offers a major simplification. You only need to manage one platform instead of two. Your skills transfer directly, and your tools integrate seamlessly. The operational approach stays consistent across all workloads.

The Path Forward

KubeVirt provides your transition path. It removes the requirement to containerize everything before adopting Kubernetes. It lets you migrate workloads incrementally. It preserves your VM investments while enabling container adoption.

Start by identifying candidates for early migration. Look for VMs that are relatively standalone. Development and test environments make good first targets. Non-critical internal applications reduce risk while you learn. Success with simple workloads builds confidence for complex ones.

Plan for eventual containerization where it makes sense. KubeVirt buys you time, but should not become a permanent destination for every workload. Applications under active development should move toward containers over time. Lift-and-shift gets workloads onto Kubernetes quickly. Refactoring to containers in the long term delivers the full benefits of cloud native architecture.

Some workloads will remain VMs indefinitely. Commercial software that ships only as VM images will remain on VMs. Legacy applications that nobody will ever modify will stay on VMs. This is acceptable. KubeVirt makes those VMs first-class citizens in your Kubernetes environment.

What’s next ?

The series continues with Chapter 3, where we map the entire VMware SDDC stack to its Kubernetes equivalents.

You will see how vSphere, vSAN, and NSX concepts translate to the cloud native world. The mental model you build will anchor every subsequent discussion of compute, storage, networking, and security.

KubeVirt bridges your present to your future. Use that bridge to move forward at a pace your organization can sustain. The destination is cloud native infrastructure. The journey accommodates your existing reality.

03

Chapter Three

Mapping the Stack

SDDC to Cloud Native

Kubernetes is infrastructure
by composition, not by monolith.

You built your career on VMware's Software Defined Data Center. You know how vSphere, vSAN, and NSX work together. You understand the management layer. The components make sense because you have deployed, configured, and troubleshot them for years.

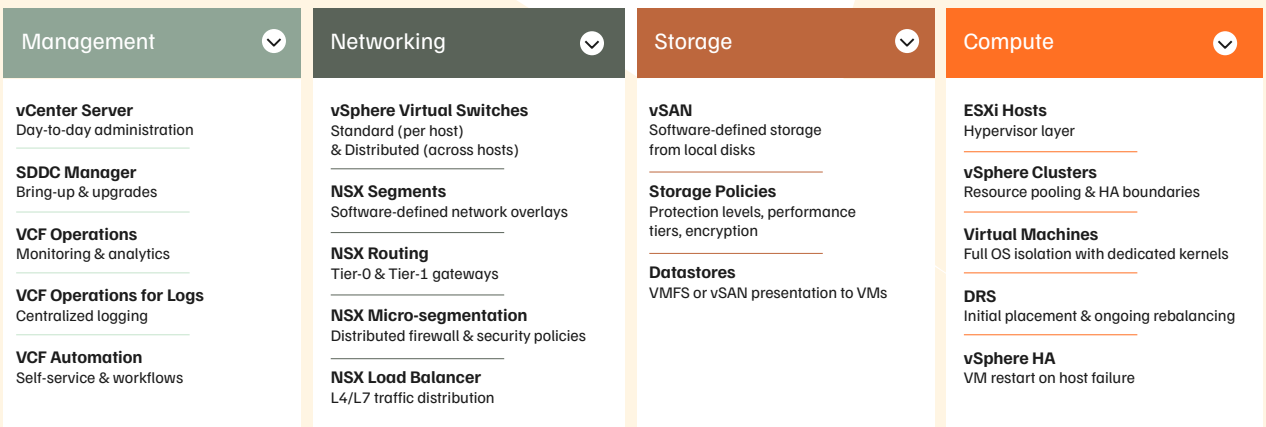
Kubernetes has equivalent components for almost every layer of your VMware stack. The names change. The interfaces change. The underlying concepts and the core building blocks remain remarkably familiar.

This chapter provides the mental map you need. We will walk through VMware Cloud Foundation layer by layer and identify what each component is replaced by in the cloud native world. Once you see the parallels, everything else in this series will click into place.

The VMware Cloud Foundation Stack

VMware Cloud Foundation represents the complete Software Defined Data Center (SDDC). VCF 9.0 bundles compute, storage, networking, and management into a unified platform. You deploy VCF as an integrated stack rather than assembling individual products.

The stack has four primary layers. Management, Networking, Storage, and Compute.



Compute sits at the foundation. ESXi hosts form clusters managed by vCenter Server. VMs run on these hosts. vCenter handles provisioning, resource management, and lifecycle operations. Distributed Resource Scheduler (DRS) automatically balances workloads across hosts, including moving running VMs to fix hotspots. vSphere HA restarts VMs when hosts fail.

Storage builds on top of compute. vSAN aggregates local storage from ESXi hosts into a shared datastore. Storage policies define protection levels, performance tiers, and capacity allocation. VMs consume storage through VMFS or vSAN datastores without knowing which physical disks hold their data.

Networking connects everything. vSphere virtual switches provide Layer 2 connectivity: standard switches per host, distributed switches across hosts. NSX adds software-defined networking and security on top, including segmentation, routing, microsegmentation, and integrated network services. This layered approach separates basic connectivity from advanced network virtualization.

Management ties the layers together. SDDC Manager orchestrates bring-up and upgrades, but in VCF 9, its UI is deprecated, and many workflows surface in VCF Operations and the vSphere Client. VCF Operations provides monitoring and operational analytics with integrated log analysis. VCF Operations for Logs provides centralized log management. VCF Automation provides self-service and workflow automation. vCenter serves as the primary interface for day-to-day administration.

This architecture works. Enterprises run production workloads on VCF because the integration is proven and the operational model is well understood.

The Kubernetes Cloud Native Stack

Kubernetes organizes infrastructure differently. Instead of having a single vendor provide all components, the cloud native ecosystem relies on interchangeable parts that follow open standards.

The stack has the same four layers. The components change.

Management	Networking	Storage	Compute
Control Plane API server, controllers, scheduler, etcd	**CNI Plugins** Calico, Cilium, Flannel for pod connectivity	**SDS Backends** Portworx, Rook-Ceph, Longhorn	**Nodes** Servers running kubelet & container runtime
Cluster API / Platform Managers Infrastructure lifecycle management	**Services** Stable virtual IPs & internal discovery	**CSI Drivers** Interface connecting K8s to storage backends	**Clusters** Control plane + worker nodes
Prometheus Metrics collection & alerting	**Gateway API** L4/L7 traffic routing, TLS termination	**Storage Classes** Provisioner parameters & tier definitions	**Pods** Container groups w/shared network/storage
Grafana Visualization & dashboards	**Network Policies** Pod traffic rules (requires CNI support)	**Persistent Volumes (PV)** Storage resources in the cluster	**Deployments / StatefulSets** Controllers managing pod lifecycle
Centralized Logging Loki, Elasticsearch, Fluentd	**Service Mesh** Istio, Linkerd for advanced traffic mgmt.	**Persistent Volume Claims (PVC)** Application storage requests	**Scheduler** Places new pods on appropriate nodes
GitOps Tools Argo CD, Flux for declarative delivery			**Descheduler** Optional rebalancing by evicting pods

Compute in Kubernetes means nodes and pods. Nodes are servers running the kubelet agent and a container runtime. Pods are groups of containers on a node that share network and storage. The Kubernetes scheduler places pods on nodes based on resource requirements and constraints. When nodes fail, controllers like Deployments or StatefulSets create replacement pods. The scheduler then places those new pods on healthy nodes.

Storage uses the Container Storage Interface (CSI) specification. AWS EBS driver, Portworx driver, and NFS CSI drivers are examples of these. CSI is an interface specification that connects Kubernetes to storage backends. Persistent Volumes represent storage resources. Persistent Volume Claims let applications request storage without knowing the underlying implementation. Storage Classes define tiers with different characteristics, passed as parameters to the storage provisioner.

Networking follows the Container Network Interface (CNI) specification. CNI plugins like Calico, Cilium, or Flannel provide pod networking. Services expose applications internally and provide stable virtual IPs. The recently added Gateway API handles external HTTP/HTTPS traffic, including routing and SSL termination. Network Policies control traffic flow between pods, though enforcement depends on your CNI plugin supporting them.

Management spans multiple tools. The Kubernetes control plane includes the API server as the front door, controllers for reconciliation, the scheduler for placement, and etcd as the backing state store. Prometheus collects metrics. Grafana visualizes data. GitOps tools like Argo CD or Flux manage application and configuration deployments declaratively. Platform engineering teams often build internal developer platforms on top of these components.

No single vendor owns this stack. You choose components based on your requirements. The interfaces between components follow open specifications, so you can swap implementations without rebuilding your entire platform.

The Conceptual Mapping

Here is how VMware components map to their Kubernetes equivalents. These mappings provide mental anchors, not exact equivalences. Each platform has distinct design philosophies, making perfect one-to-one comparisons impossible.

vSphere Clusters map to Kubernetes Clusters. Both groups compute resources into manageable units. Both support workload isolation and resource pooling. The difference: Kubernetes clusters include control plane services and an API contract. vSphere clusters are primarily compute constructs with HA and DRS behavior.

ESXi Hosts map to Kubernetes Nodes. Both provide the compute substrate. ESXi runs a hypervisor that creates hardware abstraction. Kubernetes nodes run a container runtime, such as containerd. The kubelet agent on each node reports status and accepts work from the control plane. The kubelet is an agent, not a virtualization layer.

Virtual Machines map loosely to Pods, but the analogy has limits. VMs provide full operating system isolation with separate kernels. Pods share the host kernel and isolate processes through namespaces and cgroups. The isolation model differs significantly. Pods are typically a part of controllers such as Deployments and StatefulSets. These controllers own pods and manage their lifecycle.

vCenter maps to the Kubernetes Control Plane. vCenter provides the management plane for vSphere. The Kubernetes equivalent includes multiple components working together: the API server as the entry point, controllers that reconcile desired state with actual state, the scheduler that places workloads, and etcd that stores cluster state. Comparing vCenter to the API server alone oversimplifies the architecture, but it does help with the mental model.

DRS maps to Scheduling plus Rebalancing. DRS handles initial VM placement and ongoing balancing. It moves running VMs to fix resource hotspots. Kubernetes scheduling works differently. The scheduler selects nodes for new or unscheduled pods. Kubernetes does not continuously rebalance running pods by default. If you want DRS-like rebalance behavior, you add a Descheduler. The Descheduler evicts pods based on policies, and the scheduler then places them on better-suited nodes.

vSAN maps to Kubernetes-native software-defined storage backends exposed via CSI. CSI itself is an interface standard, not a storage system. The actual storage comes from backends such as Portworx, Rook Ceph, or Longhorn. These solutions aggregate and manage storage. CSI provides the integration layer that exposes storage capabilities to pods.

Storage Policies map to Storage Classes. Both define storage tiers with specific characteristics. A VMware storage policy might specify RAID level, encryption, and performance tier. A Kubernetes Storage Class specifies a provisioner and parameters. Features like replication and encryption depend on the underlying storage backend. The Storage Class passes parameters to the provisioner, which translates them into backend-specific configurations. Portworx, for example, uses a *repl* parameter to set the replication factor (up to three copies across nodes), similar to vSAN's *Failures to Tolerate* setting. For encryption, Portworx accepts a *secure: "true"* parameter that enables AES-256 volume encryption, with key management through Kubernetes Secrets or external providers like Vault, AWS KMS, or Google KMS.

vSphere Virtual Switches map to CNI Plugins for basic connectivity. Both provide the foundational network layer. Standard switches and distributed switches in vSphere give VMs Layer 2 connectivity. CNI plugins give pods network connectivity within and across nodes.

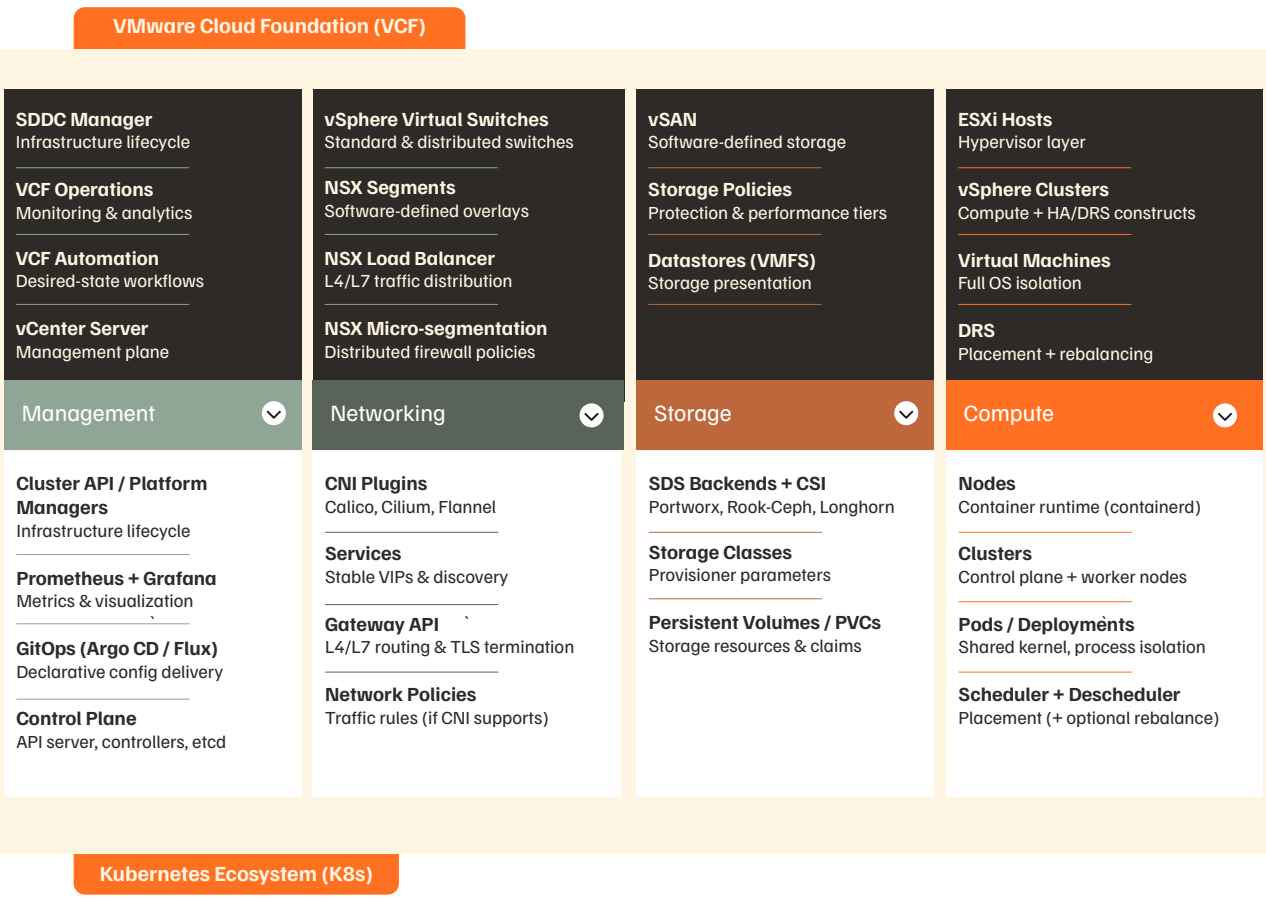
NSX maps to Network Policies plus advanced networking features. NSX adds segments, routing, micro-segmentation, and load balancing on top of vSphere networking. In Kubernetes, Network Policies define traffic rules between pods. Service meshes like Istio or Linkerd add advanced traffic management, observability, and security. One caveat: Network Policy enforcement requires a CNI plugin that supports it. Not all do, but if you choose a plugin like Cilium or Calico, you can implement fine-grained network policies.

NSX Load Balancer maps to Services plus Gateway API. Both distribute traffic to backend workloads. Kubernetes Services provide stable virtual IPs and internal discovery. Service type LoadBalancer integrates with external load balancers. Gateway API is for Layer 4 and Layer 7 HTTP/HTTPS routing. The separation of concerns differs from NSX's integrated approach.

VCF Operations maps to Prometheus and Grafana. VCF Operations provides monitoring, operational analytics, and integrated log analysis. Prometheus and Grafana fill that role in Kubernetes. VCF Operations for Logs has an equivalent in centralized logging solutions such as Loki, Elasticsearch, or Fluentd.

VCF Automation maps to GitOps tools. VCF Automation handles desired-state delivery and self-service workflows. GitOps tools like Argo CD provide similar declarative configuration management for applications and cluster add-ons.

SDDC Manager maps to Infrastructure Lifecycle Management tools. SDDC Manager handles bring-up, upgrades, and lifecycle of VCF stack components. The Kubernetes equivalent is cluster lifecycle management via tools such as Cluster API (CAPI) or vendor-specific platform managers. GitOps handles application and configuration delivery, a different layer of the problem.



The Simplified Stack Comparison

Think of the stacks as parallel structures with equivalent layers, while respecting their architectural differences.

At the compute layer, ESXi hosts running VMs map to Kubernetes nodes running pods. The isolation boundary moves from a hypervisor to the host OS kernel (namespaces and cgroups), with the container runtime managing packaging and execution.
At the storage layer, vSAN with storage policies maps to software-defined storage backends exposed through CSI with Storage Classes. Data services such as snapshots, replication, and encryption are available in both worlds, but their implementation depends on your chosen storage backend.

At the networking layer, vSphere switches plus NSX map to CNI plugins combined with the Gateway API and Network Policies. Kubernetes separates what VMware layers (basic connectivity in vSphere, advanced services in NSX). Network segmentation moves from distributed port groups and NSX segments to namespaces and Network Policies.

At the management layer, vCenter maps to the Kubernetes control plane. SDDC Manager maps to cluster lifecycle tools. VCF Operations and Automation map to observability and GitOps tooling. The operational model shifts from GUI-driven configuration to declarative manifests stored in Git.

Seven or eight component areas in VMware have counterparts in Kubernetes.
The abstraction levels align at a conceptual level. The implementations and design philosophies differ in important ways that this series will explore.

