TECHNICAL WHITE PAPER:
February 2026

Kong on vSphere Kubernetes Service

Reference Architecture

Table of Contents

Executive Summary

As Kubernetes environments scale, the proliferation of microservices and external integrations increases both traffic volume and connectivity complexity, introducing material risk to availability and performance. The primary challenge shifts from raw throughput to governance: as APIs and AI-enabled workloads expand, unmanaged service interactions make it difficult to apply consistent security controls, maintain predictable latency, and enforce consumption policies. Addressing this requires a unified, policy-driven control plane for connectivity, where routing, security, and observability can be applied consistently across the platform. Deploying as this control layer ensures the operational resilience of vSphere Kubernetes Service (VKS) is complemented by consistent enforcement and measurable, auditable management of traffic flows.

This whitepaper explores the integration of the **Kong API Gateway** as the standardized ingress and traffic management layer for VKS. By deploying Kong within the VKS ecosystem, organizations bridge the gap between traditional infrastructure and cloud-native agility, ensuring that every request, from the physical host to the deepest workload pod, is secure, optimized, and controlled.

Key Advantages of Kong on VKS:

- **Cloud-Native Ingress:** Seamlessly manages VKS service discovery and routing, providing a high-performance alternative to standard ingress controllers.
- **Centralized Security:** Enforces Zero-Trust principles through global authentication, rate limiting, and mTLS, protecting workloads regardless of their lifecycle.
- **Operational Consistency:** Empowers both VI admins and DevOps teams to manage API policies as code, ensuring consistent configurations across VCF environments.
- **Architectural Scalability:** Leverages a lightweight, sub-millisecond latency core that scales dynamically alongside VKS clusters to meet fluctuating demand.

Introduction

vSphere Kubernetes Service

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

Key benefits include:

