TECHNICAL WHITE PAPER
October 2026



Traefik on vSphere Kubernetes Service on VMware Cloud Foundation

Reference Architecture

Table of Contents

Executive Summary **3**
vSphere Kubernetes Service 3
Traefik Labs 4
Solution Architecture **7**
VKS Cluster Configuration 8
Kubernetes Packages 10
Workloads Outside the Cluster 11
Licensing and Connectivity Modes 12
Solution Validation **14**
Model 1: Traefik Proxy 14
Prerequisites 15
Installation 15
Verification 18
Model 2: Traefik Hub (API Management, AI Gateway and MCP Gateway) 20
Prerequisites 20
Installation 21
Verification: AI Gateway 24
Verification: MCP Gateway 26
Verification: API Management Installation 31
Model 3: Traefik Across VMs and Containers 32
Prerequisites 33
Installation 33
The routing contract 33
Child A: vCenter-managed virtual machines 34
Child B: stand-alone VMs through VM Service 36
Parent gateway on the VKS cluster 37
Verification: Traffic Migration Between Control Planes 38
Conclusion **40**

Executive Summary

Enterprises are standardizing their modern applications, and increasingly their AI workloads, on VMware Cloud Foundation (VCF) so they can run them on infrastructure they already govern. That move puts a new problem in front of platform teams. Every microservice, model endpoint and MCP tool server exposed on a vSphere Kubernetes Service (VKS) cluster opens another network path that has to be authenticated, authorized and rate limited. When each application team handles this separately, controls drift, credentials sprawl, and the gaps show up later in an incident review, a failed audit, or a model-usage bill nobody forecasted. AI agents raise the stakes because they call APIs and tools on their own, at machine speed, on behalf of users. The durable fix is to enforce policy once, at the cluster edge: a single gateway that applies the same identity, access and usage controls to every request, whether it comes from a browser, a service or an agent, before it reaches a workload.

Traefik provides that gateway on VKS. Traefik Proxy and Traefik Hub install from one Helm chart and share one routing model. A platform team can start with Traefik Proxy (open-source ingress) and upgrade the same release in place to Traefik Hub, which adds API Management, an AI Gateway and an MCP Gateway for agentic workloads. Traefik Labs validates the deployment on VKS 3.7.0 with Traefik Hub v3.21.0.

Key advantages of Traefik on VKS:

- **Policy as custom resources.** OpenID Connect and JWT middlewares, mutual TLS through TLS options, a native Coraza WAF and distributed rate limits live in Git next to the application manifests, so a policy change goes through the same review as a change to a Deployment.
- **Usage control for LLM and agent traffic.** Token-aware rate limits and quotas per consumer, model and parameter governance, and content guards keep AI spend and behavior inside limits the platform team sets. Task-Based Access Control restricts which MCP tools each agent can call.
- **Air-gapped operation.** Offline Mode, a licensed feature, validates the token with no platform connection, so the API, AI and MCP gateways run without the Traefik Hub platform. FIPS 140-3 and hardened "Distro Zero" images target regulated environments.
- **VMs and containers behind one gateway.** The multi-cluster provider (Early Access) connects gateways that run beside vCenter-managed and Supervisor-managed virtual machines to the cluster gateway. To move a workload between them, you change a weight on that gateway.
- **One chart, one upgrade path.** A license token switches the release to the Traefik Hub image, and you turn on each gateway feature with a Helm value. Proxy Mode runs the Traefik Hub image with no token.
- **Kubernetes-native routing.** Traefik reads VKS services through its own custom resources and the Kubernetes Gateway API v1.5.1. It updates its routes as pods scale or move, with no restart.

vSphere Kubernetes Service

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

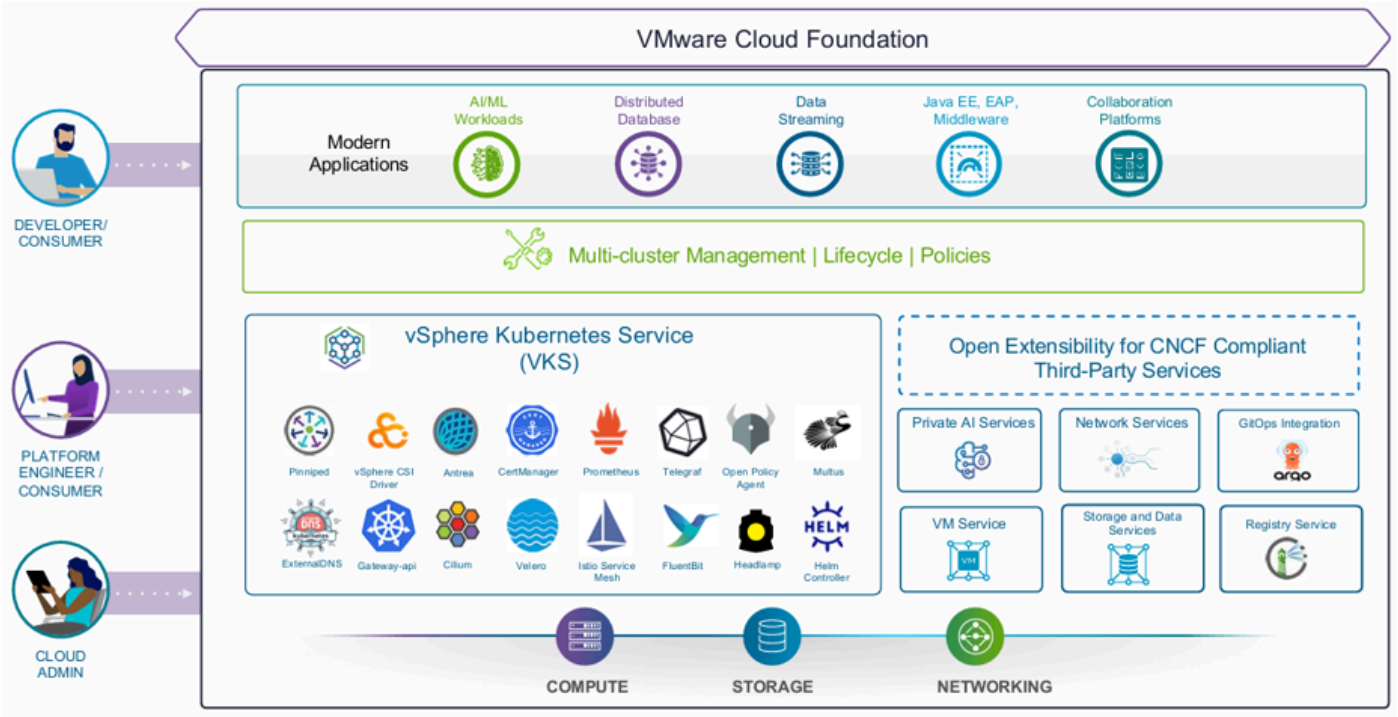


Figure 1: VKS on VCF Ecosystem

Benefits of using VKS with modern applications:

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

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

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

Traefik Labs

Traefik Proxy is an open-source reverse proxy and Kubernetes ingress controller. It watches the Kubernetes API through its own custom resources (`IngressRoute`, `Middleware`, `TraefikService`) and through the Kubernetes Gateway API. It updates its routing table when a pod starts or stops. A team attaches authentication or a rate limit to a route with a small declarative resource.

Traefik Hub is the commercial gateway from Traefik Labs. It embeds Traefik Proxy and installs from the same Helm chart, with its own image. A license token adds:

- **API Management.** A developer portal, an API catalog and consumption plans, declared as Kubernetes custom resources.
- **AI Gateway.** Request-level control over LLM traffic: model and parameter governance, semantic caching, content and LLM guards, and token-aware rate limits and quotas.
- **MCP Gateway.** An OAuth 2.1 resource server in front of Model Context Protocol servers. It applies Task-Based Access Control (TBAC) to the tools an AI agent may call.
- **Security middlewares.** OpenID Connect authentication, JWT validation, a native Coraza WAF and distributed rate limiting. They work in connected mode and in Offline Mode.
- **Multi-cluster routing (Early Access).** A parent gateway that routes to workloads advertised by child gateways on other clusters or on virtual machines.

For regulated environments, Traefik Hub v3.21.0 adds a FIPS 140-3 image and a hardened image, "Distro Zero", built from scratch with no shell and no package manager.

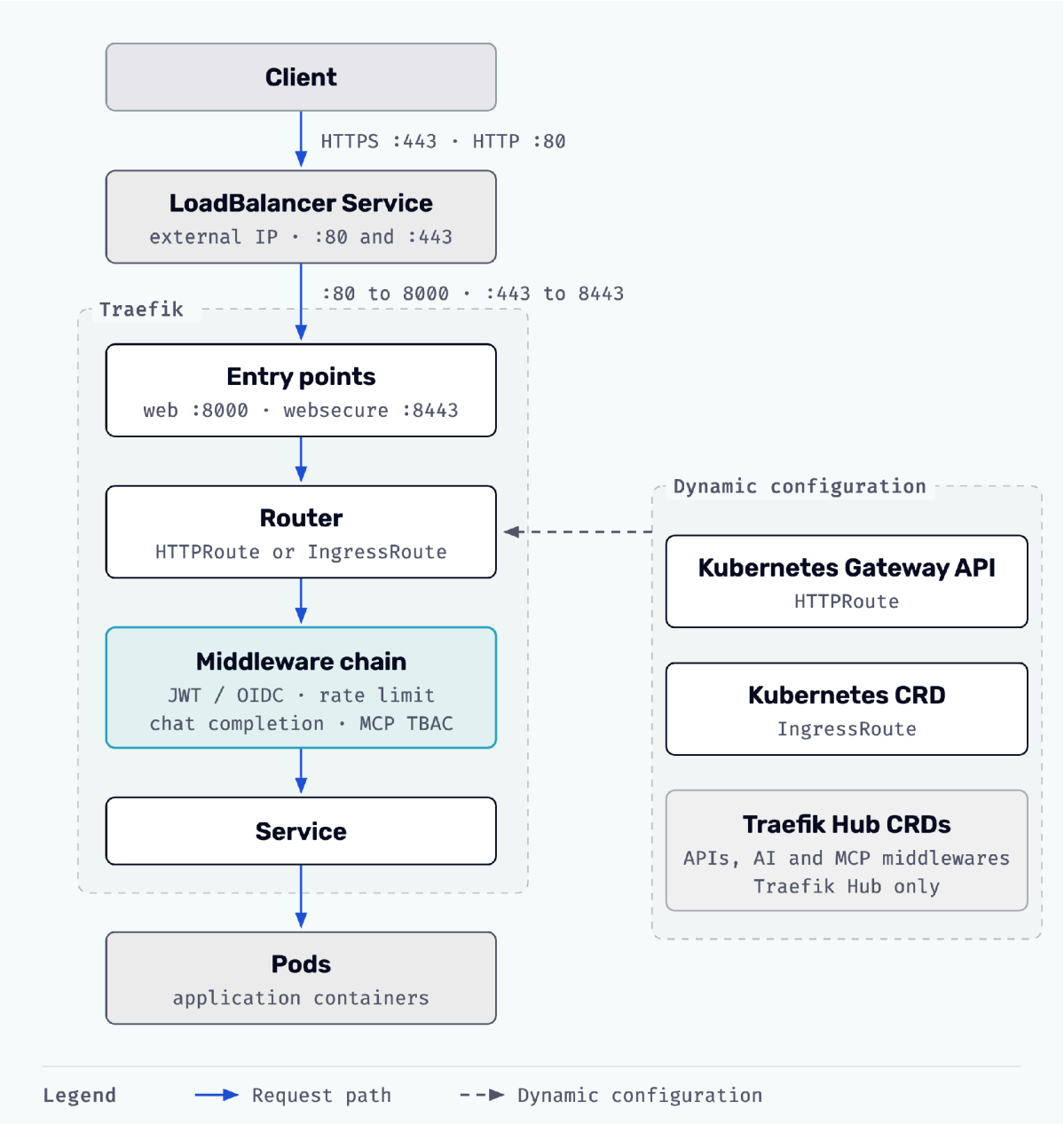


Figure 2: Request flow through Traefik on VKS

Figure 2 shows one request path. The client reaches the LoadBalancer Service, an entry point accepts the connection and a router matches the request. The middleware chain applies the policy, and Traefik forwards the request to the workload pods. The Kubernetes providers feed routing changes to the gateway without a restart.