Why This Mapping Matters

Understanding these parallels accelerates your learning. You already know what DRS does. Learning that Kubernetes scheduling handles initial placement while the Descheduler handles rebalancing takes less effort than learning scheduling concepts from scratch.

The mapping also helps you evaluate solutions. When a vendor pitches a Kubernetes storage product, you can assess it against your vSAN experience. Does it support snapshots? What about replication? How does it handle node failures? Your VMware knowledge frames the right questions. Just remember that features depend on the storage backend, not the CSI interface.

What’s next ?

Most importantly, this mapping provides context for everything that follows in this series. Chapter 4 will explore compute in depth. Chapter 5 covers storage. Chapter 6 addresses networking.

Each part builds on this foundation, showing you how familiar concepts translate to cloud native implementations while highlighting where the platforms diverge.

Your SDDC expertise transfers. The nuances matter. The next step is learning both the new approach and the new mental models.

04

Chapter Four

Compute

From ESXi Hosts to Kubernetes Nodes

For VMware administrators, K8s compute is not foreign— it's reframed.

You understand compute virtualization. ESXi hosts run your VMs. Clusters pool those hosts together. DRS balances workloads automatically. vCenter provides centralized management. These concepts translate directly to Kubernetes.

The terminology changes. The operational model shifts. The underlying principles remain exactly the same: abstract physical resources, schedule workloads intelligently, and provide high availability when things fail.

This chapter maps your ESXi and vSphere knowledge to Kubernetes compute concepts. By the end, you will understand how nodes, pods, and containers relate to the infrastructure you already manage. You will see where DRS and the Kubernetes scheduler align and where they differ.

The Foundation: ESXi Hosts vs Kubernetes Nodes

An ESXi host runs the hypervisor that enables virtualization. Physical CPU, memory, network, and storage become pooled resources available to virtual machines. You manage ESXi hosts through vCenter.

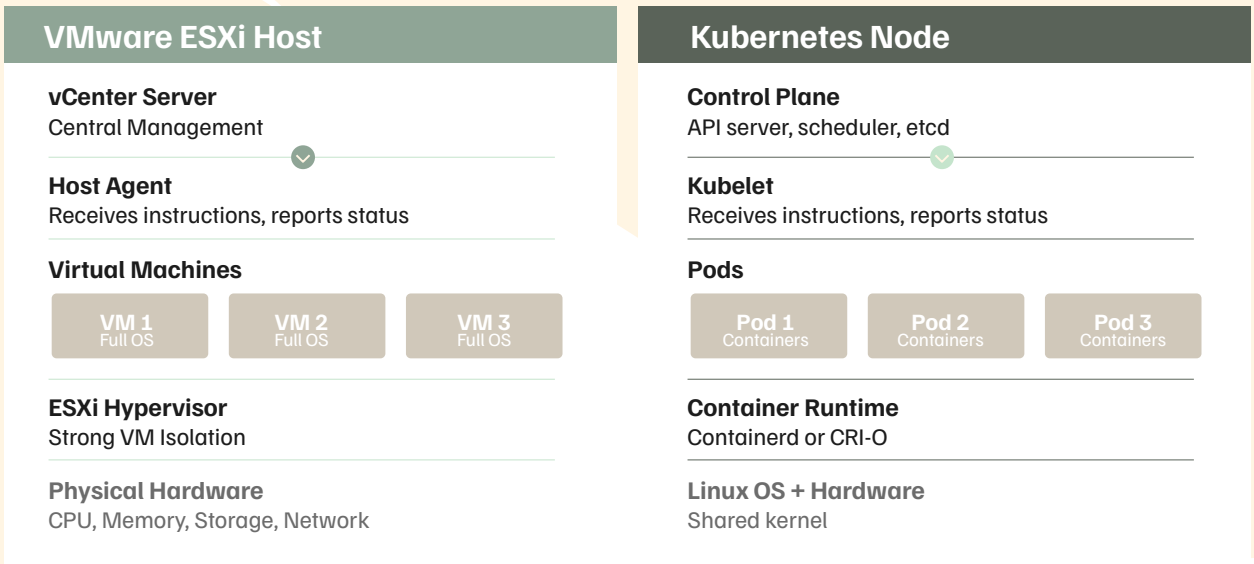
A Kubernetes node serves the same foundational role. Nodes are machines that run containerized workloads. They provide CPU, memory, and storage to pods. Each node runs a few key components that make this possible.

The kubelet runs on every node. This agent communicates with the Kubernetes control plane, receives instructions about which pods to run, and reports node status back to the cluster. The kubelet is to a node what the ESXi host agent is to vSphere. It makes the node manageable.

Every node also runs a container runtime. These runtimes pull container images and start the actual containers that run your applications. The container runtime sits below Kubernetes in the stack, similar to how the hypervisor sits below vCenter.

A third component, kube-proxy, handles Service routing on each node. It maintains network rules that direct traffic to the correct pods when you access a Service's virtual IP. Think of it as the Service proxy that makes Service VIPs and load balancing work. Some CNI plugins like Cilium replace kube-proxy entirely with their own implementation. We will cover networking in depth in Chapter 6.

When you add an ESXi host to a vSphere cluster, vCenter discovers its resources and makes them available for VM placement. Kubernetes works similarly. When a node joins a cluster, the kubelet registers with the control plane. The scheduler then considers that node when placing new workloads.



The Workload: VMs vs Pods vs Containers

Understanding the relationship between VMs, pods, and containers requires careful attention. These are different levels of abstraction with different isolation characteristics.

A virtual machine on ESXi has its own operating system. The hypervisor provides strong isolation between VMs. Each VM boots its own kernel, runs its own init system, and manages its own processes. You can run Windows and Linux VMs side by side on the same ESXi host because each VM is fully isolated.

Containers share the host operating system kernel. A container includes your application and its dependencies, but not a full operating system. Multiple containers on the same node all use that node's Linux kernel. This makes containers lighter than VMs but provides weaker isolation.

A pod is a Kubernetes abstraction that groups one or more containers. Containers within a pod share the same network namespace and can communicate over localhost using IPC and ITS protocols. They also share storage volumes. Pods are the smallest deployable unit in Kubernetes.

For VMware administrators, the most useful mental model is this: think of a pod as the Kubernetes equivalent of "the thing you deploy and scale." You deploy VMs in vSphere. You deploy pods in Kubernetes.

The abstraction levels differ in practice. A single pod might run one container for your main application and another for logging. These containers start and stop together. They share an IP address. This sidecar pattern has no direct equivalent in the VM world.

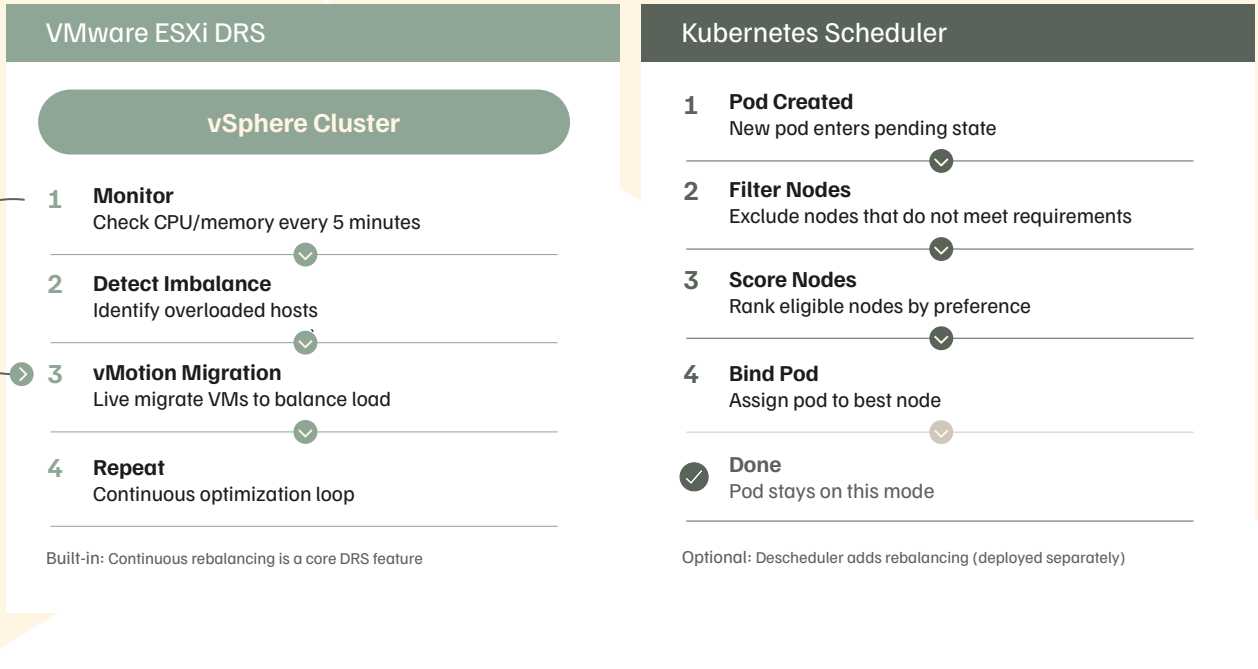
Most production workloads run pods through higher-level controllers. Deployments manage stateless applications and ensure your desired number of replicas run at all times. StatefulSets manage stateful applications with stable network identities and persistent storage. These controllers are closer to how you think about deploying and scaling workloads than raw pods.

Resource Scheduling: DRS vs the Kubernetes Scheduler

DRS monitors CPU and memory utilization across your vSphere cluster. When it detects an imbalance, it migrates VMs between hosts using vMotion. The goal is to balance resource distribution and prevent hotspots.

The Kubernetes scheduler handles initial pod placement. When you create a pod, the scheduler evaluates all available nodes and selects the best one based on resource requests, constraints, and scoring rules.

A critical difference exists here. DRS provides ongoing workload balancing through live migration. The Kubernetes scheduler only places pods at creation time. Once a pod runs on a node, the default scheduler does not move it.



This distinction surprises many VMware administrators. DRS continuously optimizes your cluster. The Kubernetes scheduler makes a decision once and leaves pods in place.

Kubernetes addresses ongoing optimization differently. The Descheduler project provides cluster rebalancing capabilities. It runs as an optional component, evaluates pod placement against policies, and evicts pods that should move. When the Descheduler evicts a pod, the controller creates a replacement, and the scheduler places it on a better node.

The Descheduler is not a standard Kubernetes component. You deploy it separately if you need rebalancing. Many managed Kubernetes services provide built-in alternatives. The important point is that Kubernetes separates initial placement from ongoing optimization.

Both systems consider resource requests when scheduling. In vSphere, you configure VM reservations and limits for CPU and memory. In Kubernetes, you specify resource requests and limits in pod specifications.

Requests tell the scheduler how much CPU and memory your pod needs. The scheduler will not place a pod on a node unless that node has enough unrequested capacity. This prevents overcommitment at the scheduling layer. Limits define the maximum resources a container can consume. If a container exceeds its memory limit, Kubernetes terminates it.

Requests impact placement and quality of service. Kubernetes scheduler honors the requests so that the requested CPU and memory fit, and the runtime enforces isolation with cgroups. But this is not the same as vSphere reservations that carve out untouchable capacity. CPU is compressible and managed through the Completely Fair Scheduler (CFS) shares and throttling. Memory requests affect scheduling and eviction priority but do not create dedicated physical partitions. Under pressure, Kubernetes evicts lower-priority pods rather than guaranteeing reserved capacity the way vSphere does.

Affinity and anti-affinity rules exist in both systems. DRS supports VM-VM and VM-Host rules that keep certain workloads together or apart. Kubernetes provides pod affinity, pod anti-affinity, and node affinity. You can require that two pods run on the same node, require they run on different nodes, or prefer certain node characteristics without mandating them.

Topology spread constraints in Kubernetes distribute pods across failure domains. You can ensure pods spread evenly across availability zones or nodes. This capability closely resembles VM-Host anti-affinity rules, which distribute VMs across hosts to improve availability.

The Control Plane: vCenter vs Kubernetes API Server

vCenter Server is your central management point for vSphere. It maintains the inventory of hosts, clusters, VMs, and networks. It provides the interface for configuration and operations. vCenter runs DRS, manages vMotion, and stores configuration in its database.

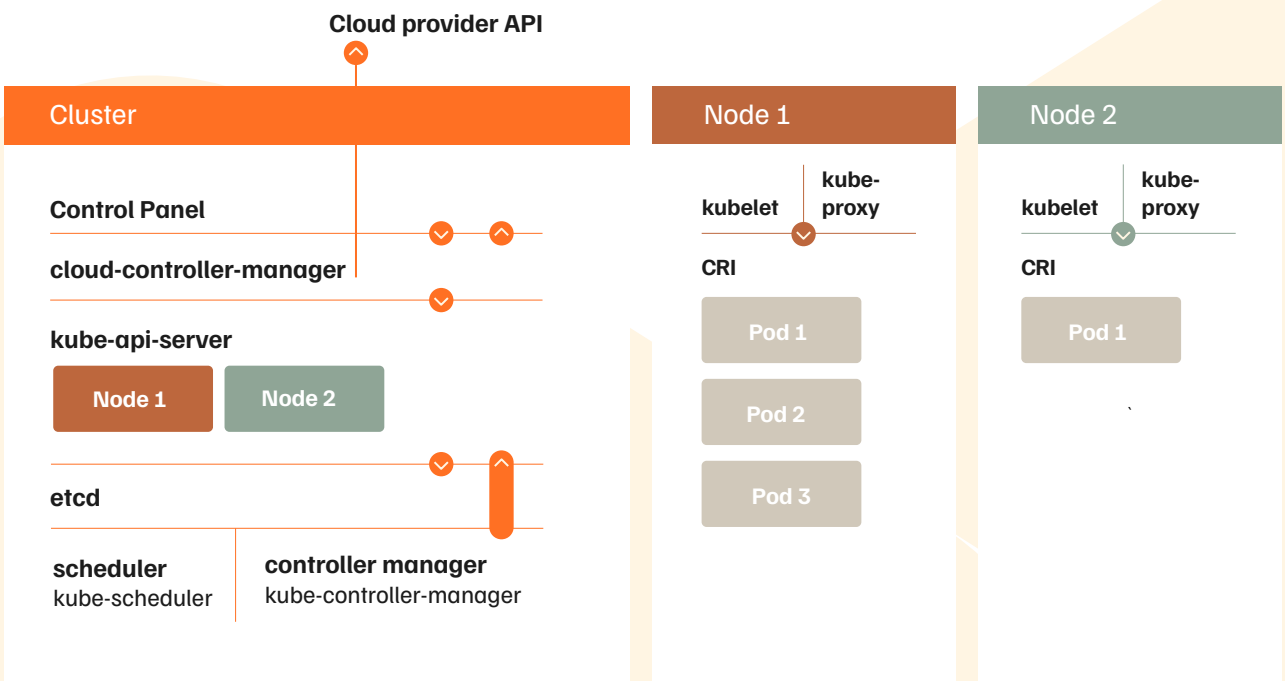
Kubernetes distributes these responsibilities across multiple control plane components.

The API server is the frontend for all cluster operations. Every kubectl command, every controller action, and every kubelet status update flows through the API server. It validates requests, notifies interested components of changes, and serves as the gateway to cluster state.

etcd stores all cluster state. This distributed key-value store holds the definitions of every pod, service, deployment, and configuration in your cluster. The API server reads from and writes to etcd. Without etcd, the cluster loses its memory.

The scheduler watches for unscheduled pods and assigns them to nodes. It functions as a single-purpose component focused on placement decisions.

Controller managers run control loops that reconcile the desired state with the actual state. When you create a Deployment requesting three pod replicas, the Deployment controller creates three pods. When a node fails, the node controller marks it as unavailable, and other controllers create replacement pods elsewhere.



This architecture differs from vCenter's monolithic approach. vCenter combines inventory management, scheduling, migration, and operations into a single server. Kubernetes separates these concerns into components that communicate through the API server.

High availability patterns also differ. vCenter HA requires vCenter High Availability configuration with witness nodes. Kubernetes control planes achieve HA by running multiple replicas of each component behind a load balancer, with etcd running as a clustered store.

In managed Kubernetes services like EKS, AKS, and GKE, the cloud provider operates the control plane. You interact with the API server without managing etcd or the scheduler directly. This resembles how VMware Cloud on AWS abstracts vCenter management from customers. Note that VMware Cloud on AWS is now available directly from Broadcom rather than through AWS reselling.

Putting the Pieces Together

Your ESXi cluster has hosts managed by vCenter. DRS schedules VMs onto hosts and rebalances them over time. Each VM runs a full operating system isolated by the hypervisor.

Your Kubernetes cluster has nodes that register with the control plane. The scheduler places pods onto nodes at creation time. Each pod runs one or more containers that share the host kernel.

The kubelet on each node acts as the local agent, similar to the ESXi host agent. The container runtime runs below the kubelet, providing the actual container execution.

Resource management uses requests and limits rather than reservations and limits. Requests drive scheduling and QoS classification, with cgroups enforcing isolation at runtime. Scheduling rules use affinity, anti-affinity, and topology spread constraints instead of DRS rules.

The control plane distributes responsibilities across components rather than centralizing them in vCenter. etcd stores state. The API server serves as the frontend for all communication. The scheduler handles placement. Controllers handle reconciliation.

These differences matter for operations. Kubernetes expects workloads to fail and restart. Pods are ephemeral by default. The system constantly reconciles the actual state with the desired state. Your VM mindset treats workloads as long-lived and stable. Kubernetes assumes they will move, restart, and scale.

What's next ?

Chapter 5 examines storage. You will learn how vSAN concepts translate to persistent volumes, how storage policies become storage classes, and how the Container Storage Interface standardizes storage integration across providers.

05

Chapter Five

Storage

From vSAN to Container-Native Storage

The storage principles you trust in VMware still apply—Kubernetes just expresses them differently.

Storage is the concern that keeps VMware administrators up at night when they think about Kubernetes. You spent years building a reliable storage infrastructure. vSAN aggregates local disks into shared datastores. Storage policies enforce protection levels. Snapshots and replication protect your data. Everything works because the stack is integrated and the operational model is proven.