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

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

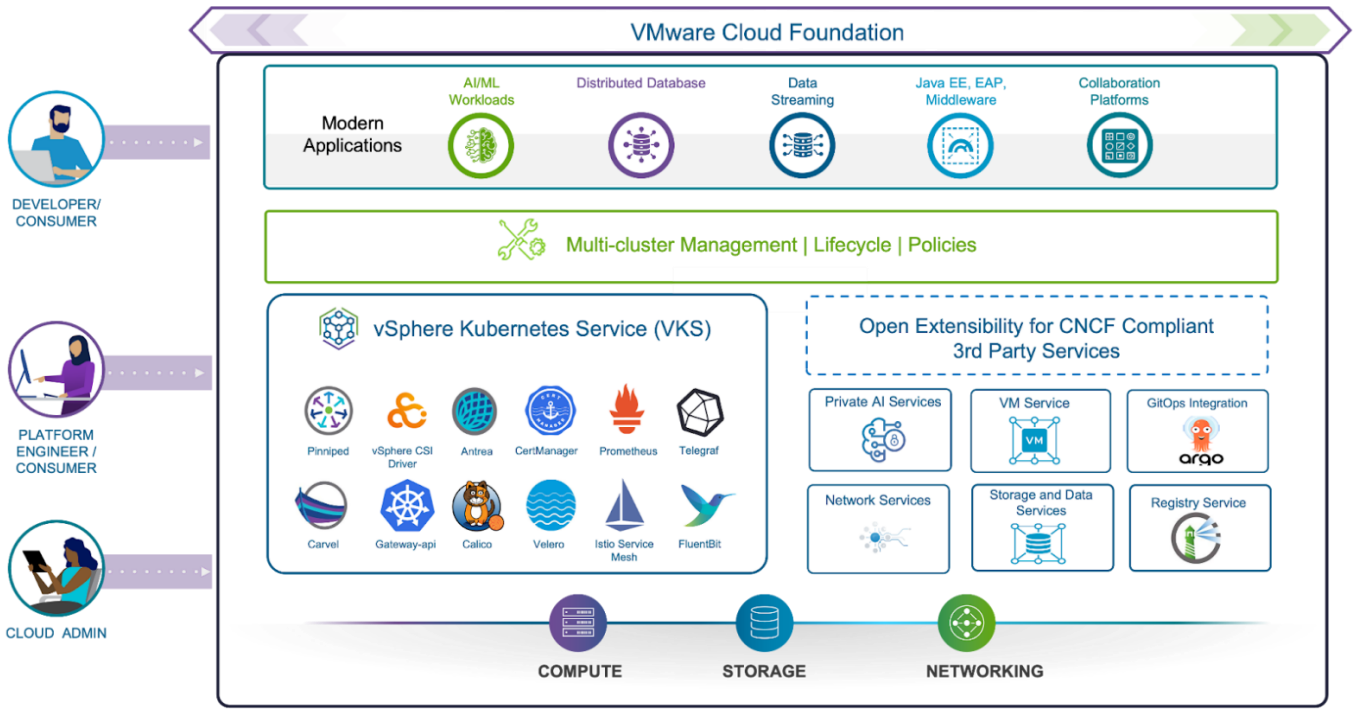

Figure 1: VMware Cloud Foundation with vSphere Kubernetes Service

Kong

Kong is a lightweight open-source API Gateway and microservice management platform. Within a VKS environment, Kong acts as the intelligent "front door," translating external requests into actionable internal traffic with sub-millisecond latency.

Unlike legacy gateways that struggle with the ephemeral nature of Kubernetes pods, Kong was designed and built in an age of modern architectures. It offers:

- **High-Performance Routing:** Leveraging a lightweight NGINX core to process thousands of requests per second without becoming a bottleneck.
- **Dynamic Scalability:** Seamlessly integrating with VKS ingress controllers to auto-discover services as they scale up or down.
- **Extensible Middleware:** A vast library of plugins for authentication (OIDC, JWT), rate limiting, and real-time observability.
- **Declarative Configuration:** Aligning with GitOps workflows, allowing teams to manage API policies as code alongside their application deployments.

By layering Kong over VKS, enterprises move beyond simple connectivity. They gain a unified control plane that ensures every API call is authenticated, every spike in traffic is managed, and every service interaction is visible.