Solution Architecture

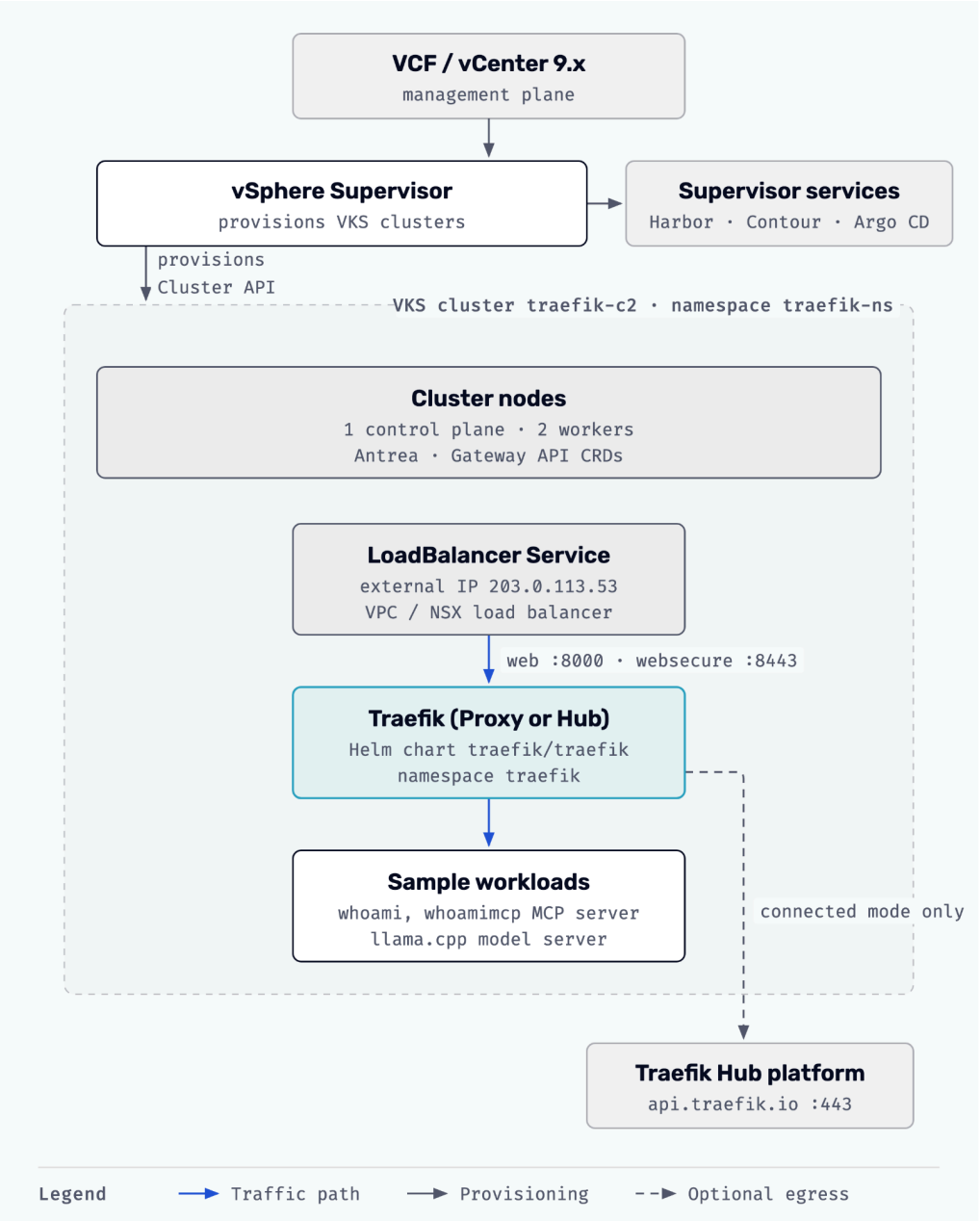


Figure 3: Traefik on a VKS workload cluster

Figure 3 shows the deployment: the vSphere Supervisor provisions the VKS workload cluster traefik-c2 through Cluster API, and Helm installs Traefik onto that cluster. A LoadBalancer Service exposes the gateway on an NSX VPC address. The link to the Traefik Hub platform exists in connected mode; this validation runs with an offline token.

At its core, the Supervisor control plane provides essential services, such as 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 white paper is 3.7.0.

Table 1: Installed Supervisor Components

Component	Version	Notes
vCenter Server	9.0.2 (build 25148086)	Management plane; API version 9.0.0.0
vSphere Supervisor	Kubernetes v1.32.9+vmware.2-fips	Hosts the vSphere Namespace `traefik-ns`
vSphere Kubernetes Service	3.7.0	Add-on repository `vks-addons-3.7.0-20260723` ClusterClass `builtin-generic-v3.7.0` Cluster API `v1beta2`
VM Service	vmoperator.vmware.com	Reconciles stand-alone `VirtualMachine` objects (Model 3)

The example manifests, Helm values, and supporting configuration referenced throughout this paper are available in the accompanying repository:

https://github.com/vmware/vks-validated-solutions/tree/main/Traefik

VKS Cluster Configuration

The cluster contains one control plane node and two worker nodes. Storage is vSAN Express Storage Architecture (ESA) with the default RAID 5 storage policy. Transparently, the vSphere CSI driver exposes this policy to an adjacent Kubernetes storage class, vsan-esa-default-policy-raid5.

Control plane and worker nodes use the best-effort-medium VM class. The cluster runs in the vSphere Namespace traefik-ns, on an NSX VPC; the example subnet of the VM Service VMs is 198.51.100.0/24. LoadBalancer Services receive their addresses from the external IP block of the VPC.

Table 2: Kubernetes Cluster Configuration Summary

Component	Configuration	Notes
Kubernetes Control Plane	1 replica 2 vCPU, 8 GiB RAM vSAN ESA RAID 5	VM class `best-effort-medium` Storage class `vsan-esa-default-policy-raid5`
Kubernetes Worker Nodes	2 replicas 2 vCPU, 8 GiB RAM 19.3 GiB node disk vSAN ESA RAID 5	VM class best-effort-medium Storage class `vsan-esa-default-policy-raid5`
Kubernetes Release	v1.36.2+vmware.2	KubernetesRelease `v1.36.2---vmware.2-vkr.3` containerd 2.3.2 (FIPS build)
OS Image	Ubuntu 24.04.4 LTS	Kernel 6.17.0-35-generic

The VKS cluster configuration is defined declaratively in manifests/vks.yaml, shown here in full. As previously outlined, applying this manifest to the Supervisor triggers the creation of the cluster, which is then managed and reconciled by the Cluster API.

Shell

```
# The VKS cluster traefik-c2 used for this validation, applied to the Supervisor with -n
# (your vSphere Namespace, guide 2 step 5). Cluster API reconciles it into a cluster.
apiVersion: cluster.x-k8s.io/v1beta2
kind: Cluster
metadata:
  name: traefik-c2
spec:
  clusterNetwork:
    pods:
      cidrBlocks: ["192.168.0.0/16"]
    services:
      cidrBlocks: ["10.96.0.0/12"]
    serviceDomain: cluster.local
  topology:
    classRef:
      name: builtin-generic-v3.7.0
      namespace: vmware-system-vks-public
    version: v1.36.2+vmware.2
    controlPlane:
      replicas: 1
      metadata:
        annotations:
          run.tanzu.vmware.com/resolve-os-image: os-name=ubuntu,os-version=24.04
    workers:
      machineDeployments:
        - class: node-pool
          name: node-pool-1
          replicas: 2
          metadata:
            annotations:
              run.tanzu.vmware.com/resolve-os-image: os-name=ubuntu,os-version=24.04
    variables:
      - name: vmClass
        value: best-effort-medium
      - name: storageClass
        value: vsan-esa-default-policy-raid5
      - name: vsphereOptions
        value:
          persistentVolumes:
            defaultStorageClass: vsan-esa-default-policy-raid5
      - name: kubernetes
        value:
          certificateRotation:
            enabled: true
            renewalDaysBeforeExpiry: 90
          security:
            podSecurityStandard:
              deactivated: false
              enforce: baseline
      - name: bootstrapAddons
        value:
          cniRef:
            name: antrea
```

```
        namespace: vmware-system-vks-public
```

The Supervisor serves `cluster.x-k8s.io/v1beta2`, which references the ClusterClass through `spec.topology.classRef`. The namespace `vmware-system-vks-public` contains the ClusterClasses `builtin-generic-v3.1.0` to `v3.7.0`.

Kubernetes Packages

VKS clusters (via the VKr release) include a default set of core packages, such as Antrea for networking and Pinniped for authentication.

For this deployment, cert-manager is installed with its Helm chart, and Traefik is installed using Helm with the official Traefik Labs chart. Traefik Proxy and Traefik Hub use the same chart. Traefik supports Gateway API v1.5.1 and has a published conformance report for that version. From the Standard channel, Traefik supports `HTTPRoute` (all core and some extended features), `GRPCRoute`, `TLSRoute` and `BackendTLSPolicy`. From the Experimental channel, it supports `TCPRoute`. VKS ships the Standard channel, and the installation in Model 1 adds the Experimental-only `TCPRoute` and `UDPRoute` CRDs from the upstream v1.5.1 bundle.

Table 3: Kubernetes Cluster Packages

Package	Version	Notes
Antrea	2.6.2	VKS core package
Gateway API	1.5.1 (Standard channel)	VKS core package; `TCPRoute` and `UDPRoute` CRDs added from the v1.5.1 Experimental bundle
Guest-cluster auth-service	1.4.8	VKS core package
Helm Controller	1.5.5	VKS core package
Metrics Server	0.8.1	VKS core package
Pinniped	0.46.0	VKS core package
Secretgen Controller	0.21.1	VKS core package
vSphere CPI	1.36.0	VKS core package
vSphere CSI	3.8.0	VKS core package
cert-manager	v1.18.2	jetstack Helm chart Alternative: VKS package `cert-manager.kubernetes.vmware.com` (1.18.2 to 1.20.3)
Traefik Helm chart	41.6.1	Installs Traefik Proxy or Traefik Hub; repository `https://traefik.github.io/charts`
Traefik Proxy	v3.7.13	Chart `appVersion`; image `docker.io/traefik` (Model 1)
Traefik Hub	v3.21.0 (GA)	Image `ghcr.io/traefik/traefik-hub` Chart default when `hub.token` is set (Models 2 and 3)

Package	Version	Notes
		Model 3 child gateways: Early Access build of Traefik Hub v3.21 (vsphere and vmoperator providers), provided by Traefik Labs on request (https://info.traefik.io/en/request-demo).

Workloads Outside the Cluster

A VKS cluster does not hold all the workloads of an estate. vCenter manages the virtual machines in a resource pool. The vSphere Supervisor manages stand-alone VMs through VM Service, with one `VirtualMachine` object for each VM. Different teams administer the two estates with different credentials.

Traefik Hub covers both with its multi-cluster provider, in a parent and child model (Figure 4). The Traefik Hub gateway on the VKS cluster is the parent. A Traefik Hub child runs on a virtual machine in each estate. The child discovers the machines of its estate through the `vsphere` provider (against vCenter) or the `vmoperator` provider (against the Supervisor). Each provider turns the discovered machines into services; the child's file provider declares the routers and the Uplinks that reference these services.

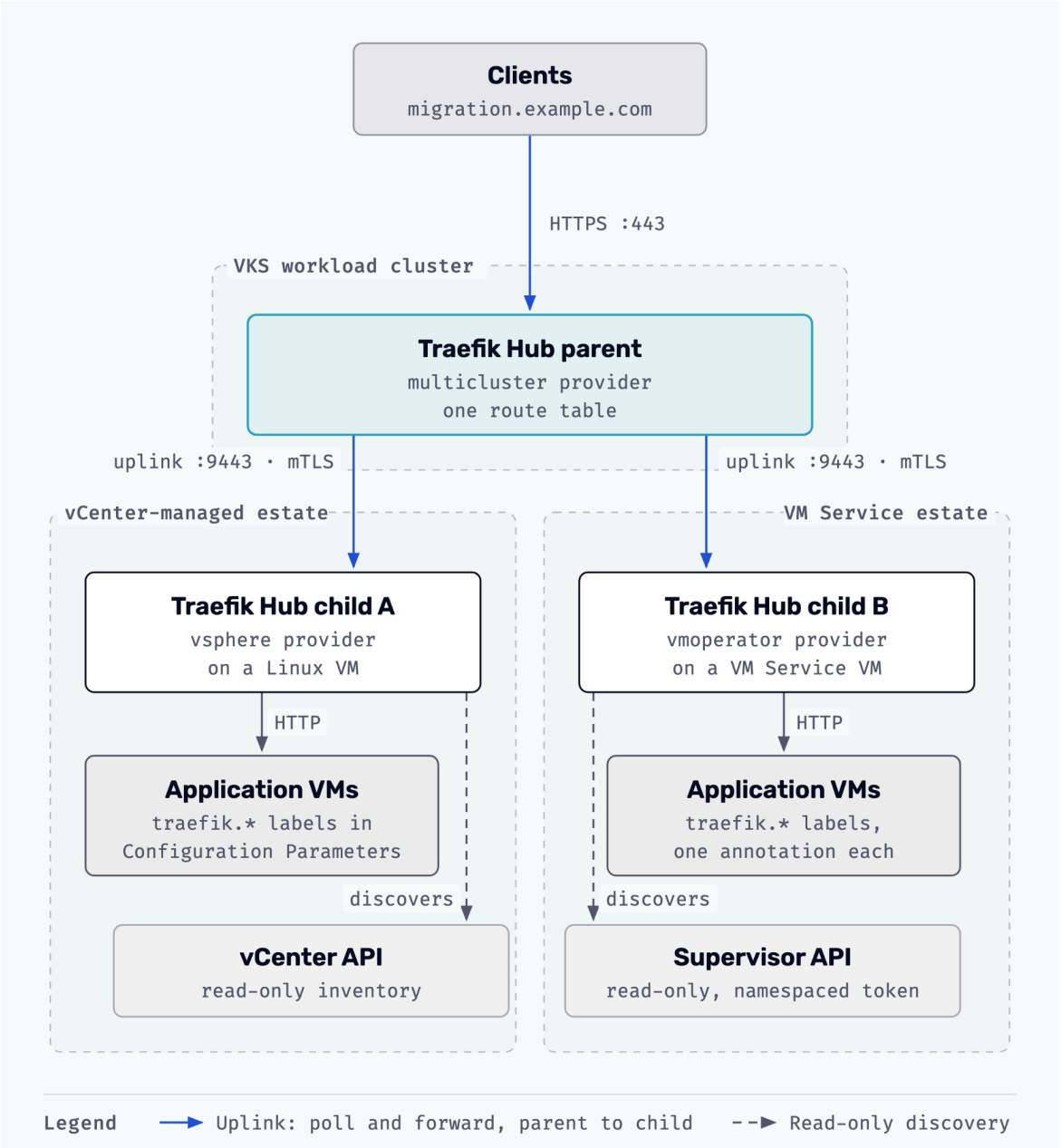


Figure 4: Parent and child topology across two VM estates

The parent polls the uplink entry point of each child (`GET /api/uplinks`). It forwards the matching client traffic to the child over the same entry point. Traffic flows from parent to child only. Each child connects only to its own control plane, so no credential crosses an estate boundary.