Kubernetes handles storage differently. Containers are ephemeral by default. When a pod restarts, its local filesystem disappears. This design works well for stateless web frontends and API services. It falls apart the moment you need a database, a message queue, or any application that writes data to disk.

The good news: Kubernetes has a mature storage framework that maps to concepts you already know. Datastores become persistent volumes. Storage policies become storage classes. vSAN becomes a software-defined storage backend that exposes capacity through a standard interface. The abstractions are different. The goals are identical.

This chapter walks through each VMware storage concept and shows you where it lands in the Kubernetes world.

Datastores and VMFS vs Persistent Volumes and Claims

In vSphere, a datastore is an abstraction over physical storage. VMFS datastores sit on top of block storage from a SAN. vSAN datastores aggregate local disks from ESXi hosts into a distributed storage pool. NFS datastores present file-based storage from a NAS appliance. VMs consume storage from these datastores through virtual disks (VMDKs) without knowing or caring about the underlying hardware.

Kubernetes uses a two-part abstraction: Persistent Volumes (PVs) and Persistent Volume Claims (PVCs).

A PV represents a storage resource in the cluster. Think of it as the Kubernetes equivalent of a datastore LUN or a vSAN object. A PV has a specific capacity, access mode, and backend type. The cluster administrator or a storage provisioner creates PVs to make storage available.

A PVC is a storage request made by an application. Think of it as the equivalent of creating a VMDK on a datastore. A developer writes a PVC that says, "I need 50GB of fast block storage." Kubernetes finds a matching PV and binds the two together. The application gets its storage without knowing which backend provided it.

This separation of concerns is intentional. Cluster administrators manage PVs and storage backends. Application developers create PVCs. Neither needs to understand the other's domain in detail. In vSphere, a VM administrator works with datastores that a storage administrator provisioned. The model is the same.

Storage Abstraction Comparison

	VMware / vSphere	Kubernetes
Workload	**VM** Consumes VMDK	**Pod** Mounts PVC
Volume	**VMDK** Virtual disk on datastore	**PV / PVC** Persistent Volume + Claim
Storage Layer	**Datastore (VMFS / vSAN / NFS)** Admin-provisioned storage pool	**StorageClass + CSI Driver** Dynamic provisioning on demand
Infrastructure	**Physical Disks / SAN / NAS** Hardware or array	**Physical Disks / Cloud Volumes** Local, array, or cloud block

The critical difference lies in dynamic provisioning of volumes in Kubernetes. In vSphere, datastores exist before VMs request storage. Someone creates the datastore, configures VMFS, and presents it to the cluster. Kubernetes supports this static provisioning model, but most production environments use dynamic provisioning instead. When a PVC is created, and no existing PV matches it, Kubernetes tells the storage backend to create one on the fly. The PV appears, binds to the PVC, and the pod gets its volume. No manual intervention required.

Dynamic provisioning changes the operational model. You stop pre-creating storage and start defining policies that describe what types of storage are available. This is where storage classes come in.

Storage Policies vs Storage Classes

In vSAN, storage policies define the characteristics of your storage. You create a policy that specifies the number of failures to tolerate (FTT), the failure tolerance method (RAID-1 mirroring or RAID-5/6 erasure coding), stripe width, and IOPS limits. When you provision a VM, you assign a storage policy. vSAN ensures the data meets those requirements.

VCF 9.0 extends this model with Storage Policy Based Management (SPBM), which provides a unified policy framework across vSAN, VMFS, and NFS datastores. You define what you need. The platform delivers it.

Kubernetes storage classes serve the same purpose. A StorageClass is a declarative object that defines a type of storage available in the cluster. It specifies which provisioner creates the volumes, which parameters to use, and the reclaim policy that applies when the PVC is deleted.

Here is a simplified comparison.

In vSAN, you create a storage policy with FTT=1, RAID-1 mirroring, and assign it to a VM. The VM gets replicated storage that survives a single host failure.

In Kubernetes, you create a StorageClass that points to a storage backend, specifies a replication factor of 3 and a fast SSD tier, and sets the reclaim policy to delete. When a PVC references this StorageClass, the backend provisions a volume that matches those parameters.

vSAN Storage Policy	Kubernetes StorageClass
Failures to Tolerate (FTT) FTT = 1	**Replication Factor** repl: 3
Failure Tolerance Method RAID-1 Mirroring	**Provisioner Config** Backend-specific replication
Storage Tier All-Flash / Hybrid	**Tier Parameter** tier: ssd
IOPS Limit 5000 IOPS	**QoS / Bandwidth** Provisioner-specific QoS
RECLAIM BEHAVIOR	
Remove from Inventory VM files stay on datastore	**Retain** PV and data kept for manual cleanup
Delete from Disk VM and VMDKs removed	**Delete** PV and data removed when PVC deleted

The mapping is direct. Storage policy name becomes StorageClass name. Protection parameters become provisioner-specific parameters. Assign-on-create works the same way. The PVC references a StorageClass the way a VM references a storage policy.

One key aspect for operations is reclaim policies. In vSphere, choosing "Remove from Inventory" retains all VM files on the datastore, including VMDKs. "Delete from Disk" permanently removes the VM and its files. Kubernetes gives you two options. "Delete" removes the PV and its data when the PVC is deleted. "Retain" keeps the PV and data for manual cleanup. Choose based on your data lifecycle requirements. Production databases should use "Retain" to prevent accidental data loss.

Storage classes also support volume binding modes. "Immediate" creates the volume immediately when the PVC is created. "WaitForFirstConsumer" delays volume creation until a pod needs it. The second mode is important for topology-aware storage, where the volume needs to be created on the same node or in the same fault domain, such as a zone, as the pod that will use it.

The Container Storage Interface (CSI)

In vSphere, storage integration happens through VAAI (vStorage APIs for Array Integration) and VASA (vStorage APIs for Storage Awareness). These APIs enable vSphere to communicate with storage arrays to offload operations such as cloning, snapshotting, and thin provisioning. The storage vendor writes a VAAI/VASA provider. vSphere talks to the provider. The array does the work.

Kubernetes uses the Container Storage Interface (CSI) to achieve the same goal. CSI is an open standard that defines how container orchestrators interact with storage systems. A storage vendor writes a CSI driver. Kubernetes talks to the driver. The storage system does the work.

Before CSI, Kubernetes storage drivers were compiled into the Kubernetes binary itself. Adding support for a new storage system required changes to the core Kubernetes codebase. CSI moved storage drivers out of the core and into independent, pluggable components. Storage vendors now ship their own CSI drivers and update them on their own release cadence.

This is similar to how VMware moved from in-box drivers to the modern VAAI plugin model. The storage vendor owns the integration layer. The platform provides the interface. Neither needs to ship on the other's schedule.

Every major storage vendor provides a CSI driver. Cloud providers ship CSI drivers for their managed block and file storage services. Everpure, NetApp, Dell, HPE, and dozens of others provide CSI drivers for their hardware arrays. Software-defined storage platforms like Portworx by Pure Storage, Rook (for Ceph), and Longhorn include CSI drivers as part of their deployment.

A CSI driver handles volume creation, deletion, attachment, detachment, mounting, snapshotting, cloning, expansion, and health monitoring. The driver runs as pods in your Kubernetes cluster. When Kubernetes needs a storage operation, it calls the CSI driver through gRPC. The driver translates the request into the format required by the backend storage system.

For VMware professionals, CSI replaces the combination of VAAI, VASA, and the vSphere storage stack's native integrations. The difference is that CSI is vendor-neutral and works with all Kubernetes distributions. Your CSI driver works the same way on Red Hat OpenShift, Rancher, EKS, AKS, and GKE.

Software-Defined Storage in Kubernetes

vSAN is software-defined storage purpose-built for vSphere. It aggregates local disks from ESXi hosts into a distributed storage pool, replicates data according to storage policies, and delivers enterprise data services such as snapshots, encryption, and compression. vSAN ESA (introduced with vSAN 8) is optimized for modern NVMe hardware, delivering improved efficiency, scalability, and performance. VCF 9.0 (GA June 17, 2025) builds on ESA with features such as vSAN-to-vSAN replication for disaster recovery and disaggregated storage clusters that separate compute from storage.

Kubernetes does not ship with a built-in storage layer. You choose a software-defined storage platform that runs on top of your cluster. Several mature options exist, each with different strengths.

Portworx by is a Kubernetes data services platform built for enterprise production environments. Portworx Enterprise provides persistent storage, high availability across availability zones, automated data placement, and encryption. Portworx runs on any infrastructure: bare metal, cloud VMs, or enterprise storage arrays from Everpure and others. For organizations running KubeVirt to bridge VMs and containers (as covered in Chapter 2), Portworx provides unified storage and data management across both workload types. It supports ReadWriteMany block storage for KubeVirt VMs, synchronized disaster recovery with zero data loss (zero RPO), and file-level backups for Linux VMs on Kubernetes.

Rook is a CNCF Graduated project that orchestrates Ceph storage on Kubernetes. Ceph is a distributed storage system that provides block, file, and object storage. Rook deploys Ceph as a Kubernetes-native operator that manages the Ceph cluster lifecycle through custom resources. Choose Rook when you need multi-protocol storage (block, file, and S3-compatible object), you run on bare metal with local disks, and you want a fully open-source stack. Rook requires significant operational investment. Ceph is capable but demands expertise to tune and maintain at scale.

Longhorn is a CNCF Incubating project maintained by SUSE. Longhorn provides distributed block storage using a lightweight, microservices-based architecture. Each volume gets its own dedicated storage controller, which simplifies management and isolates failures. Longhorn includes built-in backup to S3-compatible storage, incremental snapshots, and a web-based management UI. Choose Longhorn when you want simple, easy-to-operate storage for general-purpose workloads, and you do not need the complexity of a full distributed storage system like Ceph.

For VMware professionals evaluating these options, think about it this way. vSAN is a single, integrated solution from one vendor. Kubernetes storage gives you a choice. That choice comes with a tradeoff: you are responsible for selecting, deploying, and maintaining the storage platform. Enterprise platforms like Portworx reduce that burden with automation, support contracts, and proven integrations across distributions.

How Data Services Translate

Storage is more than capacity allocation. Your vSphere environment provides snapshots, backup, replication, and disaster recovery as integrated services. These capabilities exist in Kubernetes, but they work differently.

Snapshots: vSAN snapshots capture VM state at a point in time. Kubernetes supports volume snapshots through the CSI Snapshot API. You create a VolumeSnapshot object that references a PVC. The CSI driver creates a point-in-time copy on the backend. You restore from a snapshot by creating a new PVC that references the VolumeSnapshot as its data source. The workflow is declarative. You define what you want. The system creates it.

Not all CSI drivers support snapshots. Check your storage platform's documentation. Portworx, Rook-Ceph, and Longhorn all support CSI snapshots. Cloud provider CSI drivers for EBS, Azure Disk, and GCE Persistent Disk also support these storage types.

Backup: In vSphere, you backup VMs using tools like Veeam, Commvault, or VMware Live Recovery. VMware Live Recovery combines VMware Live Site Recovery (formerly Site Recovery Manager) and VMware Live Cyber Recovery (formerly VMware Cloud Disaster Recovery). These tools capture VM state, including disks, configuration, and memory if needed.

Kubernetes backup requires a different approach. A Kubernetes application is not a single VM. It is a collection of pods, PVCs, ConfigMaps, Secrets, and custom resources spread across multiple nodes. Backing up a single volume is not enough. You need application-consistent backup that captures all components together.

Velero (an open-source project from VMware) is the most widely adopted Kubernetes backup tool. Velero backs up both Kubernetes resources (the YAML definitions) and persistent volume data. It stores backups in S3-compatible object storage and supports scheduled backups, retention policies, and cross-cluster restores. Portworx provides its own backup solution (PX-Backup) that integrates with its storage platform for application-aware backups across Kubernetes clusters.

Disaster Recovery: vSphere provides DR through vSAN stretched clusters, VMware Live Recovery, and array-based replication. VCF 9.0 introduces vSAN-to-vSAN replication for asynchronous DR across sites.

Kubernetes DR for workloads follows similar patterns. Portworx supports synchronous replication for zero RPO across metro distances and asynchronous replication for longer distances. Rook-Ceph supports Ceph's native mirroring for cross-cluster replication. Longhorn provides a built-in disaster recovery mechanism through backup and restore from S3 storage.

Data Services: VMware vs Kubernetes

Service	VMware / vSphere	Kubernetes
Snapshots	**vSAN Snapshots** vSAN Data Protection Point-in-time VM state capture. Managed through vCenter.	**CSI Snapshot API** VolumeSnapshot object Declarative snapshot via VolumeSnapshot. Restore by creating PVC from snapshot source.
Backup	**Veeam / Commvault** VMware Live Recovery VM-level backup: disks, config, memory. Single VM = single unit of protection.	**Velero / PX-Backup** Application-aware backup Captures pods, PVCs, ConfigMaps, Secrets together. Stores in S3-compatible storage.
Disaster Recovery	**vSAN Stretched Clusters** VMware Live Site Recovery / VLR vSAN-to-vSAN replication for async DR. Failover and failback at the VM level.	**Portworx Sync/Async DR** Rook-Ceph Mirroring / Longhorn S3 Zero RPO sync replication (metro). Async for longer distances. Full app-stack recovery.

The fundamental difference:
Kubernetes DR must account for the entire application stack, not a single VM. Replicating volumes is necessary but not sufficient. The Kubernetes resources, network configuration, and application state must be recoverable at the target site. Enterprise platforms like Portworx address this by providing application-aware DR that replicates both data and metadata.

What This Means for Your Operations

The storage paradigm shift from vSAN to Kubernetes is less about new technology and more about new workflows.

You stop provisioning individual datastores and start defining storage classes that describe available storage tiers. You stop manually creating VMDKs and let dynamic provisioning handle volume lifecycle. You stop relying on a single integrated stack and start evaluating storage platforms that fit your requirements.

The concepts transfer directly. Storage policies map to storage classes. Datastores map to persistent volumes. VAAI/VASA map to CSI. vSAN maps to your chosen software-defined storage backend. Snapshots, backup, and DR exist in both worlds, though the Kubernetes versions require application-aware thinking rather than VM-level operations.

Your vSAN expertise gives you a strong foundation. You understand replication factors, failure domains, storage tiering, and data protection. These concepts apply directly to Kubernetes storage platforms. The vocabulary changes. The engineering principles stay the same.

Chapter 6 examines networking. You will learn how NSX concepts translate to CNI plugins, how load balancing works with Kubernetes Services and Ingress controllers, and how network policies provide the micro-segmentation you rely on today. Read more at portworx.com/blog.

06

Chapter Six

Networking

From NSX to Kubernetes Networking

For VMware admins, networking isn't new—it's evolving. The same NSX concepts translate to Kubernetes through modular components that control traffic, connectivity, and segmentation.

You know software-defined networking. NSX creates virtual switches, distributes routing to hypervisors, and segments traffic with firewall rules. You configure overlay networks through NSX Manager. You deploy VMware Avi Load Balancer for application delivery. The network layer is the part of the VMware stack you rely on most and understand least when mapping to Kubernetes.

Kubernetes networking follows the same principles. Virtual networks connect workloads. Policies segment traffic. Load balancers distribute requests. The implementations differ, but the goals are identical: provide connectivity, enforce isolation, and deliver traffic to the right destination.

This part maps your NSX knowledge to Kubernetes networking concepts. You will see how CNI plugins replace virtual switches, how Services and the Gateway API replace load balancers, how Network Policies replace micro-segmentation, and how service meshes extend the model for complex environments.

Virtual Switches and Overlays: NSX vs CNI Plugins

In VMware Cloud Foundation, NSX provides the virtual networking layer. NSX creates overlay networks using the Geneve (Generic Network Virtualization Encapsulation) protocol. Each ESXi host runs a distributed router and a distributed firewall. VMs connect to logical segments through these constructs. Traffic between VMs on different hosts travels through Geneve tunnels. The underlying physical network sees only tunnel endpoints, not the VM traffic inside.

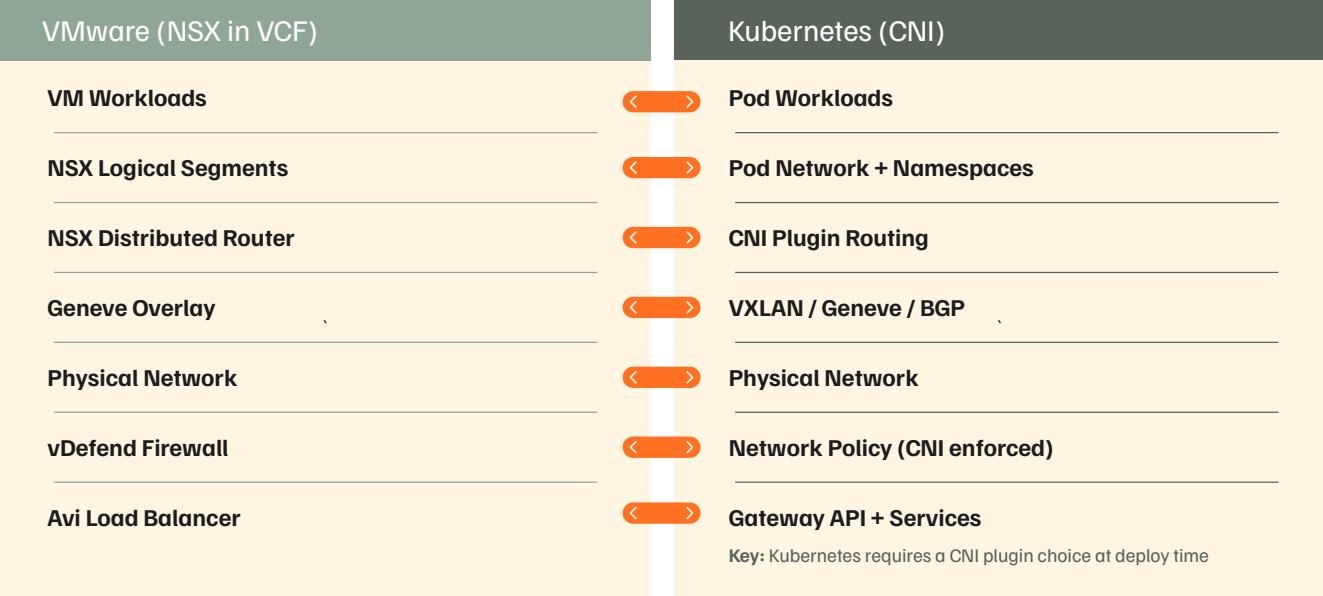
Starting with VCF 9.0, NSX is available exclusively as part of the VCF stack. Standalone NSX is no longer sold. Security capabilities previously bundled with NSX are now split into VMware vDefend, a separate add-on. Security and firewall licensing is now delivered via vDefend add-ons, while NSX handles overlay transport, distributed routing, and segment management.