Solution Architecture

Kubernetes Environment

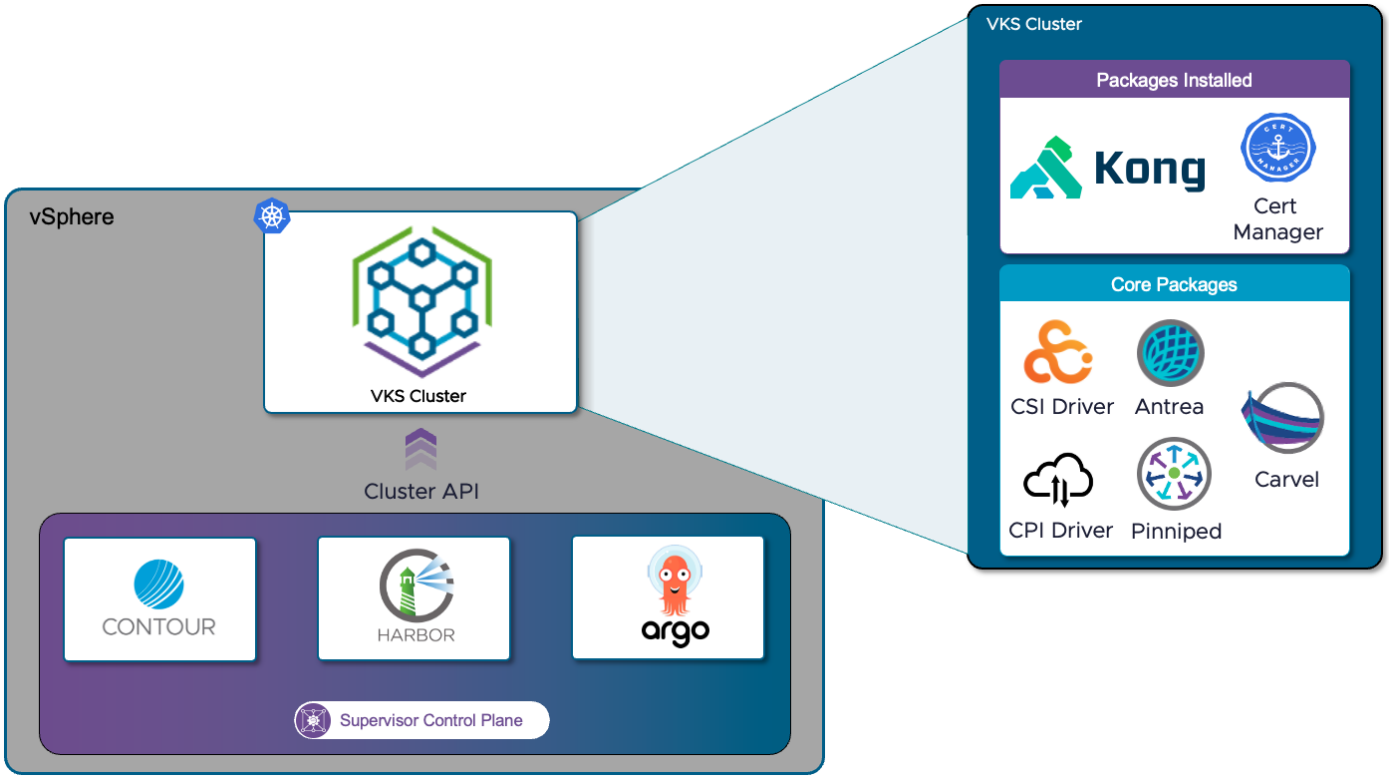

Figure 4: Supervisor Control Plane and vSphere Kubernetes Service

At its core, the Supervisor control plane provides essential services, such the Harbor container registry, the Contour ingress controller, and the Argo CD GitOps utility. For more information on vSphere Supervisor services and installation, visit: https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest.html

This control plane is responsible for the automated deployment and lifecycle management of CNCF-compliant Kubernetes clusters via the vSphere Kubernetes Service (VKS) via Cluster API. The specific version of VKS used in this whitepaper is 3.5.

VKS Cluster Configuration

The cluster comprised of three control plane nodes and three worker nodes. Storage was provided by vSAN Express Storage Architecture (ESA), using the default RAID 5 storage policy. Transparently, the vSphere CSI driver exposed this policy to an adjacent Kubernetes storage class, 'vsan-esa-default-policy-raid5'.

Worker node resources were configured using a *best-effort-2xlarge* profile, whilst control plane nodes utilized the guaranteed large profile. In addition, each node was associated with an extra 100 Gi persistent volume.

The cluster was then managed using a client VM. Refer to Appendix A for example client VM configuration and tooling

Component	Configuration	Notes
Kubernetes Control Plane	3 Replicas — 4x vCPU	VM Class: "guaranteed large" Storage Class: "vsan-esa-default-policy-raid5"

	16GB RAM vSAN RAID 5	
Kubernetes Worker Nodes	3 Replicas — 4x vCPU 16GB RAM vSAN RAID 5 + Extra 100Gi Volume	VM Class: “best-effort-2xlarge” Storage Class: “vsan-esa-default-policy-raid5”
Kubernetes Release	v1.34.2	
OS Image	Ubuntu 24.04	Kernel version 6.8.0-90-generic