- **The routing contract lives with the workload.** A machine declares its services in its own metadata. The estate team decides which services a machine exposes, and the platform team decides how the parent distributes the traffic.
- **One route table, weighted.** For each Uplink name the parent builds a weighted service across all children that advertise it, plus one per-child service (`<uplink>-<child>@multicluster`). A migration between estates is a weight change on the parent.
- **Security is explicit.** The uplink entry point is an internal API. Make it reachable from the parent alone. Use mutual TLS to authenticate the parent. Both VM providers are read-only against their control plane.

> **Important.** Availability. The multi-cluster provider is Early Access (Traefik Hub v3.21.0). It requires the `multi-cluster` feature in the license token of the parent and of each child.

Licensing and Connectivity Modes

A Traefik Hub license is a token from the Traefik Hub Online Dashboard (hub.traefik.io). The gateway runs in one of two connectivity modes. You select the mode when you create the gateway in the dashboard (Figure 5).

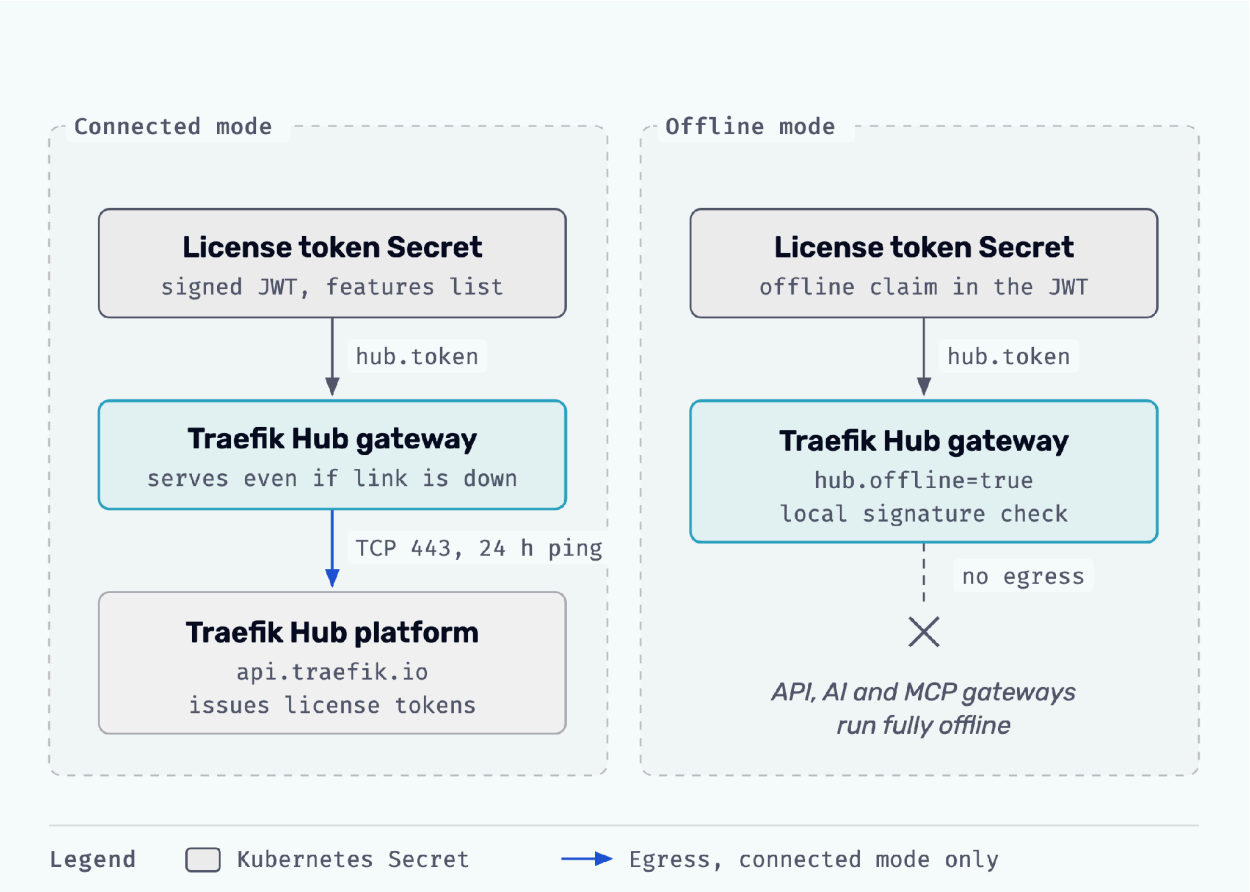


Figure 5: Connected and offline connectivity modes

- **Connected mode.** The gateway validates the token against the Traefik Hub platform at `api.traefik.io` on TCP 443 and checks in every 24 hours. The gateway stops when the platform reports a disabled token, missing features, or a token in use by another gateway. The dashboard shows the gateway fleet, and API Management synchronizes its portal through this channel. Traefik Hub v3.21.0 has downtime-tolerant token validation: the gateway serves traffic while the platform is unreachable and retries the check-in.
- **Offline Mode.** Offline Mode is a licensed feature offered by Traefik Labs, and you can select it only at gateway creation. Start the gateway with `hub.offline=true` and a token with the `offline` claim. The gateway validates the token with no platform connection and enforces its expiration date. An expired token prevents the gateway from starting up but a gateway that is already running enters a 30-day grace period. The API Gateway, the AI Gateway, the MCP Gateway and API Management (Kubernetes only) can work in both online and offline mode.

Table 4: Gateway egress requirements

Destination	Port	Purpose	Mode
`api.traefik.io`	TCP 443	License validation, 24-hour check-in Error-log export (`hub.sendlogs`) API Management synchronization	Connected
Plugin registry	TCP 443	Plugin download, if you use plugins	If configured
Child uplink entry points	TCP 9443 (example)	Multi-cluster discovery and forwarding, parent to child	Model 3
vCenter (from child A)	TCP 443	Read-only VM inventory (`vsphere` provider)	Model 3
Supervisor API (from child B)	TCP 443	Read-only `VirtualMachine` list (`vmoperator` provider)	Model 3

Two Helm values control the other outbound calls. With `global.checkNewVersion` (default `true`), the gateway checks for new releases. Set it to `false` in air-gapped estates. `global.sendAnonymousUsage` (default `false`) controls anonymous usage statistics.

> **Note.** Token lifetime. License tokens have a default lifetime of one year, configurable per workspace. Rotate the token from the dashboard before it expires. After a rotation, the old token stays valid for up to 5 days. To give an offline gateway a new token, update the Secret and restart the gateway.

Proxy Mode (Traefik Hub v3.21.0) runs the Traefik Hub image without a token, as a plain Traefik Proxy. Set `hub.enabled=true` and `image.tag=v3.21.0`, and do not set `hub.token`. A team can then use one image from the start and add the license later.

Traefik Hub v3.21.0 has two image variants for regulated estates:

- **FIPS 140-3 image.** The tag suffix is `-fips` (for example `v3.21.0-fips`). It uses the Go Cryptographic Module, which the Cryptographic Module Validation Program validated. It requires the `fips` feature in the license token. FIPS mode turns off TLS 1.3 and HTTP/3.
- **Hardened image ("Distro Zero").** The image holds one static binary built from scratch, with no shell and no package manager. It runs as uid 65532 on ports 8000 and 8443. The image is `registry.traefik.io/traefik-hub` with the tag suffix `-hardened` or `-fips-hardened` (Helm `hub.hardened=true`). The pull needs the hardened-image feature on the workspace and an image pull secret (`deployment.imagePullSecrets`). The hardened image also runs without a token in Proxy Mode.

Solution Validation

This paper validates three deployment models. Models 1 and 2 run inside the VKS cluster and differ by the image and the license token. Model 3 extends the same gateway to virtual machines outside the cluster. The validation uses Traefik Hub v3.21.0 with the Traefik Helm chart 41.6.1. The multi-cluster provider in Model 3 is Early Access in v3.21.0.

The complete procedure of each model, with every command, file and expected output, is in the Traefik folder of the VKS validated solutions repository (https://github.com/vmware/vks-validated-solutions/tree/main/Traefik): Traefik/3A_TRAEFIK_PROXY.md, Traefik/3B_TRAEFIK_HUB.md and Traefik/3C_TRAEFIK_VMS_MULTICLUSTER.md, with the manifests and Helm values under Traefik/manifests/. This section shows the essential commands of each step and what to expect, with the step numbers of the guides. Each guide starts with a step that lists the values of your environment in a table, sets them as variables and reads the secrets at a prompt; the helper functions (sign_jwt, mcp, dashboard) and the generated files ($WORK/gateway-ca.crt, $WORK/big.json) come from the guides; the SSH and Compose configuration of Model 3 and the credential files of the dashboard and MCP checks go to $WORK. Names and addresses in this chapter belong to an example environment (domain example.com, vCenter vcenter.example.com, the networks 192.0.2.0/24, 198.51.100.0/24 and 203.0.113.0/24 reserved for documentation); the validation ran on a Broadcom lab, whose values were replaced one for one in the outputs. The files under manifests/ use the example environment; the guides adapt the few environment-specific values with the variables of their first step.

Model 1: Traefik Proxy

Traefik Proxy runs inside the VKS cluster with no external control plane and no license. Model 1 offers governed ingress and Kubernetes-native routing, with an in-place upgrade path to Traefik Hub. Figure 6 shows the components that this model deploys.

The full procedure for Model 1 is Traefik/3A_TRAEFIK_PROXY.md.

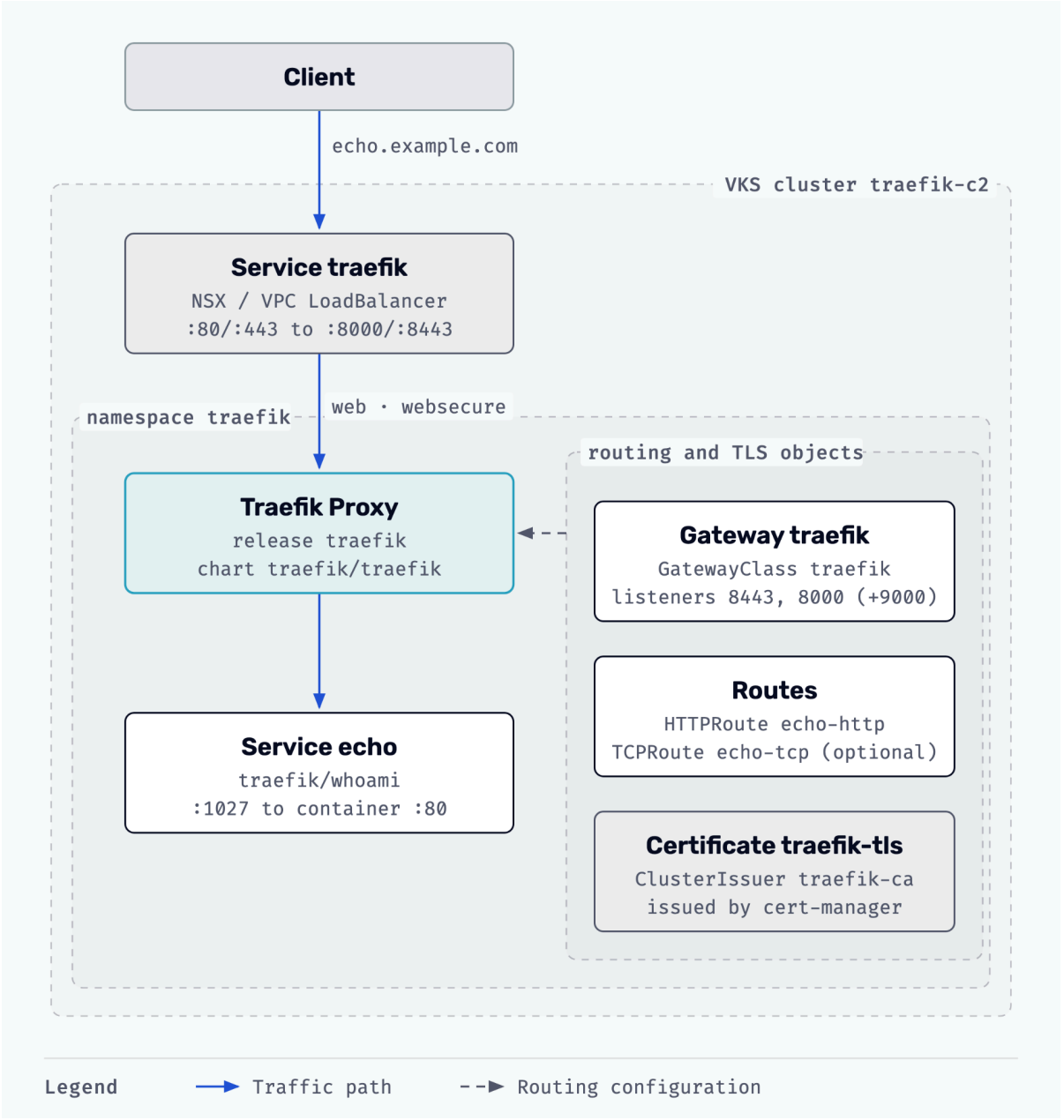


Figure 6: Components that Model 1 deploys

Prerequisites

- A VKS cluster. The reference environment runs VKS 3.7.0.
- The VCF CLI, kubectl, Helm, curl, jq and openssl on the administrator workstation.
- Gateway API CRDs: VKS provides v1.5.1 on the Standard channel; Step 2 adds the Experimental-only CRDs.
- cert-manager, installed in Step 5: it issues the certificate traefik-tls that the HTTPS listener of the Gateway and every HTTPS check in Models 1 to 3 use.