Kubernetes uses Container Network Interface (CNI) plugins to provide pod networking. The CNI specification is a CNCF project defining how container runtimes configure network interfaces. When Kubernetes creates a pod, it calls the CNI plugin to set up networking. When the pod terminates, the CNI plugin cleans up.

This mirrors how NSX integrates with ESXi. The hypervisor calls NSX components when a VM powers on to connect it to the right logical segment. The CNI plugin does the same for pods.

The key difference: upstream Kubernetes does not ship with a built-in networking stack. You choose a CNI plugin during cluster deployment. Most enterprise distributions and managed services make this choice for you. OpenShift defaults to OVN-Kubernetes. AKS uses Azure CNI. EKS uses the Amazon VPC CNI. You only need to think about CNI selection when building clusters from scratch or when your requirements outgrow the default. This is similar to choosing between vSphere Standard Switch and NSX, except in Kubernetes, the choice is mandatory. Every cluster needs a CNI plugin.

Three CNI plugins dominate production environments.



Cilium uses eBPF (Extended Berkeley Packet Filter) to implement networking at the kernel level. Cilium bypasses traditional iptables-based packet processing, which reduces latency and CPU overhead. It is a CNCF Graduated project and the default CNI in Canonical Kubernetes LTS. It provides networking, network policy enforcement, load balancing (replacing kube-proxy entirely), and observability through its companion tool Hubble. Cilium supports both overlay (VXLAN, Geneve) and direct routing modes. For VMware professionals, think of Cilium as the closest equivalent to NSX in ambition: a comprehensive networking and security platform built into the infrastructure layer.

Calico provides networking through BGP routing or VXLAN encapsulation. Calico is well established and known for its network policy engine. Organizations like Reddit and CoreWeave run Calico in production. Calico supports both iptables and an eBPF data plane. In BGP mode, routes pod traffic without overlay encapsulation, similar to how NSX handles routing at the hypervisor level. Calico is the default CNI in many enterprise Kubernetes distributions.

Flannel is the simplest option. It provides basic pod-to-pod connectivity using VXLAN overlay networking. It does not include network policy enforcement. Flannel works well for lightweight clusters, development environments, and situations where simplicity is the priority.

The choice between CNI plugins depends on your requirements. Need advanced security, observability, and kube-proxy replacement? Choose Cilium. Need proven enterprise networking with strong policy enforcement and BGP routing? Choose Calico. Need basic connectivity for a simple cluster? Flannel gets you started.

Load Balancing: NSX and Avi vs Kubernetes Services and Gateway API

In VCF, load balancing is handled by VMware Avi Load Balancer (formerly NSX Advanced Load Balancer). The embedded NSX load balancer is deprecated and will be removed in future releases. Avi separates its control plane (the Avi Controller) from its data plane (Service Engines). The Controller manages policies and configuration. Service Engines handle traffic. This architecture scales horizontally and supports local load balancing, global server load balancing (GSLB), WAF (Web Application Firewall), and container ingress.

Kubernetes implements load balancing through Services and the Gateway API.

A Kubernetes Service is the fundamental mechanism for distributing traffic. When you create a Service, Kubernetes assigns it a stable virtual IP (a ClusterIP). Any pod matching the Service's label selector becomes a backend. Traffic to the ClusterIP gets distributed across those pods.

Three Service types handle different scenarios. ClusterIP exposes the Service within the cluster only. NodePort exposes it on a static port on every node. LoadBalancer provides an external load balancer from the cloud provider or from a platform like MetalLB for bare-metal environments.

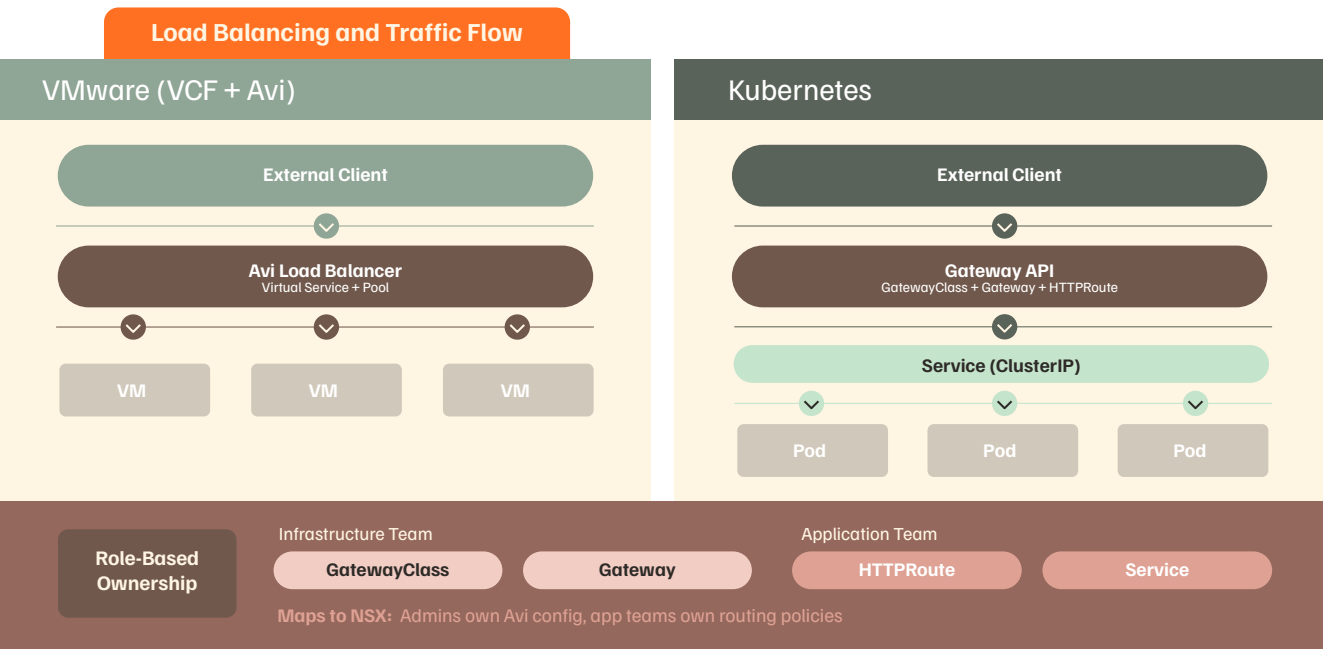
The kube-proxy component on each node implements Service VIP routing and load balancing. In clusters running Cilium, kube-proxy is often replaced entirely by Cilium's eBPF implementation, which provides the same functionality with lower latency.

For more sophisticated traffic management, Kubernetes introduced the Gateway API. The Gateway API is an add-on CRD set. It is the successor to the older Ingress resource. Where Ingress was limited to basic HTTP routing and required vendor-specific annotations for advanced features, the Gateway API provides a role-oriented, portable model.

The Gateway API uses three resource types. GatewayClass defines which controller implementation handles traffic (similar to choosing between different Avi deployment profiles). Gateway configures listeners, ports, and protocols. HTTPRoute, TCPRoute, GRPCRoute defines routing rules that direct traffic to backend Services.

This role separation maps to organizational boundaries. Infrastructure teams manage GatewayClass and Gateway resources. Application teams manage Route resources. In NSX terms, this is similar to how network administrators configure Avi virtual services and pools while application owners define their health monitors and routing policies through self-service workflows.

The Gateway API is vendor-neutral. The same HTTPRoute configuration works with NGINX Gateway Fabric, Istio, Cilium, Envoy Gateway, and VMware Avi (which now supports Gateway API for Kubernetes workloads). This portability eliminates the vendor-specific annotation sprawl plaguing the old Ingress model.



Network Policies: The Kubernetes Equivalent of Micro-Segmentation

Micro-segmentation is one of NSX's defining capabilities, now delivered through VMware vDefend in VCF. vDefend applies firewall rules at the virtual NIC level. Every VM gets its own firewall enforcement point. You define rules based on VM attributes, security groups, tags, and IP sets. Traffic between VMs in the same segment gets inspected and filtered. This level of granularity makes NSX valuable for zero-trust architectures.

Kubernetes Network Policies achieve the same goal. A NetworkPolicy is a namespace-scoped resource controlling traffic flow to and from pods. By default, all pods in a Kubernetes cluster accept traffic from any source. When you apply a NetworkPolicy to a set of pods, those pods become isolated according to the rules you define.

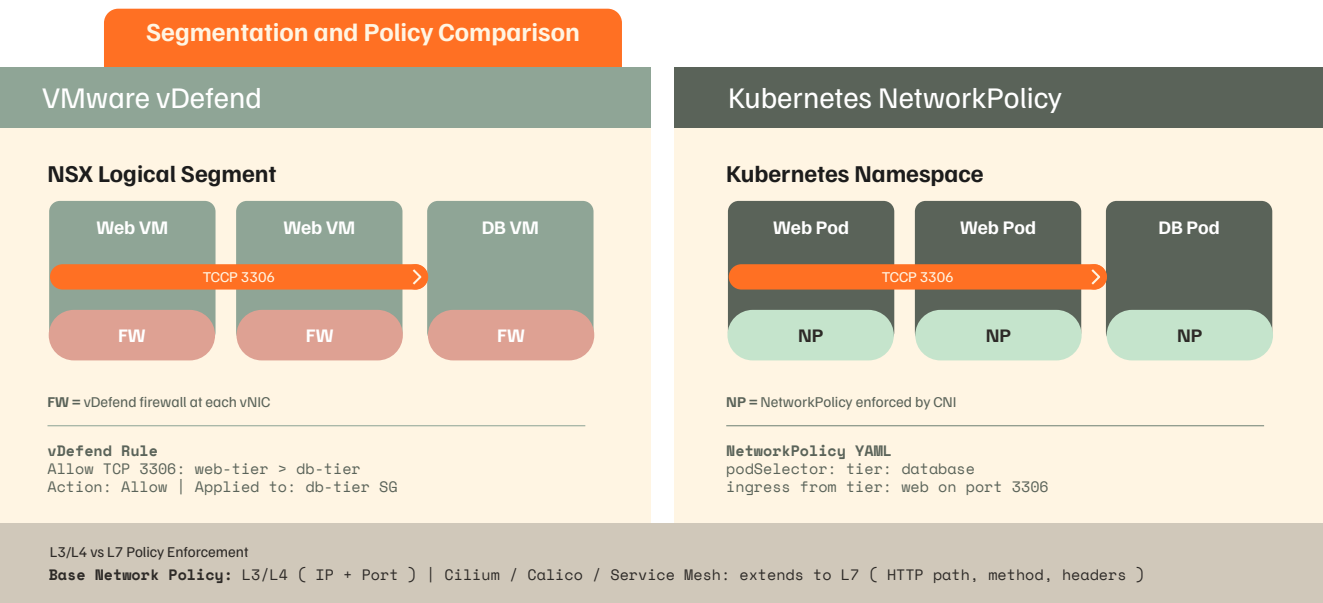
Network Policies use label selectors to identify pods, namespace selectors to scope rules across namespaces, and IP blocks for external traffic. You define ingress rules (what traffic a pod accepts) and egress rules (what traffic a pod sends).

Here is a concrete comparison. In vDefend, you might create a rule: "Allow TCP 3306 from web-tier security group to database-tier security group. Deny all other traffic to database-tier." In Kubernetes, the equivalent NetworkPolicy selects pods with label tier: database, allows ingress on port 3306 from pods with label tier: web, and denies everything else by default.

The enforcement mechanism depends on your CNI plugin. The Kubernetes API server stores the NetworkPolicy object. The CNI plugin reads it and enforces the rules on the data plane. Cilium enforces policies using eBPF programs in the kernel. Calico enforces them through iptables or its eBPF data plane. Flannel does not enforce Network Policies, which is why production clusters rarely run Flannel alone.

An important distinction: Kubernetes Network Policies operate at L3/L4 (IP addresses and ports). For L7 policies (HTTP methods, URL paths, headers), you need a CNI plugin with extended capabilities. Cilium provides L7-aware network policies natively. Calico offers similar L7 enforcement through its enterprise features. Service meshes like Istio also enforce L7 policies through their sidecar proxies. We cover service meshes in detail in the next section, including when and why you would add one to your cluster.

VMware professionals will notice Kubernetes Network Policies are declarative, defined in YAML, and version-controlled in Git. This aligns with the GitOps operational model described in Chapter 1. Your network security rules become code, reviewed in pull requests, and applied through CI/CD pipelines. If you prefer a visual approach, there are options. Rancher and OpenShift provide UIs for editing network policies. Calico includes a visual policy editor. These tools are not as integrated as vSphere's interface, but they bring familiar ClickOps workflows to Kubernetes networking for teams that want them.



Service Meshes: When You Need More

NSX provides a comprehensive networking and security platform through a single product family. Distributed routing, firewall, load balancing, VPN, and traffic analysis are all provided by a single vendor. You configure everything through NSX Manager.

Kubernetes takes a more modular approach. Basic networking comes from CNI plugins. Load balancing is provided by Services and the Gateway API. Network policies come from the CNI enforcement layer. When you need advanced traffic management for service-to-service communication, you add a service mesh.

A service mesh manages east-west traffic (service-to-service within the cluster), while the Gateway API manages north-south traffic (external clients to services). Service meshes provide mutual TLS (mTLS) encryption between services, traffic splitting for canary deployments, retry and timeout policies, circuit breaking, and observability through distributed tracing.

Istio is the most widely adopted service mesh. Istio uses Envoy proxy sidecars injected alongside application containers to intercept and manage traffic. Istio's newer "ambient mode" eliminates sidecars by using a node-level ztunnel proxy for L4 concerns and optional waypoint proxies for L7 policies. This reduces resource overhead significantly. Istio supports the Gateway API natively and intends to make it the default traffic management API.

Linkerd is a lightweight alternative focused on simplicity and performance. Linkerd uses its own Rust-based micro-proxy instead of Envoy. It provides mTLS, traffic splitting, retries, and observability with lower resource consumption than Istio. Linkerd also supports the Gateway API for both ingress and mesh traffic through the GAMMA (Gateway API for Mesh Management and Administration) initiative.

Cilium offers service mesh capabilities without sidecars by handling traffic management with eBPF and proxy components, as needed, in a sidecar-less model. This approach provides mTLS, traffic policies, and observability without the per-pod proxy overhead.

Not every cluster needs a service mesh. If your workloads communicate over well-defined APIs and you need mTLS, traffic management, and service-level observability, a mesh adds real value. If your applications are simpler, CNI-level networking with Network Policies provides sufficient control.

For VMware professionals, the mental model is this: the CNI plugin is your virtual switch and distributed router. Network Policies are your distributed firewall (vDefend). The Gateway API, with a controller such as Avi, Istio, or Cilium, serves as your load balancer. A service mesh adds features equivalent to advanced NSX capabilities, such as traffic analysis, encryption, and application-level routing, for service-to-service communication.

Bringing the Concepts Together

The networking transition from VMware to Kubernetes follows a clear mapping:

NSX logical segments map to the combination of pod IPAM and cluster networking managed by your CNI plugin, with isolation enforced through NetworkPolicy, labels, and namespaces. Distributed routing at the hypervisor level becomes pod routing at the node level through Cilium, Calico, or Flannel. VMware vDefend micro-segmentation becomes Kubernetes Network Policies enforced by the CNI. VMware Avi Load Balancer becomes Kubernetes Services for L4 and the Gateway API for L7 traffic management. NSX's integrated networking and security platform maps to a combination of CNI plugins, Network Policies, the Gateway API, and, optionally, a service mesh.

The modular approach in Kubernetes means more choices and more integration work. It also means more flexibility. You pick the CNI plugin that best matches your performance and security requirements. You pick the load balancer and the Gateway API controller fitting your operational model. You add a service mesh only when the complexity of your application architecture demands it.

Your NSX expertise gives you a strong foundation for understanding Kubernetes networking. The concepts are the same. Overlays, routing, segmentation, load balancing, and traffic management all exist in both worlds. The difference is how they are packaged, configured, and operated.

Chapter 7 examines security. You will learn how vCenter roles and permissions translate to Kubernetes RBAC, how VM isolation compares to pod security standards, and how secrets management and policy enforcement work in the cloud-native model. Stay tuned.