Table 3: Kubernetes Cluster Configuration Summary

The VKS cluster configuration is defined declaratively in the YAML file below. As previously outlined, applying this manifest to the Supervisor triggers the creation of the cluster, which is then managed and reconciled by Cluster API.

```yaml
# vks-cluster.yaml
apiVersion: cluster.x-k8s.io/v1beta2
kind: Cluster
metadata:
  name: cluster-vks-kong
  namespace: supervisor-namespace
spec:
  clusterNetwork:
    pods:
      cidrBlocks: ["10.96.0.0/12"]
    services:
      cidrBlocks: ["192.168.0.0/16"]
    serviceDomain: cluster.local
  topology:
    class: builtin-generic-v3.5.0
    version: v1.34.1+vmware.1-vkr.4
    variables:
      - name: kubernetes
        value:
          certificateRotation:
            enabled: true
            renewalDaysBeforeExpiry: 90
      - name: vmClass
        value: guaranteed-large
      - name: storageClass
        value: vsan-esa-default-policy-raid5
    controlPlane:
      replicas: 3
      metadata:
        annotations:
          run.tanzu.vmware.com/resolve-os-image: os-name=ubuntu,os-version=24.04
    workers:
      machineDeployments:
```

```
    - class: node-pool
      name: cluster-vks-kong-nodepool
      replicas: 3
      metadata:
        annotations:
          run.tanzu.vmware.com/resolve-os-image: os-name=photon
      variables:
        overrides:
          - name: volumes
            value:
              - name: vol-kong-worker
                mountPath: /var/lib/containerd
                storageClass: vsan-esa-default-policy-raid5
                capacity: 100Gi
```

Kubernetes Packages

VKS clusters (via the released operating system — or *VKR* image) include a default set of core packages, such as Antrea for networking and Pinniped for authentication. For this deployment, the add-on package for cert-manager was also installed, while Kong was installed using Helm.

For details on how to install the cert-manager add-on, visit: https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/managing-vsphere-kuberenetes-service-clusters-and-workloads/managing-add-ons-in-vks-clusters/install-an-add-on-using-the-vcf-cli.html

Package	Version	Notes
Antrea	2.4.4	VKS core package
Gateway API	1.3.0	VKS core package
Guest-cluster auth-service	1.4.4	VKS core package
Metrics Server	0.8.0	VKS core package
Pinniped	0.41.0	VKS core package
Secretgen Controller	0.19.2	VKS core package
vSphere CPI	1.34.0	VKS core package
vSphere CSI	3.6.0	VKS core package
Cert-Manager	1.18.2	VKS add-on
Kong Operator	1.0.2	Helm package

Table 4: Kubernetes Cluster Packages

Solution Validation

Kong On-Prem

In an on-premises deployment, both the Control Plane (CP) and the Data Plane (DP) reside locally on the VKS cluster. This model is often chosen for high-compliance environments where the management layer cannot reside in a public SaaS environment.

Prerequisites:

- **VKS Cluster:** Version 3.5 or later.
- **Cert-manager add-on:** Required for automated mTLS certificate rotation between planes.
- **VCF CLI & Kubectl:** For cluster interaction and Supervisor management.
- **Kong Gateway Enterprise License** (optional).

Installation

Step 1: Ensure cert-manager is Installed and Functional

Switch to the supervisor context, for example:

```
# Switch to the Supervisor context, e.g.:
vcf context use supervisor-cluster:supervisor-namespace
```

See the details

```
# Show details on the cert-manager package
kubectl -n vmware-system-vks-public describe addons cert-manager

Name:         cert-manager
Namespace:    vmware-system-vks-public
Labels:       addon.kubernetes.vmware.com/addon-name=cert-manager
Annotations:  <none>
API Version:  addons.kubernetes.vmware.com/v1alpha1
Kind:         Addon
Metadata:
  Creation Timestamp:  2026-01-16T15:28:46Z
  Generation:          1
  Owner References:
    API Version:     data.packaging.carvel.dev/v1alpha1
    Kind:            PackageMetadata
    Name:            cert-manager.kubernetes.vmware.com
    UID:             4874a2b9-5841-4264-aae7-d06405bfdb9f
  Resource Version:  156246622
  UID:               142943b2-946d-40b3-a1a4-daacc785eb68
Spec:
  Description:  Certificate management
Events:         <none>
```