Installation

Step 1: Set the environment variables.

The commands run from the Traefik/ folder of the repository, in one shell session, with the VKS cluster context of Traefik/2_VKS_DEPLOYMENT.md (step 7) active. That guide trusts the Supervisor and vCenter certificates first: the Supervisor

presents a certificate signed by the vCenter certificate authority (VMCA) and names itself by IP address only, so the VCF CLI gets the vCenter roots with --ca-certificate and https://<Supervisor IP> as the endpoint. Do not skip TLS verification. DOMAIN is the domain of the routes (no DNS record is needed) and WORK a working directory outside the repository for certificates and downloads.

```shell
# bash
export DOMAIN="<your_domain>"
export CHART_VERSION="41.6.1"
export WORK="$HOME/traefik-vks-work"; mkdir -p "$WORK"; chmod 700 "$WORK"
```

Step 2: Add the Experimental-channel Gateway API CRDs.

VKS protects its Standard-channel v1.5.1 CRDs with the safe-upgrades ValidatingAdmissionPolicy. VKS also installs these CRDs with its package manager, which owns their fields: the Experimental bundle creates only the CRDs that do not exist yet (TCPRoute, UDPRoute and two x-k8s.io CRDs) and reports a field-ownership conflict for each VKS-managed object. This is expected; do not force the apply.

```shell
# bash
GWAPI=https://github.com/kubernetes-sigs/gateway-api/releases/download/v1.5.1
kubectl apply --server-side -f "$GWAPI/experimental-install.yaml" \
  || echo "exit $? (expected: conflicts with the VKS-managed CRDs)"
```

The VKS-managed CRDs stay at v1.5.1 Standard. Traefik uses `TCPRoute` from this set and does not use `UDPRoute`.

Step 3: Add the Traefik Helm repository and create the `traefik` namespace.

```shell
# bash
helm repo add --force-update traefik https://traefik.github.io/charts
kubectl create namespace traefik --dry-run=client -o yaml | kubectl apply -f -
kubectl label namespace traefik --overwrite pod-security.kubernetes.io/enforce=baseline
```

Step 4: Install Traefik with the Gateway API provider.

The chart also creates the GatewayClass traefik and a default Gateway traefik-gateway with a single HTTP listener on the web entry point; this validation uses its own Gateway traefik (step 6), which also terminates HTTPS on websecure with traefik-tls, and leaves the default one unused. `TCPRoute` needs the Experimental-channel flag.

```shell
# bash
helm upgrade --install traefik traefik/traefik -n traefik --version 41.6.1 \
  -f manifests/3a/values-proxy.yaml   # Gateway API provider, Experimental channel
```

The release deploys `docker.io/traefik:v3.7.13`, the chart's `appVersion`.

Step 5: Install cert-manager and create a TLS certificate.

This step applies manifests/3a/clusterissuer.yaml (a private CA for the validation) and manifests/3a/certificate.yaml (the wildcard Certificate traefik-tls), then saves the CA that clients use to verify the gateway. In production, reference an internal CA or another issuer that cert-manager supports.

```shell
# bash
helm repo add --force-update jetstack https://charts.jetstack.io

helm upgrade --install cert-manager jetstack/cert-manager -n cert-manager \
  --create-namespace --version v1.18.2 --set crds.enabled=true

kubectl apply -f manifests/3a/clusterissuer.yaml       # private CA for the validation

sed "s/example.com/$DOMAIN/g" manifests/3a/certificate.yaml | kubectl apply -f -

kubectl -n traefik wait certificate/traefik-tls --for=condition=Ready --timeout=120s

# The CA that clients use to verify the gateway (curl --cacert below)
kubectl -n traefik get secret traefik-tls -o jsonpath='{.data.ca\.crt}' \
  | base64 -d > "$WORK/gateway-ca.crt"
```

Step 6: Create the Gateway.

The listener ports must match the entry points `web` (8000) and `websecure` (8443), not the Service ports 80 and 443. Traefik rejects a listener on port 80 or 443 with `PortUnavailable`.

```shell
# bash
sed "s/example.com/$DOMAIN/g" manifests/3a/gateway.yaml | kubectl apply -f -

kubectl -n traefik get gateway traefik
# output
NAME      CLASS     ADDRESS         PROGRAMMED   AGE
traefik   traefik   203.0.113.53    True         15s
```

The Gateway is programmed on the NSX load-balancer address, and the certificate is ready.

Verification

Step 1: Deploy a test workload and its route.

This step applies the whoami Deployment and Service (manifests/3a/whoami.yaml) and the HTTPRoute (manifests/3a/httproute.yaml). The HTTPRoute has no sectionName, so it attaches to the http and https listeners.

```shell
# bash
kubectl apply -f manifests/3a/whoami.yaml

sed "s/example.com/$DOMAIN/g" manifests/3a/httproute.yaml | kubectl apply -f -
```

Step 2: Send test requests with curl.

The first command reads the LoadBalancer address of the traefik Service into TRAEFIK_IP, which the later steps use. --resolve sends the host name to that address, so no DNS record is needed; HTTPS is verified with the CA of Step 5.

```shell
# bash
TRAEFIK_IP=$(kubectl -n traefik get svc traefik \
  -o jsonpath='{.status.loadBalancer.ingress[0].ip}')

curl -sS --resolve "echo.$DOMAIN:80:$TRAEFIK_IP" -i "http://echo.$DOMAIN/" \
  | grep -E '^HTTP|^Hostname|Forwarded-Proto'
# output
HTTP/1.1 200 OK
Hostname: echo-7698b9cd96-s6bfs
X-Forwarded-Proto: http

# bash
curl -sS --cacert "$WORK/gateway-ca.crt" --resolve "echo.$DOMAIN:443:$TRAEFIK_IP" \
  -i "https://echo.$DOMAIN/" | grep -E '^HTTP|^Hostname|Forwarded-Proto'
# output
HTTP/2 200
Hostname: echo-7698b9cd96-s6bfs
X-Forwarded-Proto: https
```

Traefik routes the request to the whoami pod over HTTP and over TLS.

Step 3: Expose a TCP service through the Experimental channel (optional).

Open a `tcp` entry point on the chart. Then create and apply the Gateway with a `TCP` listener (`manifests/3a/gateway-tcp.yaml`) and the `TCPRoute` (`manifests/3a/tcproute.yaml`).

```shell
# bash
helm upgrade traefik traefik/traefik -n traefik --version 41.6.1 --reuse-values \
  -f manifests/3a/values-tcp.yaml

sed "s/example.com/$DOMAIN/g" manifests/3a/gateway-tcp.yaml | kubectl apply -f -

kubectl apply -f manifests/3a/tcproute.yaml

curl -sS "http://$TRAEFIK_IP:9000/" | grep '^Hostname'
# output
Hostname: echo-7698b9cd96-s6bfs
```

The Gateway accepts the `TCPRoute`, and Traefik serves raw TCP traffic on port 9000.

Model 2: Traefik Hub (API Management, AI Gateway and MCP Gateway)

In Model 2, Traefik Hub installs with the same Helm chart and the same release as Traefik Proxy. When `hub.token` contains the name of the license Secret, the chart uses the `ghcr.io/traefik/traefik-hub` image. The chart selects the Hub version that it supports, so you do not need an `image.*` override. The routes from Model 1 continue to work. Figure 7 shows the components that this model deploys.

The full procedure for Model 2 is Traefik/3B_TRAEFIK_HUB.md.

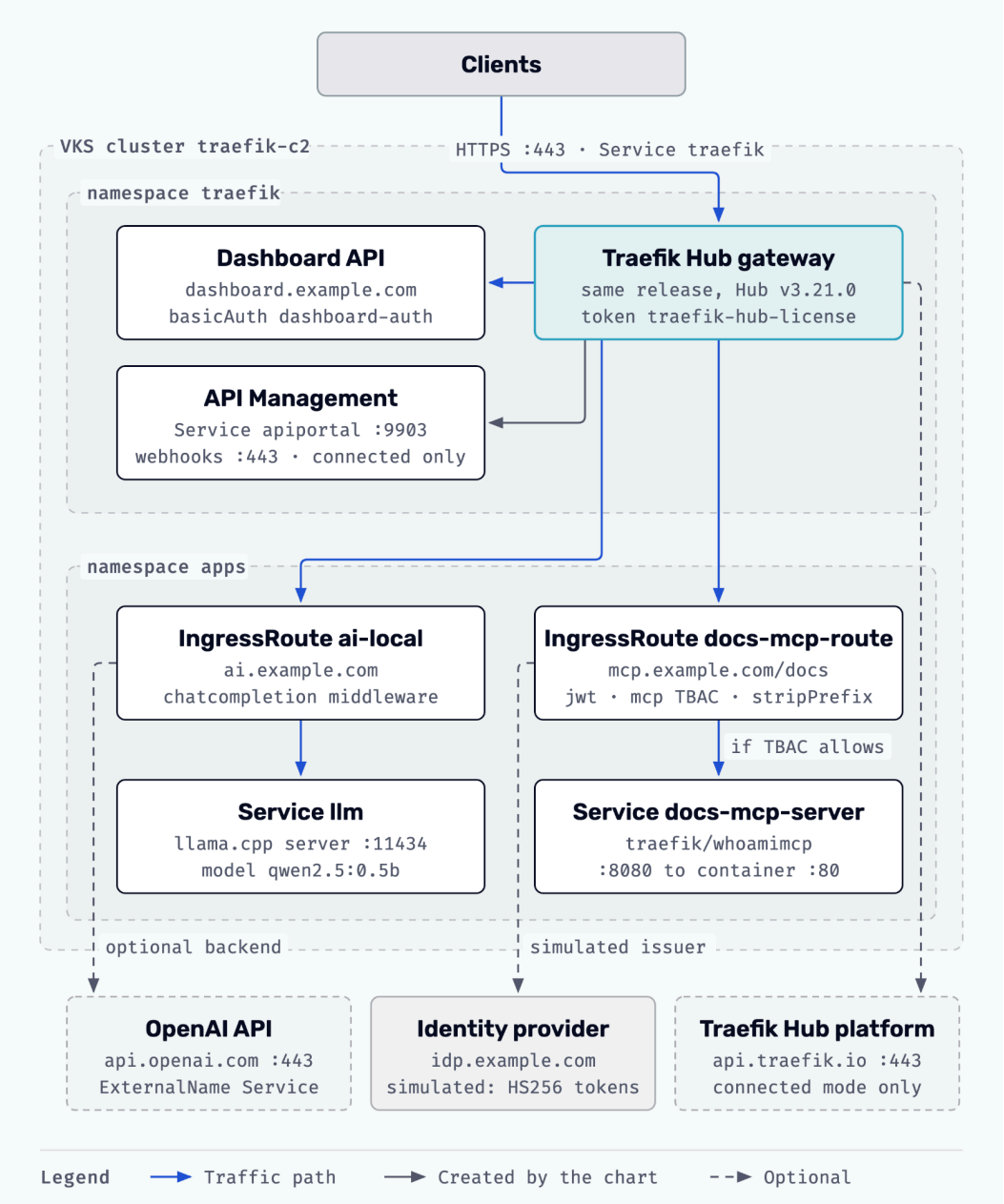


Figure 7: Components that Model 2 deploys

Prerequisites

- Traefik Proxy installed as in Model 1. The steps use its Helm release, the `traefik` namespace, the Traefik/manifests/ folder and the variable TRAEFIK_IP.

- A Traefik Hub license token. The Traefik Hub Online Dashboard issues the token when you create the gateway. For the offline variant, enable the "Offline" toggle at creation. For API Management, enable the "API management features" toggle.
- Connectivity: outbound HTTPS to `api.traefik.io` (TCP 443) in connected mode. Offline Mode requires no connection to the Traefik Hub platform. Egress to configured backends and infrastructure services still depends on the deployment. A fully air-gapped installation requires locally available images, charts and model artifacts.
- The Kubernetes CRD provider (a chart default). The Hub middlewares are `Middleware` custom resources.

Important. License features. Enable `hub.apimanagement` when the token includes API Management. This validation uses an offline token with the api-management and multi-cluster features.

Installation

Step 1: Set the environment variables and enter the secrets.

The secrets are typed at the prompt: they never reach the command line or the shell history; the dashboard password is kept only in $WORK/dashboard.netrc (mode 0600) for the later curl calls. The step also reads the values of Model 1: the LoadBalancer address and the CA of the gateway.

```shell
# bash
export DOMAIN="<your_domain>"; export CHART_VERSION="41.6.1"

export WORK="$HOME/traefik-vks-work"; mkdir -p "$WORK"; chmod 700 "$WORK"

TRAEFIK_IP=$(kubectl -n traefik get svc traefik \
  -o jsonpath='{.status.loadBalancer.ingress[0].ip}')

kubectl -n traefik get secret traefik-tls -o jsonpath='{.data.ca\.crt}' \
  | base64 -d > "$WORK/gateway-ca.crt"

read -rsp 'Traefik Hub license token: ' HUB_TOKEN; echo

read -rsp 'Dashboard password for admin: ' DASHBOARD_PASSWORD; echo
```

Step 2: Store the license token in a Secret.