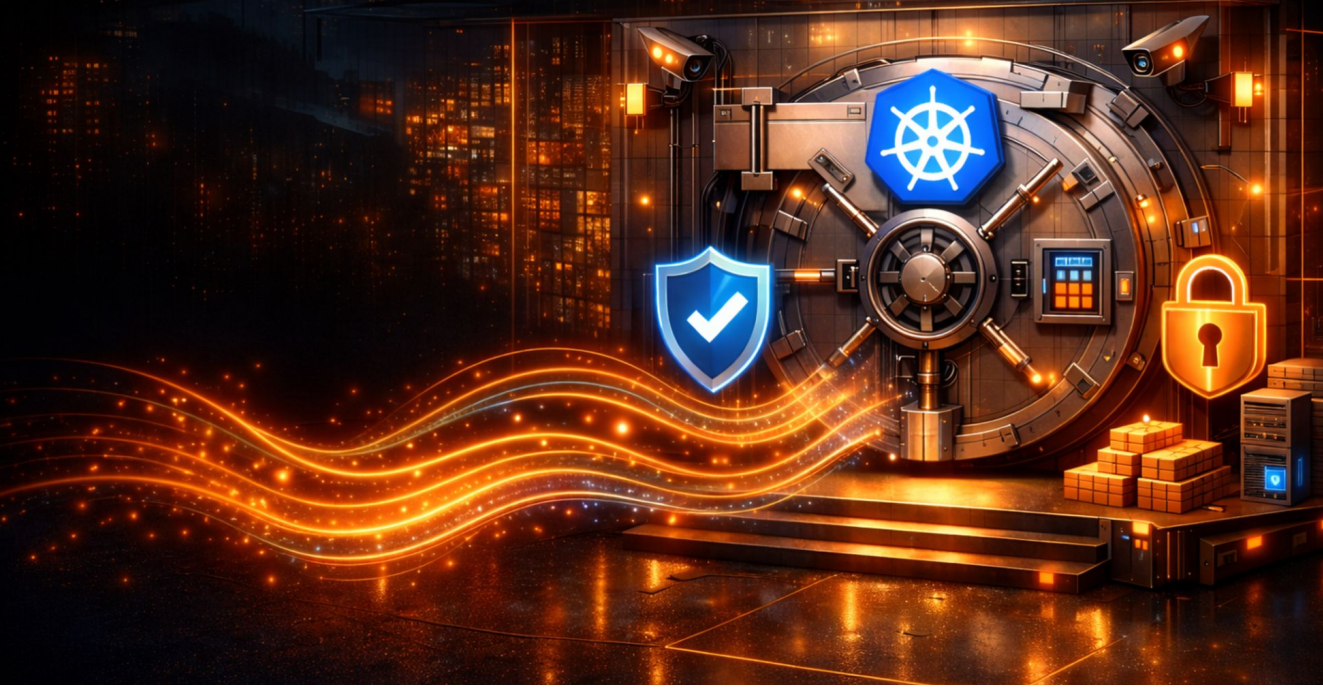
07

Chapter Seven

Security

vSphere Security Model vs Kubernetes Security

For VMware admins, security doesn't disappear—it expands. The same principles translate to Kubernetes across identity, policy, and workload layers with more granular control.



A change to Kubernetes doesn't change your security mission. While the tools evolve, the risks remain. In this chapter, we'll bridge proven vSphere security techniques into the cloud-native primitives required to maintain a consistent security posture across your entire stack.

You protect your VMware environment with layers. vCenter Single Sign-On authenticates users. Roles and permissions control what they do. The hypervisor isolates VMs from each other. VMware vDefend segments east-west traffic with distributed firewall rules. Every credential lives in a managed store, and compliance audits verify policies remain enforced.

Kubernetes security follows the same layered approach. Authentication verifies identity. RBAC controls access. Pod security standards enforce workload isolation. Network policies segment traffic. Secrets store credentials. The goals are identical. The mechanisms differ.

This chapter maps your vSphere security knowledge to Kubernetes security concepts.

Identity and Access Control

In vSphere, vCenter Single Sign-On (SSO) handles authentication. Users and groups come from Active Directory, LDAP, or the local SSO domain. Once authenticated, the permission model determines what each user can do. You assign roles to users or groups on specific objects in the vCenter inventory hierarchy. A role bundles a set of privileges. The Administrator role grants full control. The Read-Only role limits users to viewing. You create custom roles to grant specific combinations of privileges. Permissions propagate down the object hierarchy unless you override them at a lower level.

Kubernetes separates authentication from authorization. Authentication happens before any API request reaches the authorization layer. Kubernetes has no built-in user directory for human accounts. Instead, the API server trusts external identity providers for human users through X.509 client certificates, OIDC tokens from providers like Okta or Azure AD, and similar mechanisms. Kubernetes natively manages service accounts, which are first-class identities within the cluster. Pods use service accounts to authenticate with the API server and access cluster resources. The API server validates the token or certificate and extracts the identity for authorization.

Authorization uses Role-Based Access Control (RBAC). The model has four objects: Roles, ClusterRoles, RoleBindings, and ClusterRoleBindings. A Role defines permissions within a single namespace. A ClusterRole defines permissions across the entire cluster. A RoleBinding attaches a Role to a user, group, or service account within a namespace. A ClusterRoleBinding does the same at cluster scope.

The vSphere model maps to Kubernetes RBAC with a few key differences: vCenter roles apply to inventory objects such as datacenters, clusters, hosts, and VMs. Kubernetes Roles apply to API resources like pods, deployments, services, and secrets. vCenter permissions propagate through the object hierarchy by default. Kubernetes RBAC does not propagate. You must create explicit bindings at each scope.

In vSphere, a common pattern grants a team the Virtual Machine Power User role on a specific resource pool. In Kubernetes, the equivalent creates a Role with permissions to manage deployments, pods, and services within a namespace, then binds the Role to the team's group.

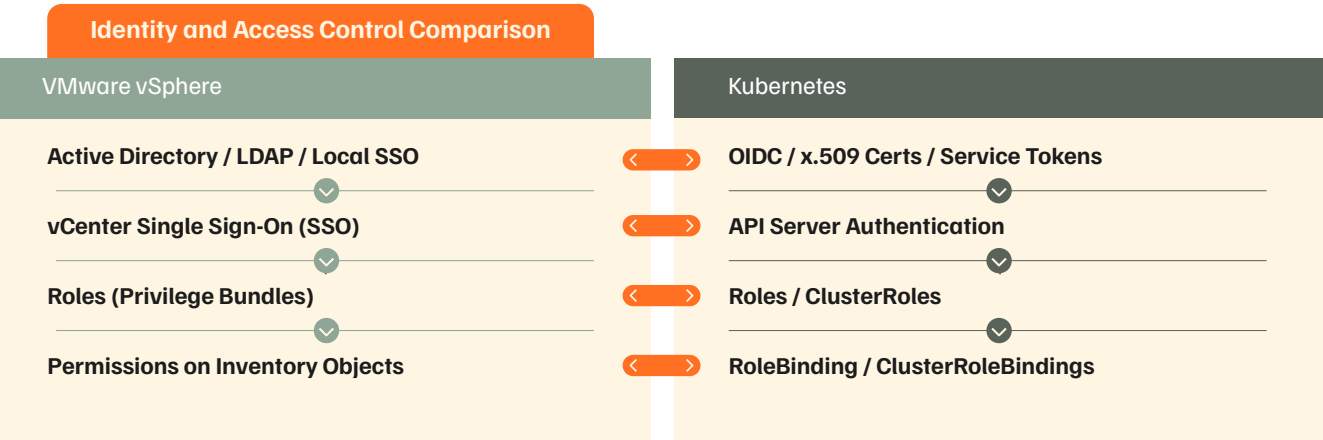


Figure 1: Kubernetes identity and access control versus VMware

Both systems share a core principle: least privilege. Grant only the permissions needed for the task. In vSphere, avoid using the Administrator role where a custom role suffices. In Kubernetes, avoid the cluster-admin ClusterRole unless the use case requires full cluster access. Audit RBAC bindings regularly. The kubectl auth can-i command verifies the actions a specific user or service account is allowed to perform.

Workload Isolation

In vSphere, the hypervisor provides strong isolation between VMs. Each VM runs its own operating system kernel. The hypervisor enforces memory isolation, CPU scheduling boundaries, and separate virtual hardware. One compromised VM has no direct access to another VM's memory or processes. VMware vDefend Distributed Firewall adds network-level micro-segmentation, controlling east-west traffic between VMs based on tags, groups, and Layer 7 rules. This combination of hypervisor isolation and network segmentation forms the security boundary for VMware workloads.

Kubernetes workload isolation operates differently. Containers within the same node share the host kernel. This shared-kernel model means that isolation depends on Linux security primitives: namespaces, cgroups, seccomp profiles, and AppArmor or SELinux policies. The isolation is strong when properly configured, but requires explicit enforcement.

Pod Security Standards (PSS) define three profiles for workload isolation. The Privileged profile imposes no restrictions. The Baseline profile blocks known privilege escalation paths. The Restricted profile enforces best practices, including running as non-root, restricting Linux capabilities, requiring seccomp profiles, and blocking privilege escalation. Many teams additionally enforce readOnlyRootFilesystem through Kyverno, Open Policy Agent Gatekeeper, or CEL policies, but this is not a built-in PSS Restricted control.

Pod Security Admission (PSA) enforces these standards. PSA is a built-in admission controller replacing the deprecated PodSecurityPolicy (removed in Kubernetes 1.25). You label namespaces with the desired enforcement level: enforce rejects non-compliant pods, warn generates warnings, and audit logs violations. A namespace labeled with the Restricted profile in enforce mode blocks any pod requesting root access or elevated privileges.

The VMware model enforces isolation at the hypervisor layer with no guest OS configuration needed. The Kubernetes model shifts the responsibility for isolation to the platform team. You define security contexts on pods, apply PSA labels to namespaces, and use admission controllers to prevent unsafe configurations from reaching the cluster.

For stronger isolation, Kubernetes supports sandboxed container runtimes. gVisor interposes a user-space kernel between the container and the host kernel. Kata Containers runs each pod in a lightweight VM, providing hypervisor-level isolation similar to what VMware administrators expect. Organizations handling sensitive workloads often deploy these runtimes alongside standard containerd for workloads requiring stronger boundaries.

Admission controllers extend isolation enforcement beyond PSA. OPA Gatekeeper and Kyverno are the two leading policy engines. Both intercept API requests and validate them against policies before the objects persist in etcd. Gatekeeper uses the Rego policy language from the Open Policy Agent project. Kyverno uses Kubernetes-native YAML policies, making the tool more accessible for teams already comfortable with Kubernetes manifests. Kubernetes now includes ValidatingAdmissionPolicy (stable in v1.30), a built-in alternative using Common Expression Language (CEL), reducing the need for external policy engines in some scenarios.

A typical enforcement pipeline applies multiple layers. PSA blocks obviously unsafe pods at the namespace level. An admission controller enforces organizational policies, such as resource limits, approved container registries, and mandatory labels. Network policies segment pod-to-pod traffic. Together, these layers approximate the defense-in-depth VMware vDefend provides in the vSphere environment. Unlike vDefend, native Kubernetes NetworkPolicy is limited to L3/L4 and depends on a supporting CNI for enforcement. L7 controls, IDS/IPS, and NDR require additional tooling such as Cilium with Hubble, a service mesh, or dedicated security platforms.

Secrets Management

In vSphere, credentials management is relatively contained. vCenter stores service account passwords, ESXi host credentials, and integration tokens. For vSphere platform components, identity and certificate trust are centralized through vCenter SSO and the VMware Certificate Authority (VMCA). Operational credentials and application secrets often reside in Active Directory, external vaults, or guest operating systems.

Kubernetes Secrets are the native mechanism for storing sensitive data like passwords, tokens, and certificates. A Secret is a Kubernetes API object stored in etcd. By default, Secrets are base64-encoded, not encrypted. This is an important distinction. Base64 encoding is not a security measure. Anyone with API access to read Secrets in a namespace sees the data in clear text after decoding.

To secure Secrets at rest, you enable encryption at the API server level. Kubernetes supports multiple encryption providers including AES-CBC, AES-GCM, and integration with external Key Management Services (KMS). With KMS integration, the encryption keys reside outside the cluster in a dedicated service like AWS KMS, Azure Key Vault, or Google Cloud KMS. The API server encrypts Secrets before writing them to etcd and decrypts them on read.

In many production environments, organizations integrate an external secrets manager with Kubernetes. HashiCorp Vault is the most widely deployed external secrets manager. The Vault Secrets Operator (VSO) syncs secrets from Vault into Kubernetes-native Secret objects. Applications can consume Kubernetes Secrets without requiring direct Vault access. The External Secrets Operator (ESO) provides the same pattern for AWS Secrets Manager, Azure Key Vault, Google Cloud Secret Manager, and other backends.

This architecture mirrors what vSphere administrators already do with credential management systems. The difference is scope. A VMware environment typically has hundreds of credentials managed through vCenter. A Kubernetes environment with microservices often manages thousands of secrets across multiple namespaces, clusters, and cloud providers. Automation through operators and dynamic secret generation becomes essential at this scale.

Both VMware and Kubernetes share a common vulnerability: overly broad access to credentials. In vSphere, granting a team full Administrator access to vCenter exposes every credential. In Kubernetes, granting read access to Secrets across all namespaces does the same. RBAC policies should restrict Secret access to the specific namespaces and service accounts requiring them.

Policy Enforcement and Compliance

VMware provides the vSphere Security Configuration Guide (formerly the Hardening Guide) for each release. This guide defines security settings administrators should configure and audit. VMware vDefend adds runtime policy enforcement through the Distributed Firewall, IDS/IPS, and Network Detection and Response (NDR). Compliance frameworks like PCI DSS, HIPAA, and SOC 2 have established audit procedures for VMware environments. VMware Cloud Foundation (VCF) Advanced Cyber Compliance, announced at VMware Explore 2025, addresses compliance automation specifically for regulated industries.

Kubernetes policy enforcement is distributed across multiple components. RBAC controls API access. PSA enforces pod security. Admission controllers validate resource configurations. Network policies segment traffic. Each component handles a specific enforcement domain.

The CNCF ecosystem provides tooling for compliance automation. Kyverno and OPA Gatekeeper enforce policies at admission time. Kyverno generates compliance reports showing which resources comply with defined policies and which violate them. Falco provides runtime security monitoring, detecting anomalous behavior in containers like unexpected process execution, file access, or network connections. Falco maps detections to the MITRE ATT&CK framework, similar to how VMware vDefend NDR correlates threats.

The Center for Internet Security (CIS) publishes Kubernetes Benchmarks defining security best practices. Tools like kube-bench automate CIS Benchmark checks against running clusters. The NSA/CISA Kubernetes Hardening Guide provides additional government-grade security recommendations.

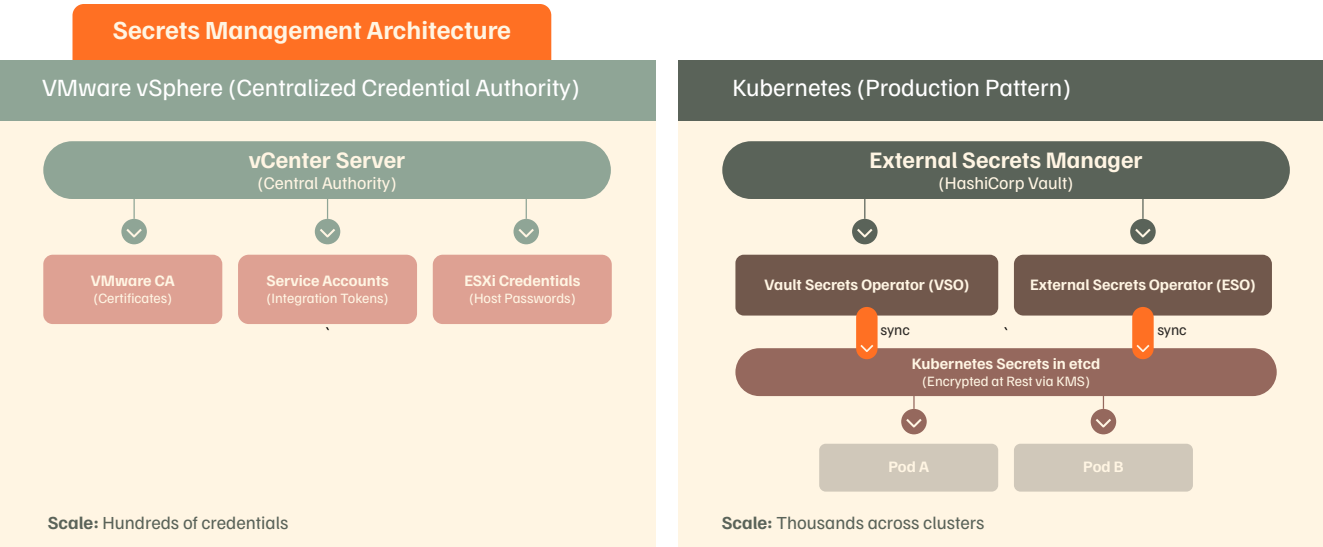


Figure 2: Comparing secrets management in VMware with Kubernetes

A significant difference exists in the maturity of audit and compliance tooling. VMware environments have decades of established compliance tooling, third-party integrations, and auditor familiarity. Kubernetes compliance tooling is maturing rapidly, but requires more platform team effort to implement and maintain. Organizations migrating from VMware should plan for this gap and invest in building compliance automation early in their Kubernetes adoption.

The Shared Responsibility Model

In vSphere, the responsibility model is straightforward. VMware provides the platform. Your infrastructure team manages hosts, storage, networking, and security configuration. Your team owns the full stack from hypervisor to guest OS.

Kubernetes introduces a shared responsibility model varying by deployment type. With self-managed Kubernetes (kubeadm, Kubespray), your team owns everything: control plane, worker nodes, networking, storage, and security configuration. With managed Kubernetes (EKS, AKS, GKE, and managed OpenShift offerings such as ROSA, ARO, and OpenShift Dedicated), the cloud provider or platform vendor manages the control plane. Your team manages worker node configuration, workload security, network policies, RBAC, and secrets management.

This split responsibility creates confusion for teams transitioning from VMware. In vSphere, you own the hypervisor and everything above the hardware layer. In managed Kubernetes, the provider owns the control plane, but you still own security configuration for namespaces, RBAC, pod security, network policies, and secrets. The provider does not enforce least-privilege RBAC for your workloads. The provider does not create network policies for your namespaces. The provider does not manage your secrets lifecycle.

VMware administrators moving to Kubernetes should map their existing security responsibilities to the Kubernetes model early in the transition. Define who owns RBAC policy creation and review. Define who enforces pod security standards. Define who manages the secrets lifecycle and rotation. Define who monitors for compliance violations and runtime threats. These responsibilities do not disappear. They are redistributed across platform and application teams in ways that require explicit documentation.