Step 2: Add the Kong chart to Helm

```
# Switch back to the VKS cluster context, e.g.:
vcf context use cluster-vks-kong

# Update helm and add Kong repo
```

```
helm repo add kong https://charts.konghq.com
helm repo update kong
```

Step 3: Create the namespaces ‘kong’ and ‘kong-system’

We create new namespaces for the installation of Kong and set the security profile to ‘baseline’

```
# Create namespaces
kubectl create namespace kong
kubectl create namespace kong-system

# Label both for baseline security
kubectl label ns --overwrite kong pod-security.kubernetes.io/enforce=baseline
kubectl label ns --overwrite kong-system pod-security.kubernetes.io/enforce=baseline
```

Step 4: Create a secret for the Kong License

```
# Assuming licence is located in ‘license.json’
kubectl create secret generic kong-enterprise-license \
  --from-file=license=license.json \
  --namespace kong
```

Step 5: Install the Kong Operator

```
# Note image repo, update as needed
helm upgrade --install kong-operator kong/kong-operator -n kong-system \
  --set image.tag=2.0.6 \
  --set image.repository=harbor.lab/kong/kong-operator
  --set global.webhooks.options.certManager.enabled=true
```

Step 6: Defining the Data Plane Profile

In this step, we establish a GatewayConfiguration. This resource serves as the blueprint for our Data Plane, defining its physical characteristics, such as the specific container image and resource specs.

This configuration acts as a template for the Gateway resource (detailed in the next step). Once the Gateway is applied, the Operator will provision the corresponding pods (e.g., dataplane-kong-xxxxx) in the kong namespace, each featuring a high-performance proxy container based on the Kong 3.13 image. Note: kong/kong-gateway is the enterprise image, whereas the OSS image is kong/kong

```
# Apply GatewayConfiguration
kubectl apply -f - <<'EOF'
kind: GatewayConfiguration
apiVersion: gateway-operator.konghq.com/v2beta1
metadata:
  name: kong
  namespace: kong
spec:
  dataPlaneOptions:
    deployment:
```

```
    podTemplateSpec:
      spec:
        containers:
        - name: proxy
          image: kong/kong-gateway:3.13
EOF

# Verify Kong Operator message
kubectl -n kong-system get events -o json | \
  jq -r '.items[] |
  select(.message | contains("successfully applied Kong configuration")) |
  "\(.metadata.name)\t\(.message)"'

kong-operator-kong-operator-controller-manager-f5778d9cb-4v9pv.188d6b68f8799bbb        successfully applied
Kong configuration to https://192.168.153.67:8444
```

Step 7: Create Kong Gateway Class and Gateway

Here, we use the Kubernetes native gateway API to create a GatewayClass that points to the Kong gateway operator. We then create a gateway that consumes the GatewayClass that has both an https and http listeners. For TLS we add a secret ('kong-admin-tls') which we create in the next step.

```
# Create GatewayClass and Gateway
kubectl apply -f - <<'EOF'
kind: GatewayClass
apiVersion: gateway.networking.k8s.io/v1
metadata:
  name: kong
  namespace: kong
spec:
  controllerName: konghq.com/gateway-operator
  parametersRef:
    group: gateway-operator.konghq.com
    kind: GatewayConfiguration
    name: kong
    namespace: kong
---
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: kong
  namespace: kong
spec:
  gatewayClassName: kong
  listeners:
    - name: https
      hostname: kong-admin.content.tmm.broadcom.lab
      port: 443
      protocol: HTTPS
```

```
    tls:
      mode: Terminate
      certificateRefs:
        - kind: Secret
          name: kong-admin-tls
          group: ""
    allowedRoutes:
      namespaces:
        from: Same
  - name: http
    hostname: echo.content.tmm.broadcom.lab
    port: 80
    protocol: HTTP
    allowedRoutes:
      namespaces:
        from: Same
EOF
```