The Secret needs the key token; the chart mounts it under hub.tokenMountPath (default /etc/secrets).

```shell
# bash
printf '%s' "$HUB_TOKEN" \
  | kubectl -n traefik create secret generic traefik-hub-license \
      --from-file=token=/dev/stdin --dry-run=client -o yaml \
  | kubectl apply --server-side -f -

unset HUB_TOKEN
```

Step 3: Apply the chart CRDs.

Helm does not upgrade CRDs after the first installation. Apply the CRDs from the chart, pinned to the chart version.

```shell
# bash
helm show crds traefik/traefik --version 41.6.1 \
  | kubectl apply --server-side --force-conflicts -f -
```

The output lists 25 CRDs as `serverside-applied`, with `uplinks.hub.traefik.io` among them.

Step 4: Upgrade the release to Traefik Hub.

The upgrade uses the same release name and namespace, with no separate operator.

```shell
# bash
helm upgrade traefik traefik/traefik -n traefik --version 41.6.1 --reuse-values \
  -f manifests/3b/values-hub.yaml   # license Secret, offline: true, API/AI/MCP gateways
```

With `hub.apimanagement.enabled=true`, the chart adds two `MutatingWebhookConfiguration` objects (`hub-acp-traefik-traefik`, `hub-api-traefik-traefik`).
It also adds the webhook Secret `hub-agent-cert` and the Services `admission` (443) and `apiportal` (9903).
With `hub.offline=true`, the chart does not create the webhooks, `hub-agent-cert` or the `admission` Service.
The in-cluster AI and MCP examples use `ClusterIP` Services. The `allowExternalNameServices` option is needed for the external variants (a cloud AI provider, the public DeepWiki MCP server) that use `ExternalName` Services.

Step 5: Expose the dashboard API with basic authentication.

The chart does not serve /api by default. openssl reads the password on stdin, and only its hash is stored, in the Secret dashboard-auth. This step applies the dashboard-auth middleware (manifests/3b/dashboard-auth-middleware.yaml) and the values file manifests/3b/values-dashboard.yaml, which enables the dashboard IngressRoute on websecure behind that middleware; the file holds the example host, and the --set override puts the dashboard on your domain.

```shell
# bash
HASH=$(printf '%s\n' "$DASHBOARD_PASSWORD" | openssl passwd -apr1 -stdin)

printf 'users=admin:%s\n' "$HASH" \
  | kubectl -n traefik create secret generic dashboard-auth --from-env-file=/dev/stdin \
      --dry-run=client -o yaml \
  | kubectl apply --server-side -f -

kubectl apply -f manifests/3b/dashboard-auth-middleware.yaml

RULE="Host(\`dashboard.$DOMAIN\`)"
RULE="$RULE && (PathPrefix(\`/api\`) || PathPrefix(\`/dashboard\`))"

helm upgrade traefik traefik/traefik -n traefik --version 41.6.1 --reuse-values \
  -f manifests/3b/values-dashboard.yaml --set "ingressRoute.dashboard.matchRule=$RULE"
```

Step 6: Verify the image and the startup log.

Start leading appears when the new pod takes the leader lease, up to about 90 seconds after the rollout; the guide waits for the previous pod to terminate, then for that line.

```shell
# bash
kubectl -n traefik get deploy traefik \
  -o jsonpath='{.spec.template.spec.containers[0].image}'
# output
ghcr.io/traefik/traefik-hub:v3.21.0

# bash
kubectl -n traefik logs deploy/traefik | grep -E 'hub\.Provider|Start leading|ERR'
# output
INF Starting provider *hub.Provider
INF Start leading
```

The chart selects the Hub image, and the gateway starts with no error. In the reference environment (v3.21.0, offline token), the startup log contains no license line.

The dashboard API reports `"status": "enabled"` for each applied Hub middleware at `GET /api/http/middlewares/<namespace>-<name>@kubernetescrd`. Without `hub.mcpgateway` or `hub.aigateway`, a route that references the matching middleware fails. With the flags set but no token, the gateway does not start.

If the --hub.offline value conflicts with the token, the gateway exits with offline mode configuration mismatch.

Verification: AI Gateway

The `chat-completion` middleware changes an existing route into a governed, OpenAI-compatible AI endpoint. It validates the request against the chat-completion schema, enforces the configured model and parameters, and adds the provider credential. The IngressRoute service selects the upstream, so the same middleware can be in front of a cloud provider or an in-cluster model (Figure 8).

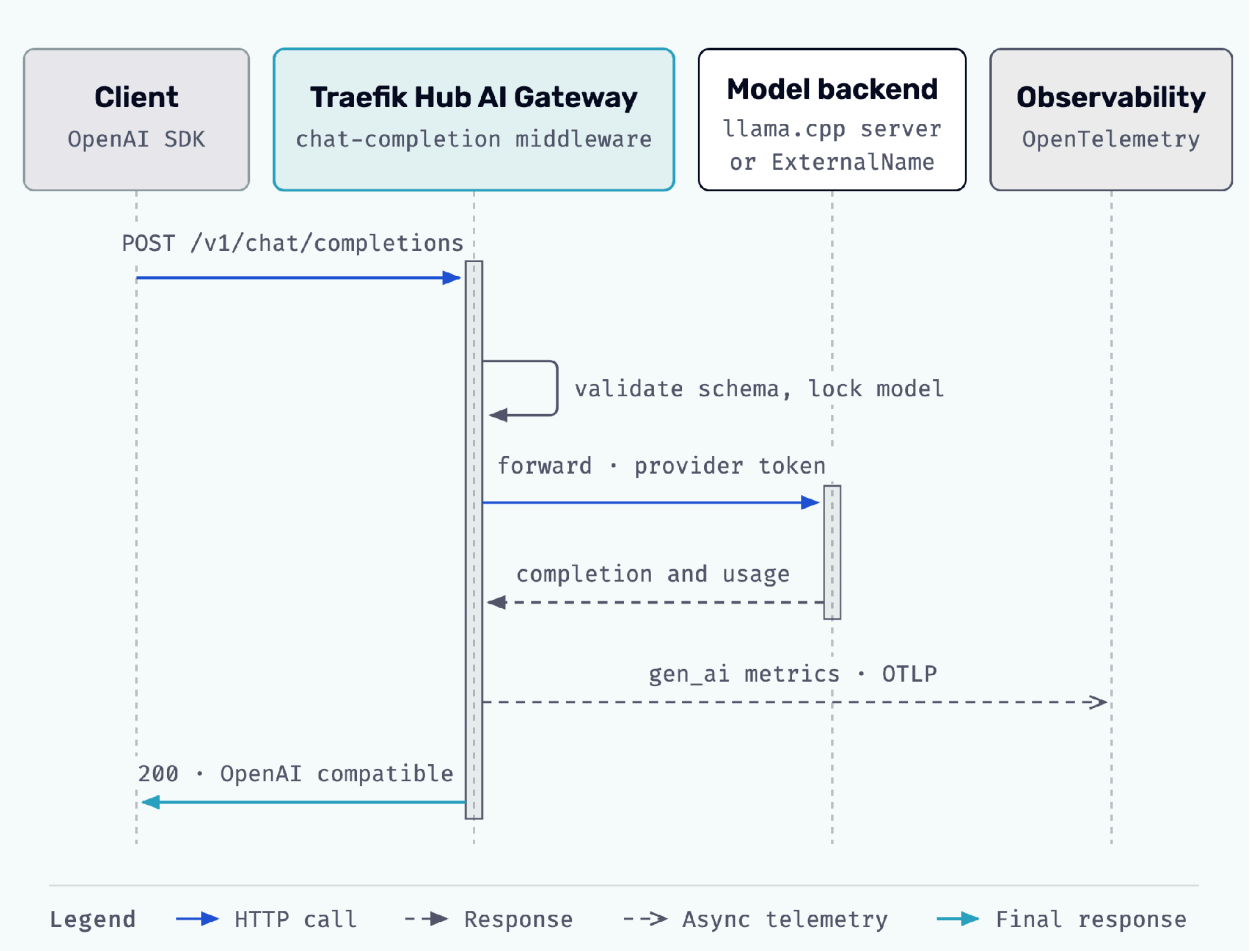


Figure 8: Chat-completion request flow through the AI Gateway

Requirements:

- In-cluster model: an OpenAI-compatible server behind a ClusterIP Service, with no API key. This option fits Offline Mode.
- Cloud provider: an `ExternalName` Service to the provider host and egress on TCP 443. Put the provider API key in a Secret in the Middleware namespace.
- The client calls the OpenAI path `/v1/chat/completions`, and the gateway does not change the path. The gateway rejects a body above `hub.aigateway.maxRequestBodySize` (default 1 MiB) with 413 and a chunked request with 400.

The validated backend is a llama.cpp server with the Qwen2.5 0.5B instruct model (about 0.5 GB). A lab observation outside the repository procedure: the Ollama image (3.8 GB) caused DiskPressure on a 19.3 GiB worker disk, which is why the validation uses llama.cpp. The --alias flag makes the server report the model name that the gateway enforces.

Step 1: Deploy the model server.

This step creates the namespace apps and applies manifests/3b/llama-cpp.yaml, the llama.cpp server. At startup, the server downloads the model from Hugging Face, so the worker nodes need egress on TCP 443 for this step.

```
Shell
# bash
kubectl create namespace apps --dry-run=client -o yaml | kubectl apply -f -
kubectl label namespace apps --overwrite pod-security.kubernetes.io/enforce=baseline
kubectl apply -f manifests/3b/llama-cpp.yaml
kubectl -n apps rollout status deployment/llm --timeout=600s
```

Step 2: Attach the `chat-completion` middleware to the AI route.

The middleware locks the model and allows parameter overrides. This step applies manifests/3b/ai-chatcompletion.yaml (the middleware) and manifests/3b/ai-route.yaml (the IngressRoute ai-local) on ai.example.com.

```
Shell
# bash
kubectl apply -f manifests/3b/ai-chatcompletion.yaml
sed "s/example.com/$DOMAIN/g" manifests/3b/ai-route.yaml | kubectl apply -f -
```

For an external OpenAI-compatible provider (documented; this paper does not validate it), replace the ClusterIP Service with an `ExternalName` Service to `api.openai.com`. The middleware adds `token: urn:k8s:secret:ai-keys:openai-token`, and the route service uses `port: 443, scheme: https, passHostHeader: false`.

Step 3: Send a request with a different model name and no API key.

The gateway replaces the model and adds the default parameters.

```
Shell
# bash
curl -s --cacert "$WORK/gateway-ca.crt" --resolve "ai.$DOMAIN:443:$TRAEFIK_IP" \
  "https://ai.$DOMAIN/v1/chat/completions" -H 'Content-Type: application/json' \
  -d '{"model":"some-other-model","messages":[{"role":"user","content":"Say ping"}]}' \
  | jq -c '{model, usage}'
# output
{"model":"qwen2.5:0.5b","usage":{"completion_tokens":33,"prompt_tokens":31,...}}
```

The gateway replaces the model of the request with the locked model and adds the default parameters; the token counts vary from run to run.

Step 4: Confirm the request-size limit.

```shell
# bash
# a 2 MB request body
curl -s -o /dev/null -w '%{http_code}\n' --cacert "$WORK/gateway-ca.crt" \
  --resolve "ai.$DOMAIN:443:$TRAEFIK_IP" "https://ai.$DOMAIN/v1/chat/completions" \
  -H 'Content-Type: application/json' --data-binary @"$WORK/big.json"
# output
413
```

The gateway rejects the 2 MB body before the body reaches the model.

The `ai-rate-limit` and `ai-quota` middlewares count tokens per consumer and keep their state in Redis. GenAI metrics (`gen_ai.client.token.usage`, `gen_ai.client.operation.duration`) are available in Traefik Hub v3.21.0. They require `observability.metrics.level: detailed` on the middleware and OTLP metrics on the gateway.

Verification: MCP Gateway

The MCP Gateway is a chain of two middlewares in front of an MCP server. The `jwt` middleware authenticates the caller and places the token claims in the request context. The `mcp` middleware acts as an OAuth 2.1 resource server: it publishes protected-resource metadata and intercepts the JSON-RPC calls. It evaluates TBAC policies against the MCP method, the tool name, the tool arguments and the JWT claims (Figure 9). The `mcp` middleware has no token validation of its own, so the order `jwt` then `mcp` is mandatory.