Your vSphere security knowledge transfers directly to Kubernetes

vCenter SSO maps to API server authentication with external identity providers and native service accounts. Roles and permissions map to RBAC with Roles, ClusterRoles, and their bindings. Hypervisor isolation and vDefend micro-segmentation map to Pod Security Standards, admission controllers, network policies, and sandboxed runtimes like Kata Containers. Credential management through vCenter maps to Kubernetes Secrets backed by external managers like HashiCorp Vault. The vSphere Security Configuration Guide and compliance automation map to CIS Benchmarks, Kyverno policy reports, and Falco runtime detection. The principles of least privilege, defense in depth, and compliance enforcement all apply. The implementations change. The discipline does not.

In the Chapter 8, *Day 2 Operations Lifecycle Management and Observability,* you will learn how vRealize Operations and centralized logging translate to Prometheus, Grafana, and Kubernetes-native observability. You will see how backup, disaster recovery, and cluster lifecycle management work in the cloud-native model.

08

Chapter Eight

Day 2 Operations

Lifecycle Management and Observability

For VMware admins, the shift isn't just technical—it's operational. Kubernetes introduces new team models, workflows, and responsibilities.

Deploying infrastructure is the easy part. Keeping it running, visible, and recoverable is where your real work begins. You know this from years of operating VMware environments. VMware Aria Operations (formerly vRealize Operations) tracks every metric across your vSphere estate. VMware Aria Operations for Logs (formerly vRealize Log Insight) centralizes log data from ESXi hosts, vCenter, and guest operating systems. VMware Live Recovery orchestrates disaster recovery with automated failover plans. vSphere Lifecycle Manager patches ESXi hosts with rolling remediation across clusters.

Following Broadcom's acquisition of VMware, standalone Aria product licenses were retired. VMware Aria Operations and Aria Operations for Logs are now bundled into VMware Cloud Foundation (VCF) and VMware vSphere Foundation (VVF) subscriptions.

Kubernetes requires the same operational rigor. The tools change. The discipline does not.

The following sections map your VMware Day 2 operations knowledge to the Kubernetes observability and lifecycle management model.

Monitoring - Aria Operations vs Prometheus and Grafana

VMware Aria Operations provides a single pane of glass for your entire vSphere environment. It collects metrics from ESXi hosts, VMs, datastores, and network components through built-in adapters. Dashboards show CPU, memory, storage, and network utilization. Predictive analytics forecast capacity exhaustion. Alerts trigger when thresholds breach defined limits. The platform integrates tightly with vCenter and operates as an end-to-end system where data collection, analysis, visualization, and alerting happen within one product.

Kubernetes splits these responsibilities across specialized tools. This modular model reflects Kubernetes' broader design philosophy. Instead of relying on a single tightly coupled platform, Kubernetes builds on open standards and interchangeable components. Organizations can adopt best-of-breed tools, replace individual pieces over time, or integrate with commercial observability platforms without re-architecting the cluster. The result is greater flexibility and significantly lower risk of vendor lock-in.

Prometheus serves as the metrics collection engine. It scrapes metrics from endpoints exposed by applications, Kubernetes components, and infrastructure exporters at regular intervals. Kubernetes control plane and node components expose Prometheus-format metrics, and production setups typically add exporters and services to round out visibility. The API server and kubelet expose /metrics endpoints. Common add-ons like kube-state-metrics (object state) and node-exporter (node OS metrics) expose additional `/metrics` endpoints for Prometheus to scrape. Prometheus stores time-series data on local disk by default, and many teams pair it with remote-write and long-term storage backends for retention and global query.

Grafana provides the visualization layer. It connects to Prometheus as a data source and renders dashboards for cluster health, node performance, pod resource consumption, and application-specific metrics. Pre-built dashboards from the Kubernetes Mixin project give you production-ready views comparable to the out-of-the-box dashboards in Aria Operations.

Alertmanager handles alert routing, grouping, and notification delivery. It receives alerts from Prometheus and routes them to Slack channels, PagerDuty, email, or other destinations based on configurable rules.

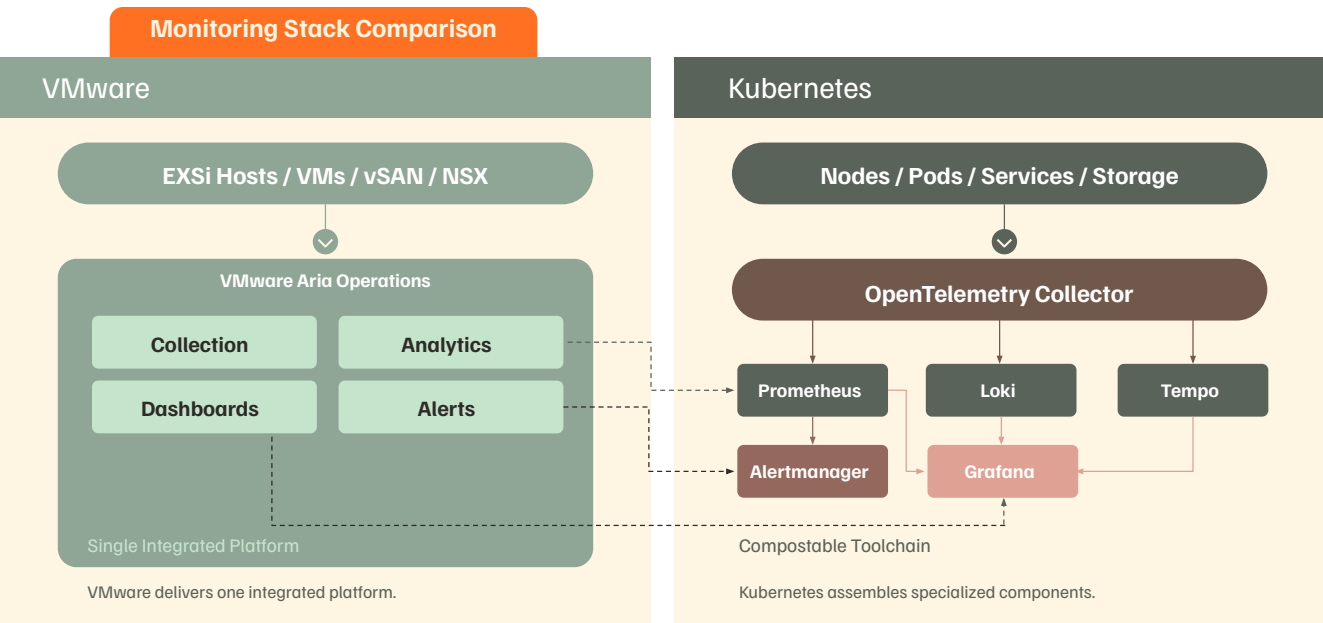
The key difference is clear. Aria Operations provides an integrated, opinionated monitoring platform. Kubernetes monitoring takes a modular approach. You choose each component independently, swap backends without re-instrumenting applications, and scale each layer (collection, storage, visualization, alerting) to match your specific environment.

Community-driven projects like Prometheus and Grafana evolve faster than proprietary alternatives, with thousands of pre-built exporters, dashboards, and integrations available at no additional licensing cost. You accept operational responsibility for the monitoring infrastructure while gaining full control over architecture decisions, avoiding vendor lock-in, and eliminating per-socket or per-core monitoring fees.

This keeps the VMware acknowledgment honest, then pivots to concrete Kubernetes advantages (modular scaling, faster innovation, zero licensing cost, backend portability) instead of treating the composable model as a tradeoff the reader must tolerate. On the broader note about positioning Kubernetes favorably, I will make sure future chapters frame the Kubernetes approach with equivalent emphasis on its strengths wherever the comparison allows it.

OpenTelemetry has emerged as the CNCF standard for telemetry data collection. The OpenTelemetry Collector is commonly deployed as a DaemonSet (agent mode) on nodes and/or as a Deployment (gateway mode), depending on scale, security, and routing needs. It scrapes Prometheus endpoints, tails container logs, and receives application traces through a single vendor-neutral pipeline. Many production Kubernetes environments run an OpenTelemetry Collector for metrics, logs, and traces, forwarding data to one or more backends.

For VMware administrators, think of Prometheus as the metrics engine (like the Aria Operations analytics cluster), Grafana as the dashboard layer (like the Aria Operations UI), and Alertmanager as the notification system (like Aria Operations alert plugins). OpenTelemetry is the universal data pipeline connecting all signal types.



Logging - Centralized vs Distributed

VMware Aria Operations for Logs provides centralized log management for your VMware environment. It collects logs from ESXi hosts, vCenter, NSX, and guest operating systems through agents and syslog forwarding. Built-in content packs parse VMware-specific log formats and provide structured search, dashboards, and automated event clustering. All logs flow to a central appliance or cluster for analysis.

Kubernetes generates logs differently. Every container writes to stdout and stderr. The container runtime captures these streams and stores them as files on the node. Kubernetes does not provide built-in log aggregation. You need a log collection and analysis stack.

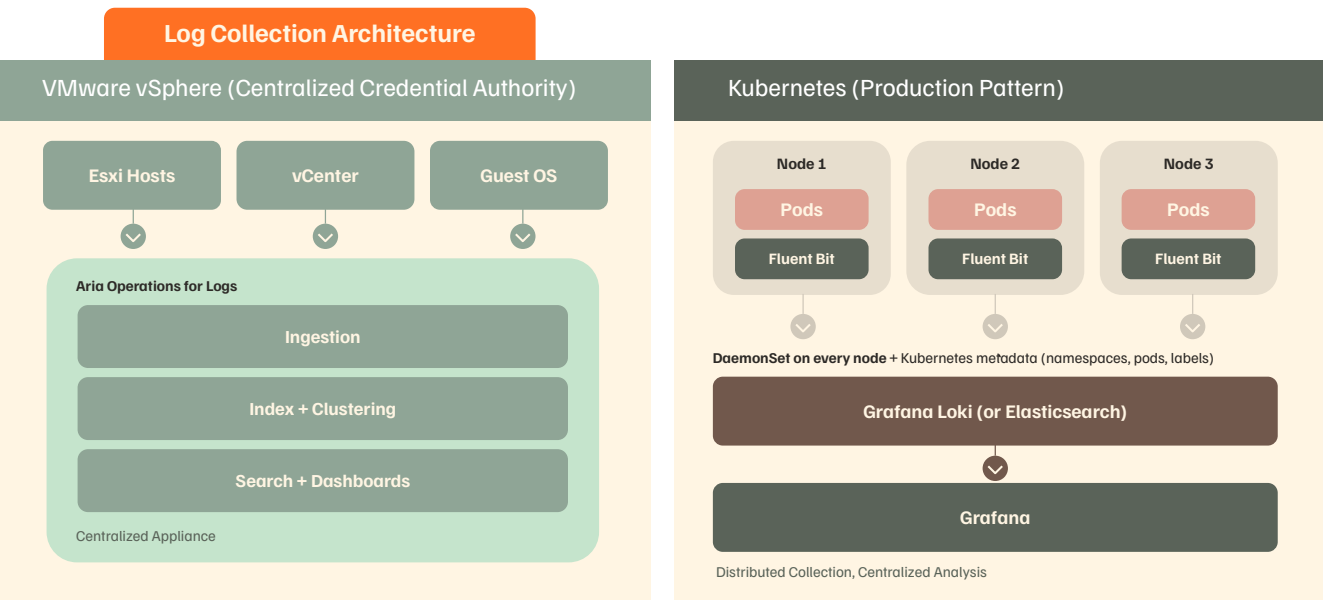
A common approach follows the centralized pattern you know. A node-level collector reads container log files, enriches them with Kubernetes metadata (pod name, namespace, labels), and forwards them to a central backend.

Fluent Bit is one of the most widely used lightweight log collectors for Kubernetes. It runs with minimal resource overhead and supports output to multiple backends. Fluentd serves a similar role with a broader plugin ecosystem for complex log routing and transformation.

Grafana Loki provides log aggregation designed for Kubernetes. It indexes logs by labels (namespace, pod, container) rather than full-text indexing, keeping storage costs low. Loki integrates with Grafana for querying logs alongside metrics in the same dashboard. LogQL, the query language for Loki, follows the same syntax patterns as PromQL for Prometheus.

Elasticsearch with Kibana (the ELK stack) remains a common alternative, especially for teams needing full-text search across log data. It requires more resources and operational effort than Loki but provides richer search and analytics features.

The concept maps directly. Aria Operations for Logs collects, indexes, and analyzes logs from a centralized appliance. In Kubernetes, Fluent Bit or the OpenTelemetry Collector collects logs from every node, and Loki or Elasticsearch stores and indexes them for search and analysis. The architectural pattern is the same. The implementation uses distributed, modular components instead of a single integrated appliance. This design reflects a broader cloud-native observability philosophy. In Kubernetes environments, workloads are ephemeral. Pods are created, rescheduled, and terminated continuously. Instead of relying on logs stored on individual machines or analyzed by a centralized appliance, logs are treated as streams of telemetry that are collected, enriched with metadata, and forwarded through a pipeline to one or more analysis backends. This decoupled model allows teams to scale observability infrastructure independently of the applications producing the data and to adopt new tooling without redesigning the platform.



Alerting and Troubleshooting Patterns

In VMware, troubleshooting follows a top-down path. Aria Operations detects an anomaly, shows the affected object in the inventory tree, and you drill into the related metrics. You check vCenter events and tasks. You open Aria Operations for Logs and correlate timestamps. The integrated platform provides context at every step.

Kubernetes troubleshooting works bottom-up. You start with kubectl to check pod status, describe resources, and read logs. You check events in the namespace for scheduling failures, image pull errors, or resource constraints. You use Prometheus metrics to identify patterns over time. You query Loki or Elasticsearch for log entries matching the timeframe.

The modern Kubernetes observability approach unifies these signals through correlation. Grafana dashboards link metrics, logs, and traces. Click a spike in CPU usage on a Grafana dashboard, and drill into the logs for the affected pods during the same window. Follow a distributed trace from Grafana Tempo or Jaeger to see request flow across services and identify where latency accumulates.

For alerting, Prometheus rules define conditions based on metrics expressions. When a rule evaluates to true for a specified duration, it fires an alert to Alertmanager. Common alerts include high pod restart counts, node resource pressure, persistent volume capacity warnings, and application error rate thresholds. Alertmanager groups related alerts, suppresses duplicates, and routes notifications to the right team.

This approach is architecture-driven rather than a limitation. The model gives you more programmable and version-controlled alerting than GUI-defined policies, because rules live as code in Git and ship through the same pipelines as the rest of your platform configuration. You write alerting rules as code, store them in version control, and deploy them through the same GitOps pipeline as your application configuration. Alert definitions become part of your infrastructure-as-code practice, auditable and reproducible across clusters.

Backup, Disaster Recovery, and Business Continuity

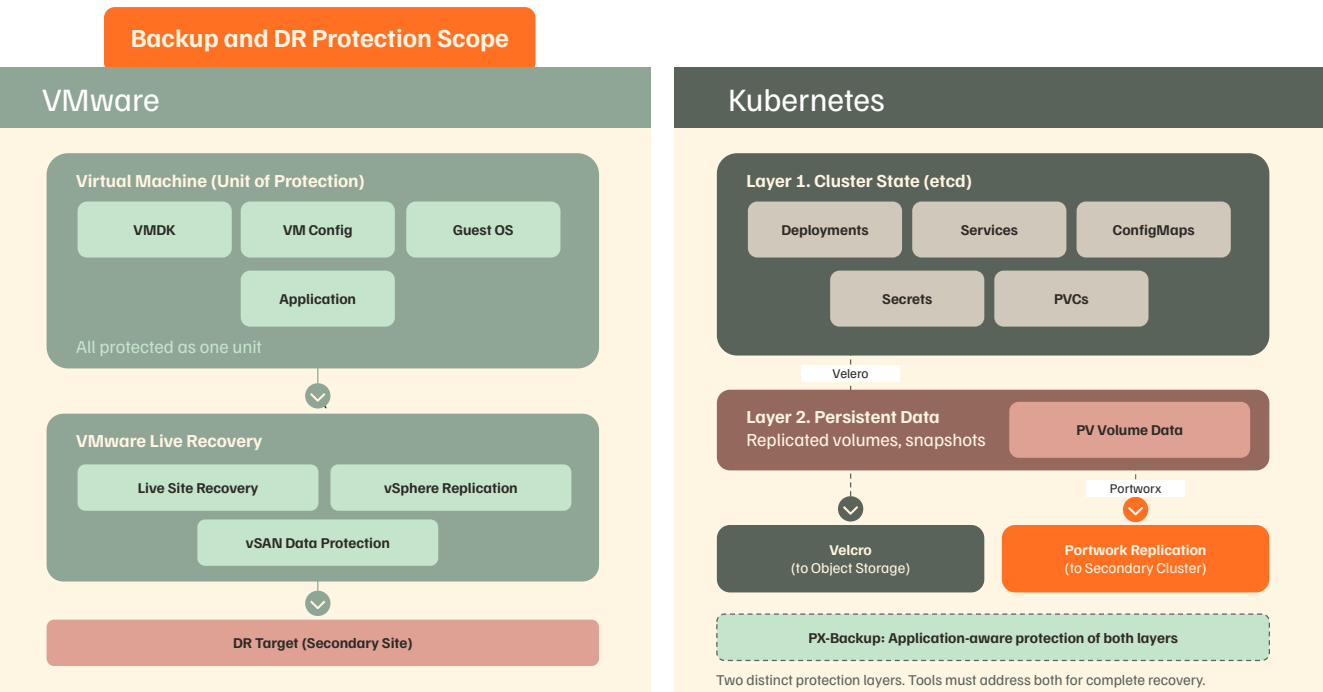
VMware Live Recovery provides integrated disaster recovery for VCF environments. It includes VMware Live Site Recovery (formerly Site Recovery Manager) for automated failover orchestration, vSphere Replication for hypervisor-based VM replication, and vSAN Data Protection for local and remote snapshots. Starting with VMware Live Recovery 9.0.3, the on-premises components ship in a single unified appliance. Recovery plans define VM startup order, network reconfiguration, and custom scripts for automated failover to a secondary site.

Kubernetes backup and disaster recovery requires a different approach because the unit of protection changes. In VMware, you protect VMs and their associated VMDK files. In Kubernetes, you protect two distinct layers, cluster state (the resource definitions stored in etcd) and persistent data (the volumes attached to stateful workloads).