Step 8: Create a TLS Certificate

For TLS termination a certificate is required. Here, we create a self-signed certificate using cert-manager (note this is for testing only; for production usage, update to use a CA or other issuer. Refer to https://cert-manager.io/docs/configuration/)

```
# Here we'll create a self-signed cluster issuer
kubectl apply -f - <<'EOF'
apiVersion: cert-manager.io/v1
kind: ClusterIssuer
metadata:
  name: selfsigned
spec:
  selfSigned: {}
EOF

# Create a certificate using the self-signed issuer
kubectl apply -f - <<'EOF'
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: kong-admin-cert
  namespace: kong
spec:
  secretName: kong-admin-tls
  secretTemplate:
    labels:
      konghq.com/secret: "true"   # required for Kong Operator default selector
  dnsNames:
    - kong-admin.content.tmm.broadcom.lab
  issuerRef:
    kind: ClusterIssuer
```

```
  name: selfsigned   # Your Issuer/ClusterIssuer
EOF
```

Step 9: Gateway Status

Check the gateway is programmed and has an IP address

```
# Check the Gateway status
kubectl -n kong get gateway kong -o wide

NAME   CLASS   ADDRESS       PROGRAMMED   AGE
kong   kong    10.163.44.47   True         6d23h
```

Verification

Step 1: Add a Test Service

We download the 'echo' test service and apply it to our cluster. This test runs a HTTP/HTTPS server (and returns details of the connection). Details of this test container can be found in the Kong Github repository: https://github.com/Kong/go-echo

```
# Apply the Kong 'echo' demo
kubectl -n kong apply -f https://developer.konghq.com/manifests/kic/echo-service.yaml
```

Step 2: Create a Route to the Test Service

```
# Define the route for the demo
kubectl apply -f - <<'EOF'
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: echo-http
  namespace: kong
spec:
  parentRefs:
    - name: kong
      sectionName: echo-http
  hostnames:
    - echo.content.tmm.broadcom.lab
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /
      backendRefs:
        - name: echo
          port: 1027
EOF
```

Step 3: Test Using CURL

Here we fetch the LoadBalancer address for the kong-dp service and store it in the PROXY_IP environment variable:

```
# Obtain the IP Address of the loadbalancer
PROXY_IP=$(kubectl -n kong get svc -o jsonpath=\
'{.items[?(@.spec.type=="LoadBalancer")]'\
'.status.loadBalancer.ingress[0].ip}')

# http test
curl -i http://$PROXY_IP/ -H 'Host: echo.content.tmm.broadcom.lab'

# https test
curl -ik https://$PROXY_IP/ -H 'Host: echo.content.tmm.broadcom.lab'
```

If we have successfully setup the Kong components, this should produce output similar to:

```
HTTP/1.1 200 OK
Content-Type: text/plain; charset=utf-8
Content-Length: 190
Connection: keep-alive
Date: Tue, 03 Feb 2026 15:53:23 GMT
Server: kong/3.13.0.1-enterprise-edition
X-Kong-Upstream-Latency: 4
X-Kong-Proxy-Latency: 1
Via: 1.1 kong/3.13.0.1-enterprise-edition
X-Kong-Request-Id: b967196f0915422ba2b2315d5756ed35

Welcome, you are connected to node kubernetes-cluster-kmnr-kubernetes-cluster-kmnr-nodepool-eck6qq.
Running on Pod echo-6f4786c4b4-scth9.
In namespace kong.
With IP address 192.168.152.43.
```

Kong Konnect

In this model, the architecture is decoupled to balance centralized management with localized performance. We utilize a hosted Control Plane (via Kong Konnect) to manage and push configurations to a local Data Plane residing within the VKS cluster. This ensures that while management is centralized in the cloud, all sensitive API traffic remains within your on-premises environment, processed at sub-millisecond speeds.