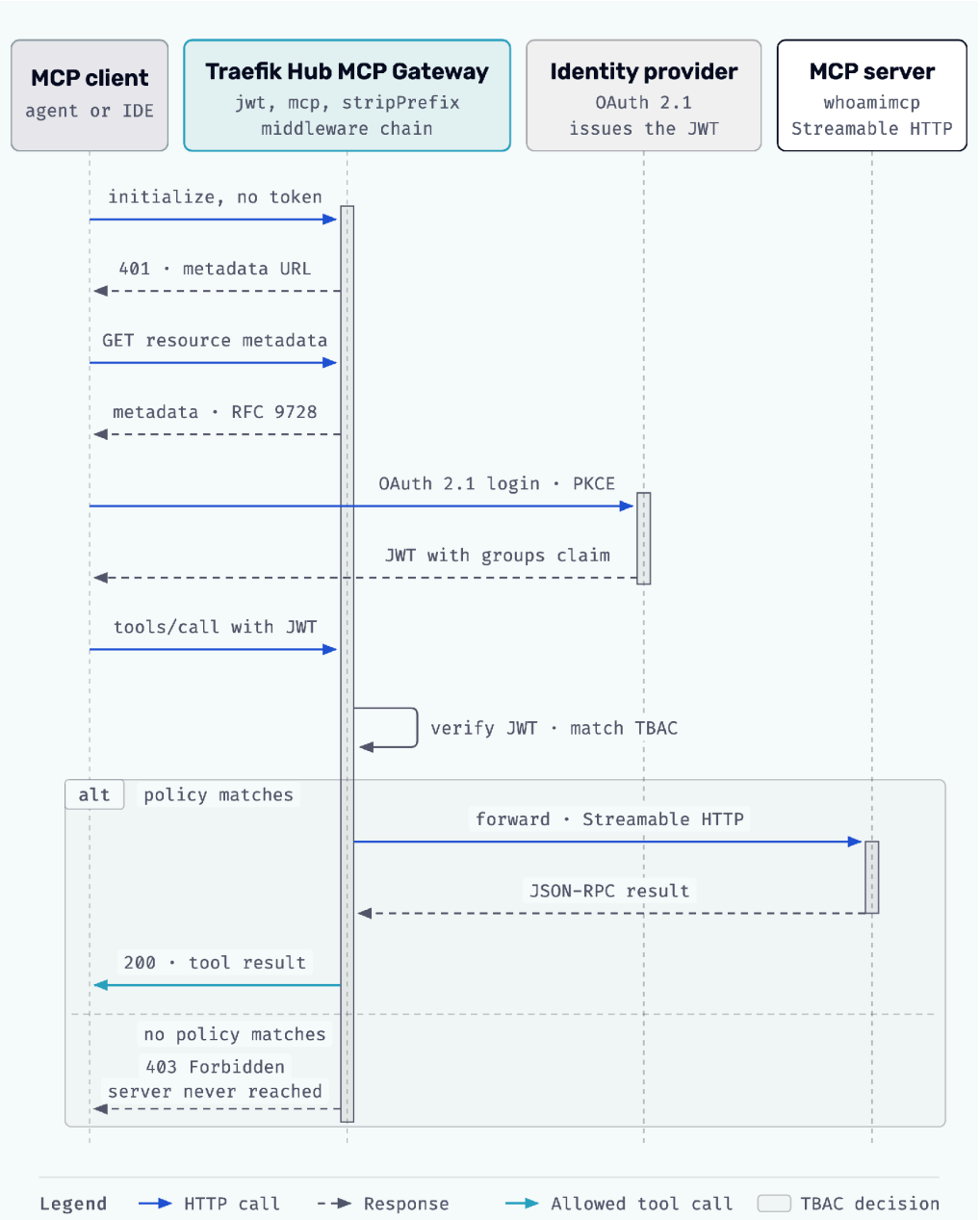


Figure 9: JWT authentication and TBAC in front of an MCP server

Table 5: MCP Gateway validation components

Component	Role	Lab choice
MCP server	The governed tool provider (Streamable HTTP transport).	`traefik/whoamimcp`: port 80, server name `greeter_s1`, one tool `greet`
Identity provider	Issues JWT access tokens with a `groups` claim.	Simulated: HS256 tokens signed with a shared key (no IdP in the lab) Production: `jwksUrl` or `trustedIssuers` (the gateway fetches the JWKS)

Component	Role	Lab choice
`jwt` middleware	Validates the token, returns 401 with `WWW-Authenticate` and forwards claims as headers.	`signingSecret` from a Secret URN
`mcp` middleware	Publishes the resource metadata and enforces the TBAC policies.	`tools/list` for all; `tools/call` on `greet` for the `admin` group
MCP client	Any Streamable HTTP client.	curl; MCP Inspector with a pasted token

Requirements:

- A TLS route. The metadata endpoint answers over HTTPS; over plain HTTP it returns 404 (lab observation). Use an `https://` value in `resourceMetadata.resource`.
- The `wwwAuthenticate` value of the `jwt` middleware must equal the metadata URL that the `mcp` middleware generates: `https://<host>/.well-known/oauth-protected-resource<path>`.
- With more than one MCP server replica, use the Highest Random Weight strategy so that an MCP session stays on one instance.

Step 1: Deploy the test MCP server.

This step applies manifests/3b/whoamimcp.yaml, the whoamimcp server and its Service. The server answers initialize with the name greeter_s1 and lists one tool, greet.

```shell
# bash
kubectl apply -f manifests/3b/whoamimcp.yaml
```

Step 2: Create the signing secret and the middlewares.

This step generates the signing key, stores it in the Secret mcp-jwt-secret and applies manifests/3b/mcp-middlewares.yaml (the jwt middleware, the mcp middleware and a stripPrefix middleware) and manifests/3b/mcp-route.yaml (the IngressRoute that routes mcp.<your domain>/docs through the three middlewares, in this order). The jwt middleware validates the token and forwards the claims sub and groups as request headers. This validation simulates the identity provider: the gateway validates HS256 tokens signed with a shared key, and https://idp.example.com stands for the address of your IdP. In production, set trustedIssuers or jwksUrl to the enterprise IdP, and list the same IdP in authorizationServers.

```shell
# bash
JWT_SIGNING_SECRET=$(openssl rand -base64 32)

printf '%s' "$JWT_SIGNING_SECRET" \
  | kubectl -n apps create secret generic mcp-jwt-secret \
      --from-file=signingSecret=/dev/stdin --dry-run=client -o yaml \
  | kubectl apply --server-side -f -

sed "s/example.com/$DOMAIN/g" manifests/3b/mcp-middlewares.yaml | kubectl apply -f -

sed "s/example.com/$DOMAIN/g" manifests/3b/mcp-route.yaml | kubectl apply -f -
```

The `mcp` middleware holds the TBAC policies. The policy field is `match`. The middleware passes `initialize` and `notifications/initialized` without a rule, so the handshake does not need a policy.

Step 3: Create two test tokens, one for each group.

The commands sign the tokens with openssl and the key `JWT_SIGNING_SECRET` from Step 2. You can use a different HS256 signer.

```shell
# bash
# HS256 test tokens; b64url, sign_jwt and mcp are defined in Traefik/3B_TRAEFIK_HUB.md
EXP=$(( $(date +%s) + 3600 ))

DEV_TOKEN=$(sign_jwt '{"sub":"dev-user","groups":["developer"],"exp":'"$EXP"'}')

ADMIN_TOKEN=$(sign_jwt '{"sub":"admin-user","groups":["admin"],"exp":'"$EXP"'}')
```

Step 4: Test the discovery and the authentication.

The `--resolve` option sends each call to the LoadBalancer address `TRAEFIK_IP`.

```shell
# bash
curl -s --cacert "$WORK/gateway-ca.crt" --resolve "mcp.$DOMAIN:443:$TRAEFIK_IP" \
  "https://mcp.$DOMAIN/.well-known/oauth-protected-resource/docs"
# output
{"resource":"https://mcp.example.com/docs","authorization_servers":
["https://idp.example.com"],"bearer_methods_supported":["header"],...}

# bash
INIT='{"jsonrpc":"2.0","id":1,"method":"initialize",'
INIT+='"params":{"protocolVersion":"2025-06-18","capabilities":{},'
INIT+='"clientInfo":{"name":"curl","version":"0"}}}'

curl -s -i --cacert "$WORK/gateway-ca.crt" --resolve "mcp.$DOMAIN:443:$TRAEFIK_IP" \
  -H 'Content-Type: application/json' -d "$INIT" "https://mcp.$DOMAIN/docs/" \
  | grep -iE '^HTTP|www-authenticate'
# output
HTTP/2 401
www-authenticate: https://mcp.example.com/.well-known/oauth-protected-resource/docs
```

The call without a token receives a 401 response that points to the metadata document.

Step 5: Run the handshake and list the tools with the developer token.

The handshake is initialize, then the notification notifications/initialized. The later requests carry the session ID and the header MCP-Protocol-Version.

```shell
# bash
SID=$(mcp "$DEV_TOKEN" -i -d "$INIT" | tr -d '\r' \
  | awk 'tolower($1)=="mcp-session-id:"{print $2}')

mcp "$DEV_TOKEN" -o /dev/null -w '%{http_code}\n' \
  -H "Mcp-Session-Id: $SID" -H 'MCP-Protocol-Version: 2025-06-18' \
  -d '{"jsonrpc":"2.0","method":"notifications/initialized"}'
# output
202

# bash
mcp "$DEV_TOKEN" -H "Mcp-Session-Id: $SID" -H 'MCP-Protocol-Version: 2025-06-18' \
  -d '{"jsonrpc":"2.0","id":2,"method":"tools/list"}' | grep -o '"name":"greet"'
# output
"name":"greet"
```

The policy allows `tools/list` for each authenticated caller.

Step 6: Call the tool with both tokens.

The policy denies the developer call before it reaches the server and allows the admin call.

```shell
# bash
CALL='{"jsonrpc":"2.0","id":3,"method":"tools/call",'
CALL+='"params":{"name":"greet","arguments":{"name":"admin"}}}'

mcp "$DEV_TOKEN" -i -H "Mcp-Session-Id: $SID" -H 'MCP-Protocol-Version: 2025-06-18' \
  -d "$CALL" | grep -E '^HTTP|"error"'
# output
HTTP/2 403
{"jsonrpc":"2.0","id":3,"error":{"code":-32003,"message":"Forbidden"}}

# bash
mcp "$ADMIN_TOKEN" -H "Mcp-Session-Id: $SID" -H 'MCP-Protocol-Version: 2025-06-18' \
  -d "$CALL" | grep -o 'Hello, admin[^"]*'
# output
Hello, admin from docs-mcp-server-665f9b469b-s2ld4 :80!
```

A lab observation outside the repository procedure: MCP Inspector, connected to https://mcp.example.com/docs/ with a Bearer header, showed the same outcomes.

Also a lab observation outside the repository procedure: the same policy model was tested against the public MCP server DeepWiki (mcp.deepwiki.com) through an ExternalName Service, with the same result. An ExternalName Service cannot replace a ClusterIP Service in place: delete the old Service first, or use a new name.

You can use the same jwt middleware on an API route or an AI route. Only the mcp middleware is specific to MCP. In Traefik Hub v3.21.0, the AI Gateway guards (Content Guard, LLM Guard, Parallel LLM Guard) can follow the mcp middleware in the chain. The guards inspect the tool arguments and the tool results.

Verification: API Management Installation

The installation in Step 4 enables API Management. This check covers the components that the chart installs; it does not publish an API. Check the objects that the chart creates. With hub.offline=true, only the apiportal Service is present.

```shell
# bash
kubectl get mutatingwebhookconfigurations -o name | grep hub- \
  || echo "no hub webhooks (offline)"
# output
no hub webhooks (offline)

# bash
kubectl -n traefik get svc apiportal
# output
NAME        TYPE        CLUSTER-IP     EXTERNAL-IP   PORT(S)    AGE
apiportal   ClusterIP   10.102.91.71   <none>        9903/TCP   2m
```

The gateway pod is Ready with `--hub.apimanagement`, so the token allows the feature. To publish an API, apply the `API`, `APIVersion`, `APIPlan` and `APIPortal` resources that the Traefik Hub API Management documentation describes.

Model 3: Traefik Across VMs and Containers

Model 3 extends the Model 2 gateway to virtual machines, so a workload can move between control planes while its client-facing address stays the same. It uses the multi-cluster provider (Early Access) and the parent and child roles from "Workloads Outside the Cluster" (Figure 4). The parent needs network access to the uplink entry point of each child, and no vCenter or Supervisor credential. Figure 10 shows the components that this model deploys.

Important. The full procedure for Model 3, including the creation of the child and application VMs, is documented in the GitHub repository in Traefik/3C_TRAEFIK_VMS_MULTICLUSTER.md.
The child gateways run an Early Access build of Traefik Hub v3.21 with the vsphere and vmoperator providers. To evaluate Model 3, contact Traefik Labs (https://info.traefik.io/en/request-demo): we provide the build and support the installation.