Velero is the most widely adopted open-source tool for Kubernetes backup and restore. It backs up Kubernetes resource definitions and persistent volume data to object storage (S3, Azure Blob, Google Cloud Storage). Velero supports scheduled backups, namespace-level granularity, and pre/post-backup hooks for application-consistent snapshotsFor enterprise-grade data protection, Portworx Backup (PX-Backup) provides a purpose-built solution for Kubernetes. It understands application context, backs up all Kubernetes objects associated with a workload (deployments, services, secrets, ConfigMaps, PVCs), and restores complete applications to the same or different clusters. Commercial alternatives include Kasten K10 from Veeam and TrilioVault.

Disaster recovery in Kubernetes extends beyond backup and restore. Portworx provides synchronous and asynchronous replication of persistent volumes across clusters and regions. Combined with application-level failover logic, this delivers DR capabilities comparable to vSphere Replication and VMware Live Site Recovery.

The conceptual mapping is straightforward. VMware Live Recovery orchestrates VM-level failover with defined recovery plans. In Kubernetes, Velero or PX-Backup provides backup and restore, while storage-level replication (Portworx, or cloud-native equivalents) handles continuous data protection. The protection scope shifts from VMs to Kubernetes resources and persistent volumes.



VMware Protects VMs as a single unit.

Kubernetes separates state from data.

Cluster Upgrades and Maintenance Windows

vSphere Lifecycle Manager (vLCM) manages the patching and upgrade lifecycle for ESXi hosts. You define a desired image or baseline, scan hosts for compliance, and remediate non-compliant hosts. vLCM orchestrates rolling upgrades across clusters, evacuating VMs from each host through vMotion before applying updates. The process is well-defined, GUI-driven, and integrated into vCenter.

Kubernetes cluster upgrades follow a different pattern based on versioning policy. Kubernetes releases a minor version roughly every four months (about three per year). Upstream Kubernetes maintains patch releases for the most recent three minor versions, which works out to roughly one year of patch support for a given minor release. The upgrade path requires moving one minor version at a time. You upgrade the control plane components first (API server, controller manager, scheduler, etcd), then upgrade worker nodes.

For self-managed clusters, you coordinate the upgrade across multiple components. kubeadm provides a structured upgrade workflow for clusters bootstrapped with kubeadm. You run kubeadm upgrade plan to check available versions, kubeadm upgrade apply to upgrade the control plane, then drain and upgrade each worker node individually.

Managed Kubernetes services (EKS, AKS, GKE) simplify this process. The provider handles control plane upgrades. You manage node pool upgrades, which typically roll new nodes into the pool and drain old ones.

The maintenance window concept differs significantly. In VMware, vMotion enables zero-downtime host maintenance. You put a host in maintenance mode, VMs migrate to other hosts, and workloads continue running. Kubernetes uses a similar approach through pod disruption budgets and node draining. When you drain a node, Kubernetes evicts pods according to disruption budget rules, and the scheduler places them on other available nodes. Applications with proper replica counts and disruption budgets experience zero downtime during node maintenance.

Cluster upgrades also include addon and operator updates. The monitoring stack, CNI plugin, storage drivers, and ingress controllers all have independent release cycles. Tools like Helm and GitOps controllers (Argo CD, Flux) manage these component upgrades declaratively. You define the desired version in your Git repository, and the GitOps controller applies the update to the cluster.

For VMware administrators, think of Kubernetes cluster upgrades as a more modular version of vLCM remediation. Instead of a single host image, you manage versions for the control plane, node OS, container runtime, and every cluster addon independently. The tradeoff is more flexibility and control, with more components to track.

The Operational Mindset Shift

The fundamental difference in Day 2 operations between VMware and Kubernetes is the shift from integrated platforms to composable toolchains.

VMware delivers monitoring, logging, backup, and lifecycle management as tightly integrated products within VCF. Kubernetes gives you building blocks and the freedom to assemble them. Rather than relying on a preset operational and automation model, teams can integrate best-of-breed tools at their own pace and evolve their platform without being locked into a single vendor's ecosystem.

This shift mirrors the broader ClickOps-to-GitOps transition introduced in Chapter 1 of this series. Your monitoring configuration, alerting rules, backup schedules, and upgrade definitions live in Git repositories. Changes go through pull requests. Deployments happen automatically. The operational infrastructure becomes code, version-controlled and auditable.

Your VMware operations experience provides the foundational knowledge for Kubernetes Day 2 management. You understand why monitoring matters. You know the importance of centralized logging. You recognize the criticality of tested backup and recovery procedures. You appreciate the discipline of planned maintenance windows. Apply these principles to the Kubernetes toolchain, and you will build a production-grade operational practice.

Chapter 9 covers the migration journey. You will learn practical frameworks for assessing your VMware estate, categorizing workloads, building a platform team, and planning a phased transition to Kubernetes.

09

Chapter Nine

The Migration Journey

Planning Your Transition

For VMware administrators, migration isn't a leap—it's a phased transition that builds on what you already run.

You understand the Kubernetes stack. The previous eight parts mapped VMware concepts to their Kubernetes equivalents across compute, storage, networking, security, and Day 2 operations. You know how KubeVirt bridges the gap between VMs and containers. You have the technical vocabulary.

Now comes the harder question. How do you move a production VMware estate to Kubernetes without disrupting the business?

Migration is a project management challenge as much as a technical one. Success depends on accurate assessment, workload categorization, team readiness, and phased execution. Organizations attempting to move everything at once fail. Organizations refusing to start at all fall behind. The right approach sits between those extremes.

This chapter provides the framework for planning your transition.

Assess Your VMware Estate

Every migration begins with inventory. You need a complete picture of your current environment before making any decisions about the target state.

Start with vCenter. Export your VM inventory, including resource allocation, storage consumption, network dependencies, and performance baselines. VMware Aria Operations (formerly vRealize Operations) provides historical utilization data for CPU, memory, storage IOPS, and network throughput. These baselines become your success criteria after migration.

Document application dependencies. A single VM rarely operates in isolation. Web servers connect to application servers. Application servers connect to databases. Databases replicate to standby instances. Map these relationships explicitly. Tools like VMware Aria Operations for Networks (formerly vRealize Network Insight) reveal east-west traffic patterns between VMs. These dependency maps determine which workloads migrate together and which require special handling.

Catalog every VM by business function, owner, SLA requirement, and compliance classification. A development workstation and a production database server belong in different migration waves. The assessment should answer four questions for each workload. What does the application do? Who owns the application? What are the uptime requirements? What are the data protection requirements?

Organizations commonly find that a significant portion of their VMs serve no active purpose during this assessment. Decommissioning unused VMs before migration immediately reduces scope and cost.

Categorize Your Workloads

The industry-standard framework for migration categorization uses the "6 Rs" model. AWS developed this framework by building on an earlier Gartner migration model, adding strategies like Retain and Retire to the original set. For a VMware-to-Kubernetes transition, four of these Rs apply directly.

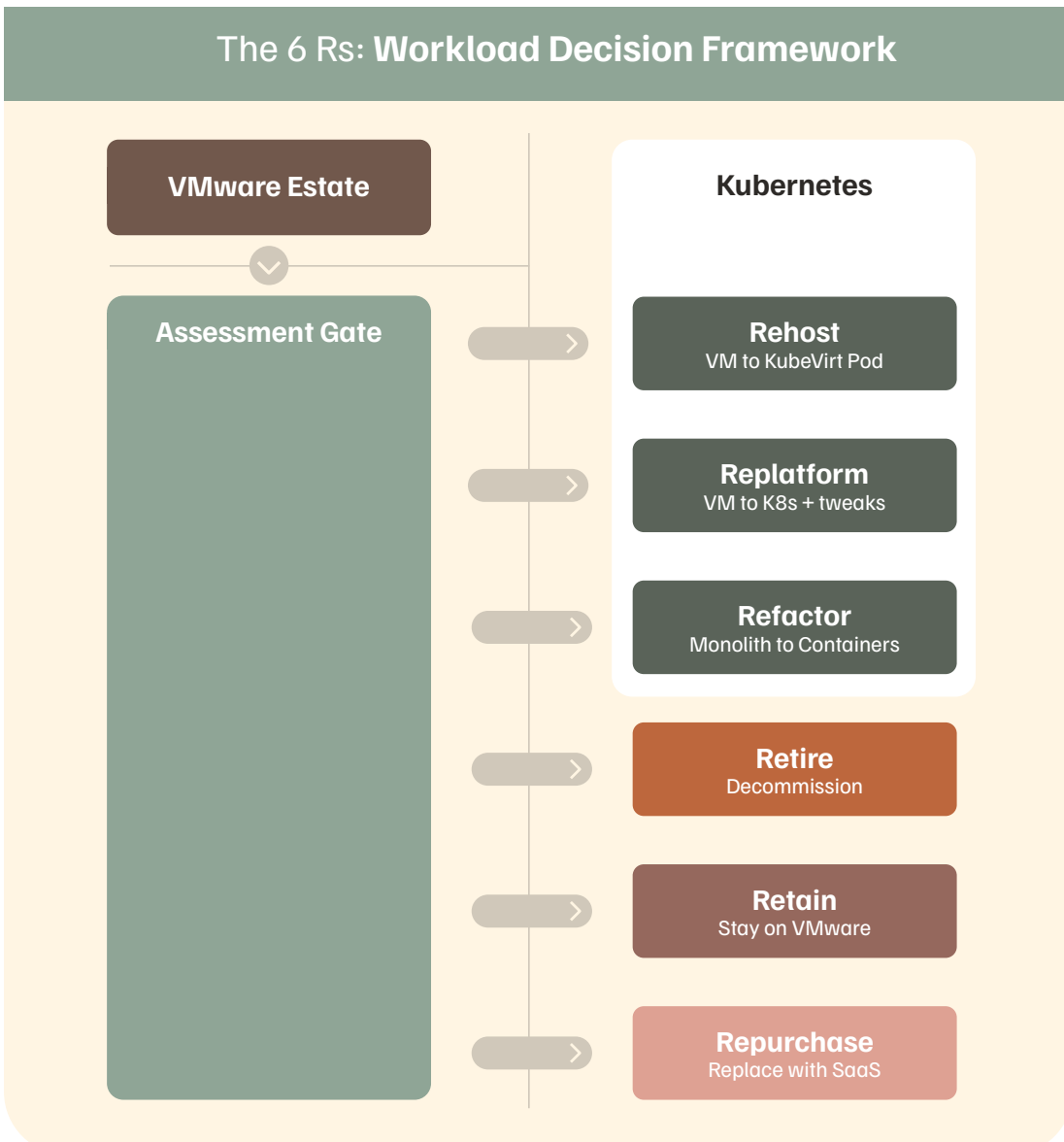


Figure 1: Workload categorization paths from VMware to Kubernetes using the 6 Rs framework.

Rehost (Lift and Shift). Move VMs to Kubernetes using KubeVirt with minimal changes. The VM runs inside a Kubernetes pod, managed by the KubeVirt operator. Operating systems, applications, and configurations remain unchanged. This approach works for legacy applications where source code is unavailable or where refactoring costs exceed the benefit. The Migration Toolkit for Virtualization (MTV) handles the actual VM migration from vSphere to KubeVirt, supporting both cold and warm migration workflows. For organizations running Pure Storage FlashArray, Portworx accelerates MTV migrations through Rapid VM Migration, which offloads data copies from the network to the array using native XCOPY operations. This bypasses host CPU and network overhead entirely, reducing migration times significantly for large VM disks. Once workloads are running on Kubernetes, Portworx delivers operational outcomes similar to Storage vMotion through its Enhanced Storage Migration feature and automated storage pool rebalancing via Autopilot policies. These features preserve familiar operational workflows on the new platform.

Replatform (Lift and Reshape) - Make targeted modifications during migration to take advantage of Kubernetes-native features without rewriting the application. Examples include replacing a VM-based load balancer with a Kubernetes Service, moving from local storage to persistent volumes backed by Portworx, or switching from cron-based scheduling to Kubernetes CronJobs. The core application architecture stays intact.

Refactor (Re-architect) - Redesign the application to run as containers. Break monolithic applications into microservices. Replace stateful session management with external state stores. Adopt 12-factor application principles. This approach delivers the greatest long-term benefit but requires the highest investment in development time and testing.

Retire - Decommission applications no longer needed. The assessment phase often reveals redundant services, abandoned projects, and legacy systems with no active users. Retiring these workloads before migration reduces complexity and licensing costs.

Two additional categories, **Retain** and **Repurchase**, complete the framework. Retain keeps certain workloads on the existing infrastructure when migration risk outweighs the benefit. Repurchase replaces on-premises applications with SaaS equivalents.

Most VMware estates follow a similar pattern. The largest share of workloads is rehost candidates. A smaller group benefits from replatforming. Only a fraction justifies full refactoring. The remainder splits between retire, retain, and repurchase. Your exact ratios depend on application age, business criticality, and available development resources.

Build Your Platform Team

Kubernetes requires a different operational model than VMware. In the VMware world, a small team of vSphere administrators manages the entire stack through vCenter. In Kubernetes, responsibilities are split across a platform team and application teams.

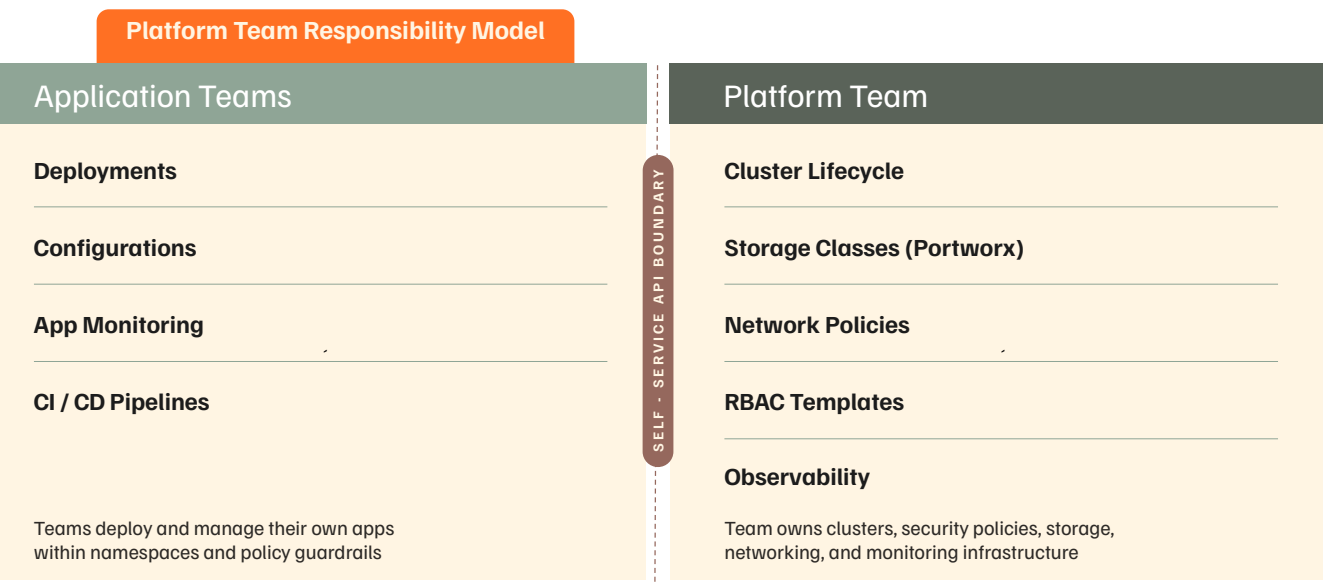


Figure 2: Ownership split between platform team and application teams in the Kubernetes operating model.

The platform team owns the Kubernetes clusters, including provisioning, upgrades, security policies, storage configuration, networking, and observability. This team builds the internal platform on top of raw Kubernetes, providing guardrails and self-service capabilities for application teams.

Start with your existing VMware administrators. Their infrastructure knowledge transfers directly. They understand resource management, high availability, disaster recovery, and storage operations. These skills apply to Kubernetes. The tools change, but the operational discipline remains the same.

A small initial platform team (often 3-5 engineers in early stages) with overlapping skills provides a starting point. At least one member should have deep Kubernetes experience. Others bring VMware operations, networking, storage, and security expertise. Cross-training fills gaps over time. Larger organizations scale this team as migration phases progress.

Define clear ownership boundaries from the start. The platform team manages cluster lifecycle, base images, storage classes, network policies, and RBAC templates. Application teams manage their deployments, configurations, and application-level monitoring within the namespaces and policies provided by the platform team.

Upskill Your Staff

The skills gap is the most underestimated risk in VMware-to-Kubernetes migrations. Technical professionals with years of VMware experience often resist change or doubt their ability to learn a new platform. Both reactions are addressable.

Start with fundamentals. Every team member should understand pods, deployments, services, namespaces, and persistent volumes. The Certified Kubernetes Administrator (CKA) exam provides a structured learning path and an industry-recognized credential. Pair formal training with hands-on lab environments where engineers practice real-world scenarios.

Create internal learning paths tailored to your environment. Storage administrators learn Portworx and CSI drivers. Network engineers learn CNI plugins and network policies. Security engineers learn RBAC, Pod Security Standards, and policy engines like Kyverno or OPA Gatekeeper. Each role maps existing expertise to Kubernetes-specific implementations.

Avoid the mistake of sending one person to a training course and expecting them to teach everyone else. Distributed learning across the team builds collective capability faster. Pair experienced Kubernetes practitioners with VMware veterans in daily work. The VMware engineer contributes operational rigor and production awareness. The Kubernetes engineer contributes platform-specific knowledge.

Upskilling timelines vary widely by team size, existing skill depth, and target complexity. Some organizations reach production readiness in a few months. Others need six months or longer, especially when teams are learning Kubernetes fundamentals alongside platform-specific tooling like Portworx, Kyverno, and GitOps workflows. Start structured training early and treat the pilot phase as a hands-on learning accelerator. This investment pays back through fewer misconfigurations, faster troubleshooting, and higher confidence during migration waves.