Prerequisites:

- **VKS Cluster:** Version 3.5 or later.
- **Cert-manager add-on:** Required for automated mTLS certificate rotation between planes.
- **VCF CLI & Kubectl:** For cluster interaction and Supervisor management.
- **Kong Gateway Enterprise Licence**: Optional
- **Kong Konnect Account:** To provide the SaaS-based Control Plane.
- **Connectivity:** Outbound internet access from the VKS cluster to the Konnect API (typically port 443 and 8444).

First, we follow steps 1-6 in the previous Kong On-premises Installation section to add the Kong operator.
Step 7: Configure API Connectivity

```
# Store the Konnect token
export KONNECT_TOKEN='xxxxx'
```

```
# Define API endpoint
kubectl apply -f - <<'EOF'
kind: KonnectAPIAuthConfiguration
apiVersion: konnect.konghq.com/v1alpha1
metadata:
  name: konnect-api-auth
  namespace: kong
spec:
  type: token
  token: "'$KONNECT_TOKEN'"
  serverURL: eu.api.konghq.com  # adjust for locale (us,au,eu)
EOF
```

Step 8: Declarative Provisioning of the Remote Control Plane

In this step, we deploy the KonnectGatewayControlPlane resource. This serves as a local declarative proxy, enabling the Kong Operator to provision and synchronize the remote control plane's state directly from the VKS cluster.

```
# Define local version of control plane
kubectl apply -f - <<'EOF'
kind: KonnectGatewayControlPlane
apiVersion: konnect.konghq.com/v1alpha2
metadata:
  name: gateway-control-plane
  namespace: kong
spec:
  createControlPlaneRequest:
    name: gateway-control-plane
    cluster_type: CLUSTER_TYPE_K8S_INGRESS_CONTROLLER
  konnect:
    authRef:
      name: konnect-api-auth
EOF
```

Step 9: Configuring the Secure Management Link

With the control plane proxy established, the KonnectExtension resource creates a secure connection between the local VKS infrastructure and the cloud management layer. It handles the critical task of mTLS certificate provisioning, ensuring that any Data Plane pods deployed later have the verified identity required to communicate with Konnect.

By setting provisioning: Automatic, we leverage cert-manager to handle the heavy lifting of security. The resource references the gateway-control-plane created in the previous step, finalizing the link between the VKS namespace and the specific remote control plane instance.

```
# Define mgmt link to the cloud endpoint
kubectl apply -f - <<'EOF'
kind: KonnectExtension
apiVersion: konnect.konghq.com/v1alpha2
metadata:
  name: my-konnect-config
```

```
  namespace: kong
spec:
  clientAuth:
    certificateSecret:
      provisioning: Automatic
  konnect:
    controlPlane:
      ref:
        type: konnectNamespacedRef
        konnectNamespacedRef:
          name: gateway-control-plane
EOF
```

Step 10: Defining the Data Plane Profile

Like the on-premises deployment, in this step, we establish a GatewayConfiguration. This resource serves as the blueprint for our Data Plane, defining its physical characteristics. Here we add further details of the remote connection.

By applying this configuration, we prepare the VKS environment to deploy high-performance proxy pods. While this resource defines the "how" (using two replicas and Konnect connectivity), the actual deployment of the dataplane-kong-xxxxx pods will be triggered by the creation of a standard Kubernetes Gateway resource that references this configuration.

```
# Apply GatewayConfiguration
kubectl apply -f - <<'EOF'
kind: GatewayConfiguration
apiVersion: gateway-operator.konghq.com/v2beta1
metadata:
  name: kong
  namespace: kong
spec:
  extensions:
  - kind: KonnectExtension
    name: my-konnect-config
    group: konnect.konghq.com
  dataPlaneOptions:
    deployment:
      replicas: 2
      podTemplateSpec:
        spec:
          containers:
          - name: proxy
            image: kong:3.13'
EOF
```

We then follow the on-premises configuration from step 7 (‘Create Kong Gateway Class and Gateway’) onwards. The verification steps remain the same.