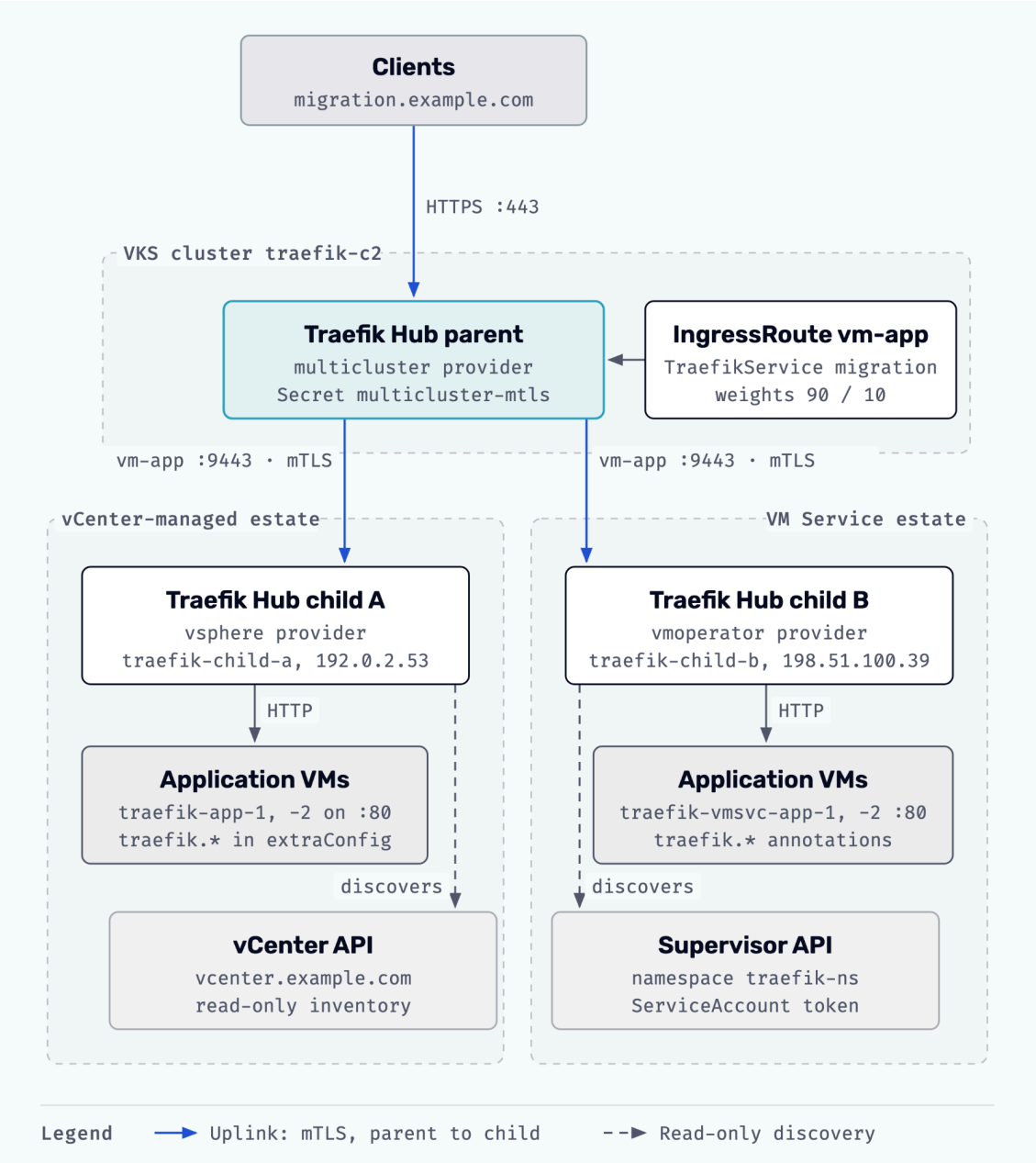


Figure 10: Components that Model 3 deploys

Prerequisites

- One Linux VM in each estate for the child, with Docker (Compose v2) and SSH access from the administrator workstation. The reference environment uses Ubuntu 24.04: child A in the resource pool traefik-rp, child B as a VirtualMachine in traefik-ns. Bootstrap child A with cloud-init through guestinfo (setting the template's vApp properties needs VApp privileges that delegated roles often lack) and child B with the cloud-init Secret of its VirtualMachine. VM Service VMs sit on the namespace VPC, which the workstation usually cannot reach: publish SSH with a VirtualMachineService of type LoadBalancer, or use a jump host inside the VPC.
- Network: the VKS cluster reaches each child on the uplink port (TCP 9443). Each child reaches its application VMs on their service ports. The application VMs serve HTTP on port 80; the examples use the `whoami` server.

- One license token per child with the `multi-cluster` feature. The reference environment uses offline tokens.
- Child A: a vCenter account that reads the VMs of the application folder. Recommended: a dedicated user with the built-in Read-only role on that folder (and on the datacenter, without propagation, when discovery is scoped to it), granted no wider, because the provider reads all Configuration Parameters of the VMs it can see; creating it needs vCenter SSO administrator rights. The validation ran the provider with the operator's own account, already limited to the application folder; the dedicated read-only user was not validated in this lab.
- Child B: permission to create a ServiceAccount, a Role, a RoleBinding and a Secret in the vSphere Namespace traefik-ns. Step 7 creates them; the token goes to a file the child reads.
- Certificates for mutual TLS: one server certificate per child and one client certificate for the parent, from a private CA. Traefik/3C_TRAEFIK_VMS_MULTICLUSTER.md creates them with openssl.
- Firewall: allow TCP 9443 to each child VM from the VKS worker nodes only. A caller that reaches the uplink port can call the advertised routes.

Installation

The routing contract

Both providers read the same routing labels as Docker does: traefik.<key>=<value> pairs that opt the machine in and describe its services. The vsphere provider reads them from the Configuration Parameters of each VM (config.extraConfig; Edit Settings > VM Options > Advanced > Configuration Parameters in the vSphere Client), one parameter per label. They can be set with the vSphere Client, govc vm.change -e, the Terraform extra_config argument or Ansible, without a parser. The vmoperator provider reads them from annotations on the VirtualMachine object, one annotation per label. The keys that this paper validates are traefik.enable, traefik.tags and the loadbalancer setting server.port; the provider supports the other loadbalancer settings, for example strategy and sticky.cookie.name, in the same form.

```shell
traefik.enable=true
traefik.tags=vm
traefik.http.services.vmapp.loadbalancer.server.port=80
```

The provider merges the machines that carry the same labels into one load-balanced service (vmapp@vsphere or vmapp@vmoperator). The file provider of the child declares the router and the Uplink that reference this service. The constraints expression of the provider filters on the traefik.* labels, for example traefik.tags. The VM Notes field is not read: it stays free for people and for other tools, and the vsphere provider does not read vSphere tags or custom attributes either.

The steps below keep the numbers of Traefik/3C_TRAEFIK_VMS_MULTICLUSTER.md. Its step 1 sets the variables and reads the secrets at a prompt (the license token for the children, the vCenter user and password, the dashboard password of Model 2), step 4 writes the SSH configuration file $WORK/ssh_config (one Host entry per child with its address, your key and a known_hosts file under $WORK; the later steps use it with ssh -F and scp -F) and waits for cloud-init on both children, and step 5 creates the

certificates for mutual TLS in $WORK: a private CA, a server certificate for each child with its IP address in the subjectAltName, and a client certificate for the parent. They are not repeated here.

Child A: vCenter-managed virtual machines

The vsphere provider opens one read-only vCenter session. Every refreshSeconds (15 by default) it reads, in one property-collector call, five properties per machine: name, power state, guest information, UUID and Configuration Parameters, and keeps only the traefik.* parameters. With exposedByDefault: false, a powered-on machine participates when it carries traefik.enable=true. The backend address is the guest address that VMware Tools reports (ipMode: private, public or ipv6). The provider ignores a machine without VMware Tools. The child verifies the vCenter certificate with the vCenter root CAs (the VMCA and, where vCenter uses one, the enterprise root CA), mounted into the container and named by SSL_CERT_FILE next to the system roots, so no TLS verification is skipped.

Step 2: Create child A and the application VMs in vCenter.

The guide clones the child and the two application VMs from the Ubuntu template with govc. cloud-init comes through guestinfo (base64, set before the first power-on) and carries no secret, only your SSH public key. The application VMs run whoami and get the routing labels as Configuration Parameters.

```shell
# bash
for n in 1 2; do
  govc vm.change -vm "$VM_FOLDER/traefik-app-$n" \
    -e traefik.enable=true -e traefik.tags=vm \
    -e traefik.http.services.vmapp.loadbalancer.server.port=80
done

govc vm.ip -wait 5m "$VM_FOLDER/traefik-child-a" \
  "$VM_FOLDER/traefik-app-1" "$VM_FOLDER/traefik-app-2"
# output
192.0.2.53
192.0.2.58
192.0.2.59

# bash
CHILD_A_IP=$(govc vm.ip "$VM_FOLDER/traefik-child-a")

govc vm.info -e "$VM_FOLDER/traefik-app-1" | grep 'traefik\.'
# output
    traefik.enable:                                          true
    traefik.http.services.vmapp.loadbalancer.server.port:  80
    traefik.tags:                                            vm
```

Step 6: Configure and run child A.

manifests/3c/child-a/compose.yaml holds the service definition and, as an embedded configuration, the vsphere provider, offline: true and the local API on the loopback address 127.0.0.1:8080; manifests/3c/child-a/dynamic.yaml the Uplink vm-app, its router to vmapp@vsphere and the strict-mtls TLS options. The license token and the vCenter password go to 0600 files that the step copies to the VM and deletes from the workstation; Compose reads the token as a secret and fills the password into the configuration: both are typed on the workstation and never appear on a command line or in the repository. The vCenter root CAs are mounted into the container and named by SSL_CERT_FILE, next to the system roots.

```
Shell
providers:
  multicluster: {}                  # Uplink handling on the child
  vsphere:                          # discover the VMs through vCenter (read-only)
    endpoint: ${VCENTER}
    username: ${VCENTER_USER}
    password: '${VCENTER_PASSWORD}'
    exposedByDefault: false         # only VMs with traefik.enable=true
    ipMode: private                 # backend = guest IP reported by VMware Tools
    refreshSeconds: 15
    constraints: Label(`traefik.tags`,`vm`)
```

The step assembles the project folder on the workstation, copies it to the VM and starts it:

```
Shell
# bash
# a re-run replaces the project folder
ssh -n -F "$WORK/ssh_config" child-a 'rm -rf child-a'

printf '%s' "$HUB_TOKEN" > "$WORK/child-a/hub-token"

printf "VCENTER='%s'\nVCENTER_USER='%s'\nVCENTER_PASSWORD='%s'\nHUB_CHILD_IMAGE='%s'\n" \
  "$VCENTER" "$VCENTER_USER" "$VCENTER_PASSWORD" "$HUB_CHILD_IMAGE" \
  > "$WORK/child-a/.env"

scp -rpq -F "$WORK/ssh_config" "$WORK/child-a" child-a:

rm "$WORK/child-a/hub-token" "$WORK/child-a/.env"     # the VM holds the only copy

ssh -n -F "$WORK/ssh_config" child-a \
  'sudo docker compose --project-directory child-a up -d --force-recreate'

until ssh -n -F "$WORK/ssh_config" child-a \
  'curl -sf http://127.0.0.1:8080/api/http/services/vmapp@vsphere |
   jq -e ".loadBalancer.servers | length == 2"' >/dev/null 2>&1; do sleep 5; done

ssh -n -F "$WORK/ssh_config" child-a \
  'curl -s http://127.0.0.1:8080/api/http/services/vmapp@vsphere |
   jq -c "{status, servers: .loadBalancer.servers}"'
# output
{"status":"enabled","servers":[{"url":"http://192.0.2.58:80"},
{"url":"http://192.0.2.59:80"}]}
```

The service vmapp@vsphere is enabled, with one server for each powered-on VM that carries the labels.

Child B: stand-alone VMs through VM Service

The vmoperator provider lists the VirtualMachine objects of the configured namespaces over the Supervisor API. It uses a namespace-scoped ServiceAccount token read from a file, and verifies the Supervisor certificate with the vCenter root CAs: on VKS the Supervisor API endpoint is signed by the VMCA, not by the cluster CA that comes with the token. The backend address is status.network.primaryIP4, so this path does not need VMware Tools. The provider uses the objects with status.powerState: PoweredOn, and Ready when that condition is present. Child B can itself be a VirtualMachine in the same namespace.

Step 3: Create child B and the application VMs with VM Service.

manifests/3c/child-b/child-b-vm.yaml creates the child VM, its cloud-init Secret and a LoadBalancer on port 22 for SSH from the workstation; manifests/3c/child-b/app-vms.yaml creates two whoami VMs with the routing labels as annotations. The sed line adapts the VM class, the image and the storage class of the example environment and inserts your SSH public key; -n selects your vSphere Namespace.

```shell
# bash
for f in manifests/3c/child-b/child-b-vm.yaml manifests/3c/child-b/app-vms.yaml; do
  sed -e "s/best-effort-medium/$VM_CLASS/" \
    -e "s/vmi-9f8e7d6c5b4a39281/$VMSVC_IMAGE/" \
    -e "s/vsan-esa-default-policy-raid5/$STORAGE_CLASS/" \
    -e "s|REPLACE_ME_SSH_PUBLIC_KEY|$SSH_PUBLIC_KEY|" "$f" \
    | kubectl --context "$SUPERVISOR_CONTEXT" -n "$VSPHERE_NAMESPACE" apply -f -
done

kubectl --context "$SUPERVISOR_CONTEXT" -n "$VSPHERE_NAMESPACE" \
  wait vm/traefik-child-b --for=jsonpath='{.status.network.primaryIP4}' --timeout=10m

CHILD_B_IP=$(kubectl --context "$SUPERVISOR_CONTEXT" -n "$VSPHERE_NAMESPACE" \
  get vm traefik-child-b -o jsonpath='{.status.network.primaryIP4}')

COLS='NAME:.metadata.name,POWER:.status.powerState,IP:.status.network.primaryIP4'

kubectl --context "$SUPERVISOR_CONTEXT" -n "$VSPHERE_NAMESPACE" \
  get vm -o custom-columns="$COLS"
# output
NAME                  POWER        IP
traefik-child-b       PoweredOn    198.51.100.39
traefik-vmsvc-app-1   PoweredOn    198.51.100.40
traefik-vmsvc-app-2   PoweredOn    198.51.100.41
```

The Supervisor serves several vmoperator.vmware.com versions and converts between them. The procedure reads v1alpha3 (apiVersion option), which every supported Supervisor serves.