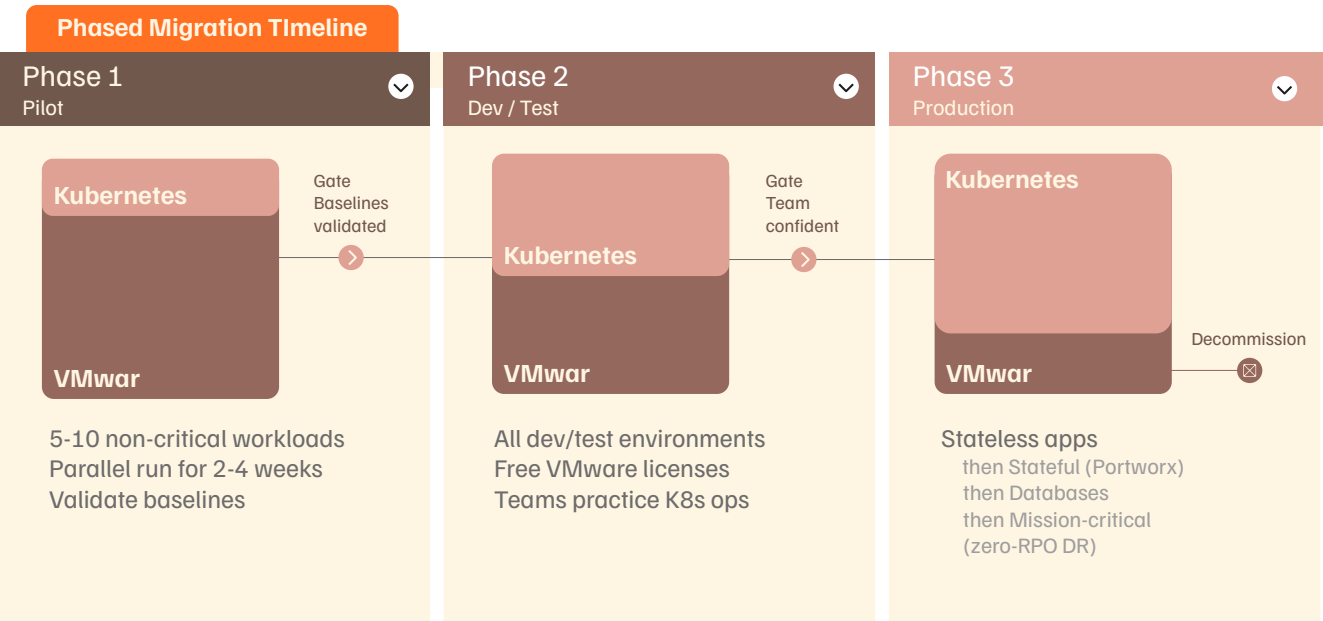


Figure 3: Three-phase migration timeline showing VMware footprint shrinking as Kubernetes adoption grows.

Execute in Phases

Phased migration reduces risk by limiting blast radius and building organizational confidence incrementally. Each wave teaches lessons applied to subsequent waves.

Phase 1 is the pilot. Select 5-10 non-critical workloads representing different application types. Include at least one KubeVirt VM workload to validate the lift-and-shift path alongside containerized workloads. Deploy Portworx for persistent storage. Configure monitoring, logging, and alerting. Run these workloads in parallel with their VMware counterparts for 2-4 weeks. Compare performance baselines. Document gaps and operational procedures.

Phase 2 targets development and testing environments. These workloads have lower SLA requirements and higher tolerance for disruption. Moving dev/test first frees VMware licenses and capacity while giving teams more practice with Kubernetes operations. Application teams begin deploying to Kubernetes as their default target.

Phase 3 addresses production workloads in waves of increasing criticality. Start with stateless web applications and API services. Progress to stateful workloads running on Portworx persistent volumes. Follow with database workloads using Portworx with features like synchronous replication and automated failover and follow careful validation of replication, failover, and consistency requirements balanced with any downtime tolerance.

Complete with mission-critical systems requiring zero-RPO disaster recovery.

Each phase should include explicit success criteria. Define acceptable latency thresholds, error rates, recovery time objectives, and team confidence levels before promoting workloads and moving to the next wave.

Measure Success

Define metrics before the first workload moves. Track both technical and organizational indicators.

Technical metrics include application response time before and after migration, storage IOPS and latency compared to VMware baselines, pod scheduling time versus VM boot time, recovery time during failure scenarios, and resource utilization efficiency.

To help identify performance issues, Portworx have developed an open-source benchmarking toolkit called "virtbench". Virtbench provides automated testing routines to measure and validate KubeVirt VM provisioning, boot times, network readiness, and failure recovery scenarios. It's designed for production environments running either KubeVirt or OpenShift Virtualization.

Organizational metrics include time-to-deploy for new applications, number of manual interventions required per week, mean time to resolution for incidents, team confidence scores through periodic surveys, and percentage of workloads migrated against the timeline.

Report progress to stakeholders regularly. Migration projects lose executive support when leadership sees no measurable outcomes. Weekly or biweekly dashboards showing workload counts, performance comparisons, and cost savings maintain visibility and funding.

Avoid Common Pitfalls

Organizations repeating the mistakes of earlier migrations lose time and credibility.

Do not skip the assessment. Moving workloads without understanding dependencies creates cascading failures when source systems go offline.

Do not attempt a single migration event. "Big bang" migrations fail at enterprise scale. Phase the work.

Do not underestimate storage complexity. Kubernetes persistent storage requires careful planning around storage classes, access modes, backup policies, and disaster recovery. Portworx addresses these requirements through a unified platform, but the configuration still demands attention and testing.

Do not ignore organizational change. Technical migration without team enablement creates a platform nobody knows how to operate. Invest in people alongside infrastructure.

Do not replicate VMware patterns on Kubernetes. Kubernetes operates differently by design. Teams that create one-namespace-per-VM or treat pods as permanent fixtures miss the benefits of the platform. Embrace the declarative model.

Do not delay decommissioning the old environment. Organizations running both platforms indefinitely pay twice the infrastructure cost and split their team's attention. Set firm dates for VMware decommission after each phase validates successfully.

Your VMware operational discipline transfers directly to migration planning.

You understand change management, maintenance windows, capacity planning, and phased rollouts. These skills apply to every stage of a Kubernetes transition. Assess the estate with the same rigor you apply to VMware upgrades. Categorize workloads using the 6 Rs framework. Build a platform team from your existing administrators. Upskill through structured learning paths and hands-on practice. Execute in phases, validate at every gate, and measure both technical and organizational outcomes. The tools change. The operational rigor stays the same.

Chapter 10 looks beyond the migration itself. You will learn how platform engineering principles shape the internal developer experience on Kubernetes. You will see how GitOps workflows, using tools like Argo CD and Flux, replace manual deployment processes. You will explore the broader Kubernetes ecosystem of Helm charts, operators, and custom resources, and understand how the platform extends to support containers, VMs, and AI workloads on a single control plane.

10

Chapter Ten

What's Next

Building Your Cloud-Native Future

For VMware administrators, what comes next isn't uncertainty—it's the opportunity to guide your organization forward.

You made the transition. You understand how Kubernetes maps to VMware across compute, storage, networking, security, and operations. You assessed your workloads and planned your migration. The question now is what comes after Day 2.

Kubernetes is a platform for building platforms. The clusters you deploy today serve as the foundation for an internal developer platform, GitOps workflows, and AI infrastructure. VMware gave you a complete stack with defined boundaries. Kubernetes gives you a programmable substrate with an ecosystem of tools you assemble to match your organization's needs.

This final chapter covers the capabilities and practices waiting beyond your initial migration.

The Cloud-Native Platform Stack

VMware VCF to Kubernetes Ecosystem

Chapter 10

Platform Engineering	GitOps Workflows	AI Workloads
Internal Developer Platforms	Argo CD / Flux	GPU / Kueue / KServe

KubeVirt (Ch 2): VMs + Containers in Unified Control Plane

VMware VCF		Kubernetes
vRealize Ops / Lifecycle Mgr	Ch 8	Prometheus / Grafana / Ops
vCenter RBAC / VM Isolation	Ch 7	RBAC / PSS / Admission
NSX / Avi Load Balancer	Ch 6	CNI / Gateway API / Mesh
vSAN / VMFS / Policies	Ch 5	PV / PVC / Portworx / CSI
ESXi / vSphere / DRS	Ch 4	Nodes / Scheduler / Kubelet

Platform Engineering and Internal Developer Platforms

VMware administrators served their organizations as a platform team, even without the title. You provisioned VMs on request. You configured networks and storage. You maintained templates and golden images. You were the platform.

Kubernetes formalizes this role through platform engineering. The DORA 2025 research program (Accelerate State of DevOps Report) found adoption of internal developer platforms has accelerated significantly across enterprises. Industry surveys consistently show a large and growing share of organizations now operate dedicated platform teams.

An internal developer platform (IDP) provides self-service capabilities to development teams. Developers request resources, deploy applications, and observe their workloads without submitting tickets or waiting for infrastructure teams. The platform team builds and maintains the IDP. Development teams consume services through defined interfaces.

Backstage, originally built by Spotify and now a CNCF incubated project, is one of the most widely adopted developer portal frameworks. Port and Cortex offer commercial alternatives. These portals provide a unified view of services, ownership, documentation, deployment status, and security posture.

Your VMware experience transfers directly to this model. You already think in terms of standardized templates, resource policies, and operational guardrails. In Kubernetes, those same concepts become golden paths. A golden path is a pre-approved, well-documented workflow for a common task. Deploying a new microservice, provisioning a database, or creating a CI/CD pipeline all follow golden paths defined by the platform team.

The shift from VMware to platform engineering changes the interface, not the mission. You still provide reliable infrastructure. You still enforce policies. The difference is that the delivery model moves from ticket-based provisioning to self-service consumption.

GitOps Workflows with Argo CD and Flux

Chapter 1 of this series introduced the shift from ClickOps to GitOps. Here is where GitOps becomes operational.

GitOps treats Git repositories as the single source of truth for your infrastructure and application state. Every change goes through a pull request. An automated agent running inside your cluster watches the repository and reconciles the live environment to match the declared state. Drift gets corrected automatically.

Two CNCF graduated projects dominate GitOps tooling. Argo CD has wider adoption. The CNCF 2025 Argo CD End User Survey found 97% of respondents running Argo CD in production, with nearly 60% of Kubernetes clusters using Argo CD for application delivery. Platform engineers account for 37% of Argo CD users, underscoring the tool's role in internal developer platforms.

Argo CD provides a web UI for visualizing application state across clusters. Teams transitioning from GUI-driven workflows will find the dashboard familiar. You see sync status, health checks, and deployment history in a single view.

Flux takes a different approach. Built entirely around Kubernetes controllers and Custom Resource Definitions, Flux operates without a built-in dashboard. Every configuration is a Kubernetes resource. Flux supports Git, Helm repositories, OCI images, and S3 buckets as sources. The modular design appeals to teams building custom platform tooling where GitOps serves as one component of a larger automation system.

For VMware professionals, Argo CD offers the gentler transition. The visual interface maps to the operational model you know. As your team matures, you adopt more declarative patterns and automated reconciliation, regardless of which tool you select.

Both tools enforce a principle that VMware environments often lack. In vSphere, administrators make changes through vCenter, and the running state becomes the source of truth. Configuration drift happens silently. GitOps reverses this relationship. Git holds the desired state. The cluster conforms. Every change is auditable, reversible, and reviewable before deployment.

The Kubernetes Ecosystem: Helm, Operators, and Custom Resources

VMware ships a complete, integrated stack. Kubernetes ships a core orchestration engine and relies on an ecosystem of projects to deliver full functionality.

Helm is the package manager for Kubernetes. A Helm chart bundles all the Kubernetes manifests needed to deploy an application into a single, versioned, configurable package. Helm charts exist for databases, monitoring stacks, ingress controllers, storage platforms, and hundreds of other components. When you deploy Portworx on a Kubernetes cluster, you use a Helm chart. When you install Prometheus and Grafana for monitoring, you use Helm charts.

Helm solves a similar problem to VMware templates and OVF files. You define a standard deployment pattern once and reuse the pattern across environments with different configurations. Helm goes further by managing versioned upgrades, rollbacks, and parameterized values across the full application lifecycle.

Operators extend Kubernetes with application-specific operational logic. An Operator is a custom controller paired with a Custom Resource Definition (CRD). The controller watches for custom resources and takes automated action based on the declared state. The Portworx Operator manages the entire lifecycle of your storage platform, including installation, upgrades, node replacement, and capacity expansion. Database operators such as the CloudNativePG Operator or the Percona Operator for MySQL automate backup schedules, failovers, scaling, and version upgrades.

Think of Operators as the Kubernetes equivalent of VMware vCenter plugins. They extend the platform with domain-specific management capabilities. The difference is that the Operator model is standardized. Any vendor or open source project delivers an Operator through the same Kubernetes API patterns.

Custom Resource Definitions (CRDs) extend the Kubernetes API itself. When you install Portworx, new resource types like StorageCluster and VolumeSnapshot become available alongside native Kubernetes objects. When you install Argo CD, ApplicationSet resources appear. CRDs allow Kubernetes to manage resources and workflows far beyond containers and pods.

This extensibility is Kubernetes' greatest strength and the reason the ecosystem continues to expand. The CNCF hosts over 200 projects spanning every layer of cloud-native infrastructure, from networking and storage to observability and security.

Extending Kubernetes for Your Organization

Every organization deploys Kubernetes differently. A financial services firm enforces strict compliance policies using admission controllers and policy engines such as Kyverno or OPA Gatekeeper. A media company optimizes for content delivery with custom ingress configurations. A healthcare provider encrypts persistent volumes and restricts network egress to comply with HIPAA.

Your Kubernetes platform should reflect your organization's requirements. Build golden paths for your most common workloads. Encode your security policies as code through admission webhooks and network policies. Define resource quotas and limit ranges per namespace to prevent runaway consumption.

Portworx fits into this organizational customization at the storage layer. Different teams get different storage classes with distinct performance profiles, replication factors, and encryption settings. Portworx storage policies enforce data protection requirements without relying on individual teams to configure storage correctly. This mirrors the storage policy approach in VMware vSAN, where administrators define policies, and VMs automatically inherit the correct protection level.

KubeVirt, introduced in chapter 2 of this series, extends Kubernetes to run VMs alongside containers. Organizations with legacy workloads use KubeVirt to bring VMs under Kubernetes management without requiring immediate containerization. Combined with Portworx for persistent storage and data mobility, KubeVirt provides a migration path where VMs and containers share infrastructure, networking, and operational tooling.

AI Workloads and the Next Frontier

Kubernetes is becoming the default platform for AI and machine learning workloads. The CNCF Annual Survey 2025 (released January 2026) found 82% of container users running Kubernetes in production. Among organizations hosting generative AI models, 66% use Kubernetes for some or all inference workloads.

This shift matters for your platform strategy. AI workloads demand GPU scheduling, high-throughput storage, and specialized networking. The Kubernetes ecosystem is responding. Kueue, a Kubernetes SIG project for batch workload management, provides quota enforcement, fair-share scheduling, and gang semantics where all workers in a distributed training job start together or not at all. KServe, a CNCF project for model serving, delivers a standardized inference layer with autoscaling, versioning, and traffic splitting.

GPU management in Kubernetes goes beyond allocating a GPU to a pod. NVIDIA Multi-Instance GPU (MIG) technology partitions a single H100 into multiple isolated instances, each with dedicated memory and compute resources. Topology-aware scheduling places workloads near-optimal PCIe or NVLink lanes to maximize throughput.

Storage-aware scheduling matters equally for data-intensive AI pipelines. Portworx's STORK (STorage Orchestrator Runtime for Kubernetes) extends the native Kubernetes scheduler to co-locate pods with their data replicas. STORK uses a scheduler extender to filter and prioritize nodes based on volume locality, ensuring pods run on hosts where their data already resides. This reduces latency for training data access and checkpoint writes. STORK also provides failure-domain awareness, automatically managing anti-affinity to distribute stateful workloads across availability zones.

For storage, AI workloads generate massive datasets and require high-bandwidth access to training data, model checkpoints, and inference caches. Portworx provides the persistent storage layer for these workloads, delivering consistent performance across training and inference pipelines while maintaining data protection and portability across clusters.

Your VMware resource management skills translate directly to AI workload management on Kubernetes. DRS balanced compute loads across hosts. Kubernetes schedulers and tools like Kueue balance GPU workloads across nodes. Storage policies in vSAN enforce protection levels. Storage classes in Kubernetes with Portworx enforce performance and replication for AI data pipelines.

Your Continued Learning Path

This series mapped nine VMware domains to their Kubernetes equivalents. You understand the parallels. Now build hands-on experience.

Start with a managed Kubernetes service. AWS EKS, Azure AKS, and Google GKE each offer free tiers or credits for learning. Deploy a simple application. Install Portworx to experience cloud-native storage. Set up Argo CD to practice GitOps workflows. Run a sample KubeVirt VM alongside a containerized workload.

The CNCF and Linux Foundation provide free introductory training courses. For formal credentials, the Certified Kubernetes Administrator (CKA) and Certified Kubernetes Application Developer (CKAD) exams each cost $445 through the Linux Foundation. Both exams are hands-on, practical tests, not multiple-choice questionnaires. The CKA validates the operational skills VMware professionals need most.

Join the community. KubeCon, the CNCF's flagship conference, draws thousands of practitioners and vendors. Regional Kubernetes meetups provide local connections. The Kubernetes Slack workspace hosts active channels for every topic covered in this series, from storage to networking to security.

The Broadcom acquisition forced a conversation about infrastructure strategy. Kubernetes answers the technical questions. Platform engineering answers the organizational questions. GitOps answers the operational questions. The tools and practices exist. The ecosystem is mature. The community is active.

You spent years mastering VMware. The skills you built, resource management, storage architecture, network design, security policy, and operational discipline, all apply to Kubernetes. The interface changes. The principles remain.

Your cloud-native future starts with the next cluster you deploy.

portworx by Everpure

Continue your journey

portworx.com

Read Now

Demystify Kubernetes for the VMware Admin

portworx.com | 800.379.PURE