Conclusion

In summary, we have demonstrated the deployment of Kong on the vSphere Kubernetes Service (VKS) through two distinct architectural models: *on-premises* and *hybrid*. The on-premises configuration localizes both the Control Plane and Data Plane within the VKS cluster, offering a self-contained solution rigorous enough for high-compliance environments where management layers must remain on-premises. Alternatively, the hybrid model, enabled by Kong Konnect, decouples the architecture by hosting the Control Plane as a SaaS service while retaining the Data Plane locally within the VKS infrastructure. This approach ensures that while operational management is centralized in the cloud, all sensitive API traffic remains within the consumer's infrastructure boundary, benefiting from low-latency processing and strict data sovereignty.

The strategic integration of Kong within the VKS ecosystem addresses the critical challenge of operating modern, ephemeral workloads alongside established enterprise systems. By functioning as the intelligent "front door" for VKS, Kong leverages a lightweight core to deliver sub-millisecond latency and dynamic scalability that legacy gateways cannot sustain. This architecture empowers organizations to bridge the gap between traditional infrastructure and cloud-native agility; IT teams can manage Kubernetes clusters using familiar vSphere workflows while simultaneously enforcing Zero-Trust security principles and declarative, code-based API policies. Consequently, layering Kong over the enterprise-grade VKS clusters transforms basic connectivity into a fully observable, secure, and optimized service delivery platform.

Appendix A:
Example Client VM Configuration & Tooling

Here, we used an Ubuntu Jammy virtual machine. At the time of writing, the OVA image can be obtained from:
https://cloud-images.ubuntu.com/jammy/current/jammy-server-cloudimg-amd64.ova

The VCF command-line can be downloaded and installed thus:

```
# Download VCF CLI & install (version 9.0)
curl -fsSL https://packages.broadcom.com/artifactory/vcf-distro\
  /vcf-cli/linux/amd64/v9.0.0/vcf-cli.tar.gz | tar xz

sudo cp vcf-cli-linux_amd64 /usr/local/bin/vcf

# (Optional) Add autocomplete for VCF command line (bash shell)
echo "source <(vcf completion bash)" >> ~/.bashrc

# Create context & login
vcf context create --endpoint=<supervisor endpoint> \
  --username administrator@vsphere.local

# Use context
vcf context use supervisor-namespace
```

To trust our connection to vCenter, we install the certificate:

```
# Get vCenter certs & install
# Download the zip file to /tmp using curl (insecure mode required)
VCENTER_IP=<vCenter IP>
curl -k -fsSL -o /tmp/vccert.zip https://${VCENTER_IP}/certs/download.zip

# Unzip and copy to SSL directory
unzip /tmp/vccert.zip -d /tmp
sudo cp /tmp/certs/lin/* /etc/ssl/certs

# Update system certs
sudo update-ca-certificates
```

Optionally install kubectl and vSphere plugin:

```
# (Optional) Get Kubectl & vSphere plugin
# Download the zip file to /tmp using curl
SUPERVISOR_ENDPOINT="<supervisor endpoint>"
curl -fsSL -o /tmp/vsphere-plugin.zip \
  https://${SUPERVISOR_ENDPOINT}/wcp/plugin/linux-amd64/vsphere-plugin.zip

# Unzip and copy to local bin directory
unzip /tmp/vsphere-plugin.zip -d /tmp
sudo cp /tmp/bin/* /usr/local/bin


# (Optional) Add autocomplete and shorthand 'k' for Kubectl (bash shell)
```

```
cat << 'EOF' >> ~/.bashrc
if command -v kubectl &> /dev/null; then
    source <(kubectl completion bash)
    alias k=kubectl
    complete -o default -F __start_kubectl k
fi
EOF

source ~/.bashrc
```

Finally, we install Helm by following the instructions:
https://helm.sh/docs/intro/install/

```
# Get Helm using download script
curl -fsSL -o /tmp/get_helm.sh \
  https://raw.githubusercontent.com/helm/helm/main/scripts/get-helm-4

chmod +x /tmp/get_helm.sh

/tmp/get_helm.sh
```

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