Step 7: Configure and run child B.

manifests/3c/child-b/rbac.yaml gives the ServiceAccount traefik-vmoperator read access (get, list, watch) to the VirtualMachines of the namespace. Its token goes to a file the step copies to the child, and the child verifies the Supervisor certificate with the vCenter root CAs. The static configuration, embedded in manifests/3c/child-b/compose.yaml, uses the vmoperator provider (endpoint, tokenFile, tls.ca, namespaces) instead of the vsphere provider; the dynamic configuration is the one of child A with the service vmapp@vmoperator.

```shell
# bash
kubectl --context "$SUPERVISOR_CONTEXT" -n "$VSPHERE_NAMESPACE" \
  apply -f manifests/3c/child-b/rbac.yaml

kubectl --context "$SUPERVISOR_CONTEXT" -n "$VSPHERE_NAMESPACE" \
  auth can-i list virtualmachines.vmoperator.vmware.com \
  --as="system:serviceaccount:$VSPHERE_NAMESPACE:traefik-vmoperator"
# output
yes

# bash
kubectl --context "$SUPERVISOR_CONTEXT" -n "$VSPHERE_NAMESPACE" \
  wait secret/traefik-vmoperator-token --for=jsonpath='{.data.token}' --timeout=60s
# output
secret/traefik-vmoperator-token condition met

# bash
kubectl --context "$SUPERVISOR_CONTEXT" -n "$VSPHERE_NAMESPACE" \
  get secret traefik-vmoperator-token -o jsonpath='{.data.token}' \
  | base64 -d > "$WORK/child-b/vmoperator-token"

ssh -n -F "$WORK/ssh_config" child-b 'rm -rf child-b'

printf '%s' "$HUB_TOKEN" > "$WORK/child-b/hub-token"

printf "SUPERVISOR_IP='%s'\nVSPHERE_NAMESPACE='%s'\nHUB_CHILD_IMAGE='%s'\n" \
  "$SUPERVISOR_IP" "$VSPHERE_NAMESPACE" "$HUB_CHILD_IMAGE" > "$WORK/child-b/.env"

scp -rpq -F "$WORK/ssh_config" "$WORK/child-b" child-b:

rm "$WORK/child-b/hub-token" "$WORK/child-b/vmoperator-token" "$WORK/child-b/.env"

ssh -n -F "$WORK/ssh_config" child-b \
  'sudo docker compose --project-directory child-b up -d --force-recreate'

until ssh -n -F "$WORK/ssh_config" child-b \
  'curl -sf http://127.0.0.1:8080/api/http/services/vmapp@vmoperator |
   jq -e ".loadBalancer.servers | length == 2"' >/dev/null 2>&1; do sleep 5; done

ssh -n -F "$WORK/ssh_config" child-b \
  'curl -s http://127.0.0.1:8080/api/http/services/vmapp@vmoperator |
   jq -c "{status, servers: .loadBalancer.servers}"'
# output
{"status":"enabled","servers":[{"url":"http://198.51.100.40:80"},
{"url":"http://198.51.100.41:80"}]}
```

The router points to vmapp@vmoperator, and the Uplink name is vm-app. Both children advertise an Uplink named vm-app, so the parent sees one workload served by two estates.

Parent gateway on the VKS cluster

Step 8: Configure the parent gateway.

This step switches to the VKS cluster context. The Secret multicluster-mtls holds the CA and the client certificate and key of the parent, mounted on /certs. manifests/3c/values-multicluster.yaml declares the children estate-vcenter and estate-vmservice with mutual TLS; their addresses are passed with --set. The Helm values take integers for the poll settings, and the install configuration takes durations (5s). The IngressRoute sends migration.<your domain> to the weighted TraefikService migration: 90 to child A, 10 to child B.

```shell
# bash
kubectl --context "$CLUSTER_CONTEXT" -n traefik \
  create secret generic multicluster-mtls --from-file=ca.crt="$WORK/mtls-ca.crt" \
  --from-file=client.crt="$WORK/client.crt" --from-file=client.key="$WORK/client.key" \
  --dry-run=client -o yaml \
  | kubectl --context "$CLUSTER_CONTEXT" apply --server-side -f -

CHILDREN=hub.providers.multicluster.children

helm --kube-context "$CLUSTER_CONTEXT" upgrade traefik traefik/traefik -n traefik \
  --version "$CHART_VERSION" --reuse-values -f manifests/3c/values-multicluster.yaml \
  --set "$CHILDREN.estate-vcenter.address=https://$CHILD_A_IP:9443" \
  --set "$CHILDREN.estate-vmservice.address=https://$CHILD_B_IP:9443"

kubectl --context "$CLUSTER_CONTEXT" apply -f manifests/3c/migration-traefikservice.yaml

sed "s/example.com/$DOMAIN/g" manifests/3c/vm-app-ingressroute.yaml \
  | kubectl --context "$CLUSTER_CONTEXT" apply -f -
```

The parent creates vm-app@multicluster, weighted across the estates, and one service for each child.

Verification: Traffic Migration Between Control Planes

A cutover between estates is a weight change. Set the weights on the Uplink of each child, or override them on the parent with the weighted TraefikService migration (parent, Step 8).

Step 1: Check the uplink discovery and the mutual TLS.

The child requires the parent's client certificate and rejects the handshake without it.

```shell
# bash
curl -s --cacert "$WORK/mtls-ca.crt" \
  --cert "$WORK/client.crt" --key "$WORK/client.key" \
  "https://$CHILD_A_IP:9443/api/uplinks"
# output
[{"name":"vm-app","entryPoints":["multicluster"],"weight":1}]

# bash
curl -s --cacert "$WORK/mtls-ca.crt" "https://$CHILD_A_IP:9443/api/uplinks" \
  || echo "exit $?"
# output
exit 56        # tlsv13 alert certificate required
```

Step 2: List the services the parent generates.

The dashboard API of Model 2 answers with the admin password, passed to curl with $WORK/dashboard.netrc (written in step 1 of Traefik/3C_TRAEFIK_VMS_MULTICLUSTER.md), not on the command line.

```shell
# bash
curl -sS --fail --cacert "$WORK/gateway-ca.crt" \
  --resolve "dashboard.$DOMAIN:443:$TRAEFIK_IP" --netrc-file "$WORK/dashboard.netrc" \
  "https://dashboard.$DOMAIN/api/http/services" \
  | jq -r '.[] | select(.provider == "multicluster") | .name'
# output
vm-app-estate-vcenter@multicluster
vm-app-estate-vmservice@multicluster
vm-app@multicluster
```

Step 3: Confirm the split, then change the weights.

The whoami response shows the address of the child that forwards the request. The cutover is a patch of the two weights.

```shell
# bash
for _ in $(seq 20); do
  curl -sS --cacert "$WORK/gateway-ca.crt" \
    --resolve "migration.$DOMAIN:443:$TRAEFIK_IP" "https://migration.$DOMAIN/" \
    | sed -n 's/^RemoteAddr: \([^:]*\).*/\1/p'
done | sort | uniq -c
# output
     18 192.0.2.53          # child A (vCenter estate), weight 90
      2 198.51.100.39       # child B (VM Service estate), weight 10

# bash
PATCH='[{"op":"replace","path":"/spec/weighted/services/0/weight","value":0},'
PATCH+='{"op":"replace","path":"/spec/weighted/services/1/weight","value":100}]'

kubectl --context "$CLUSTER_CONTEXT" -n traefik \
  patch traefikservice migration --type=json -p="$PATCH"
# output
traefikservice.traefik.io/migration patched

# bash
# The guide first waits until the parent has loaded the new weights (seconds)
for _ in $(seq 20); do
  curl -sS --cacert "$WORK/gateway-ca.crt" \
    --resolve "migration.$DOMAIN:443:$TRAEFIK_IP" "https://migration.$DOMAIN/" \
    | sed -n 's/^RemoteAddr: \([^:]*\).*/\1/p'
done | sort | uniq -c
# output
     20 198.51.100.39       # after the cutover: weights 0 / 100
```

The split follows the weights, and the route, the certificates and the policies stay the same. A child cluster can add a container target to the same service with a Kubernetes Uplink that sets spec.exposeName: vm-app. Without exposeName, the Uplink name is <namespace>-<name>. The IngressRoute on that child needs the hub.traefik.io/router.uplinks annotation and no entryPoints or tls.

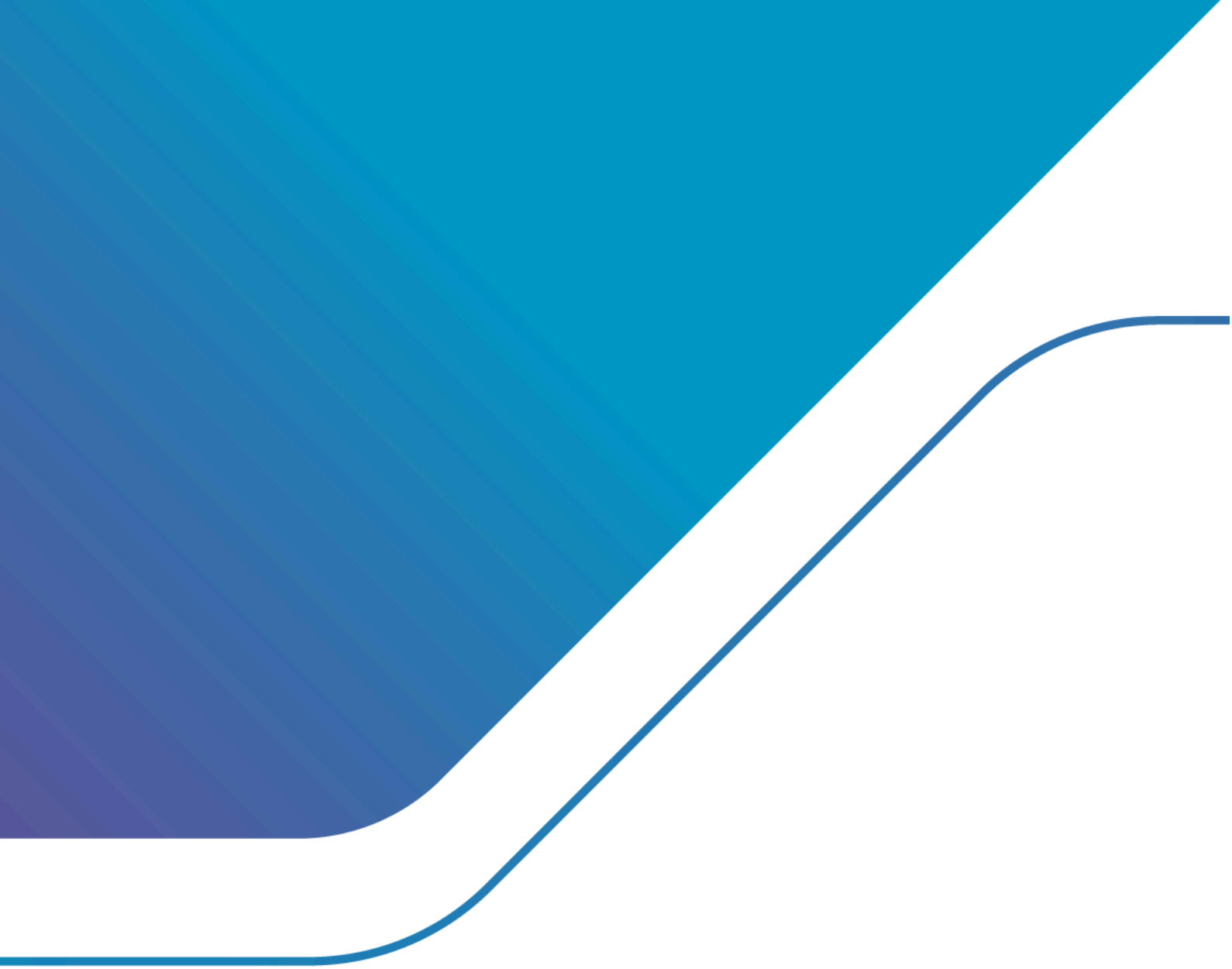
Conclusion

Platform teams standardizing on VCF should not need a separate gateway stack for each generation of workload. On VKS, Traefik puts one policy enforcement point in front of every service, model endpoint and MCP tool server, and keeps it in the same declarative form as the clusters the Supervisor manages. Routes, authentication, rate limits and tool-level access control for AI agents are Kubernetes resources in the Git repository that deploys the applications, so a policy change goes through the same review as a change to a Deployment. The controls that drift apart when every team builds its own stay consistent by design.

Adoption is incremental. A team starts with Traefik Proxy, which needs no license and no external service. A license token upgrades the same Helm release in place to Traefik Hub, adding API Management, the AI Gateway and the MCP Gateway, while the routes from the first installation keep working. For regulated and disconnected environments, Offline Mode runs the API, AI and MCP gateways with no connection to the Traefik Hub platform, and the FIPS 140-3 and hardened images install from the same chart.

The gateway also reaches past Kubernetes. Traefik Hub child gateways (Early Access) discover virtual machines through vCenter and through the Supervisor, so one gateway can front both the virtual machines and the containers in a VCF environment. Moving a workload from a vCenter-managed VM to a VM Service VM, or into a container, is a weight change on the parent gateway, and the client-facing route and its TLS certificate stay as they are. Modernization happens behind a stable front door, at whatever pace the team chooses.

Together, VKS and Traefik give VCF customers one platform for existing and new workloads and one gateway to govern them, from the first ingress route to the first AI agent in production. The accompanying repository holds the manifests and Helm values for all three models validated in this architecture, so teams can reproduce each deployment on their own VKS clusters.



Copyright © 2026 Broadcom. All rights reserved.
The term “Broadcom” refers to Broadcom Inc. and/or its subsidiaries. For more information, go to www.broadcom.com. All trademarks, trade names, service marks, and logos referenced herein belong to their respective companies. Broadcom reserves the right to make changes without further notice to any products or data herein to improve reliability, function, or design. Information furnished by Broadcom is believed to be accurate and reliable. However, Broadcom does not assume any liability arising out of the application or use of this information, nor the application or use of any product or circuit described herein, neither does it convey any license under its patent rights nor the rights of others.
Item No: vmw-bc-wp-tech-uslet-word-2026; Aug-26