TECHNICAL WHITE PAPER:
March 2026

Observability on vSphere Kubernetes Service

Reference Architecture

Contents

Executive Summary

Using the vSphere Kubernetes Service (VKS) on VMware Cloud Foundation enables organizations to run modern, containerized applications on a consistent on-premises cloud platform. As Kubernetes environments evolve, rapid state changes, short-lived workloads and frequent updates, the ability to maintain deep visibility becomes mission critical. The ability to detect issues early, identify patterns and diagnose performance bottlenecks before they impact services becomes vital.

This paper explores how to utilize a CNCF-aligned observability stack with Prometheus, Grafana, Loki and Jaeger to deliver a unified view across metrics, dashboards, alerts, and logs. In addition, we detail how VCF Operations can aid in visualizing the data. This approach ensures that both VI admins and DevOps teams have the real-time data needed to troubleshoot and optimize the entire environment, from the physical host to the workload and pod.

Organizations can achieve:

- **Holistic infrastructure monitoring**: Correlate VCF platform health with Supervisor and VKS cluster performance, linking physical resource pressure to Kubernetes node and workload behavior.
- **Operational simplicity through automation**: Use the Grafana Operator to manage dashboards and data sources as code, keeping observability consistent as environments scale.
- **Cost-efficient log management**: Leverage Loki's label-based indexing model to store and query high log volumes without the overhead typical of traditional log indexing solutions.
- **Reduced mean time to resolution (MTTR):** Provide a shared "single pane of glass" that reduces silos between infrastructure and application teams and accelerates cross-stack troubleshooting.

Introduction

The Cloud-Native Observability Challenge

With the added levels of abstraction in microservices architectures, there is a trade-off between flexibility and system complexity. Kubernetes environments pose several unique operational challenges that need to be considered:

- **Ephemeral workloads and identities:** Pods and namespaces are continuously created, destroyed, and rescheduled. This churn produces a fast stream of events and shifting endpoints, which can overwhelm monitoring approaches built around static IPs or long-lived hostnames.

- **Fragmented visibility across layers:** Resource pressure may be visible at the host or VM layer, yet difficult to attribute to a specific pod or service. Conversely, developers may observe application latency without seeing that the underlying datastore or network is the true bottleneck. This cross-layer visibility gap makes root-cause analysis harder.

- **Telemetry volume and velocity:** Microservices dramatically increase the number of metrics, logs, and traces generated. Each container emits its own telemetry, often at high frequency. Collecting, correlating, and retaining this data reliably becomes a significant operational burden.

The Three Pillars of Observability

To address these challenges, this paper proposes an architecture based on the three pillars of observability: *metrics, logs,* and *traces*, implemented through a standardized open-source toolchain that is native to the Kubernetes ecosystem. Together, these pillars provide end-to-end visibility across infrastructure and application layers, turning raw telemetry into actionable operational insight

- **Metrics (The Pulse):** Metrics are numerical measurements captured over time. They quantify system behavior, i.e. *what is changing and by how much* and form the foundation for dashboards and alerting. Using **Prometheus**, we collect signals such as CPU utilization, memory pressure, and network activity. Furthermore, by integrating the **Istio service mesh**, Prometheus automatically collects rich Layer 7 metrics (such as HTTP request volume, error rates, and latency) from every service without requiring developers to instrument their code.

- **Logs (The Narrative):** Whilst metrics indicate that something is wrong, logs provide the context needed to understand why. With **Grafana Loki**, we aggregate stdout/stderr streams from Kubernetes workloads and index them with Kubernetes metadata (namespace, pod, container, labels). This allows engineers to pivot from a metric anomaly (such as an increase in HTTP redirects) directly to the relevant log events emitted by the affected pods at the corresponding time.

- **Traces (The Journey):** Traces follow an individual request as it traverses multiple microservices, capturing timing and dependency relationships between components. While this paper focuses primarily on high-volume metrics and logs, distributed tracing completes the picture by isolating latency bottlenecks within multi-service request paths.

vSphere Kubernetes Service

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

Key benefits include:

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

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

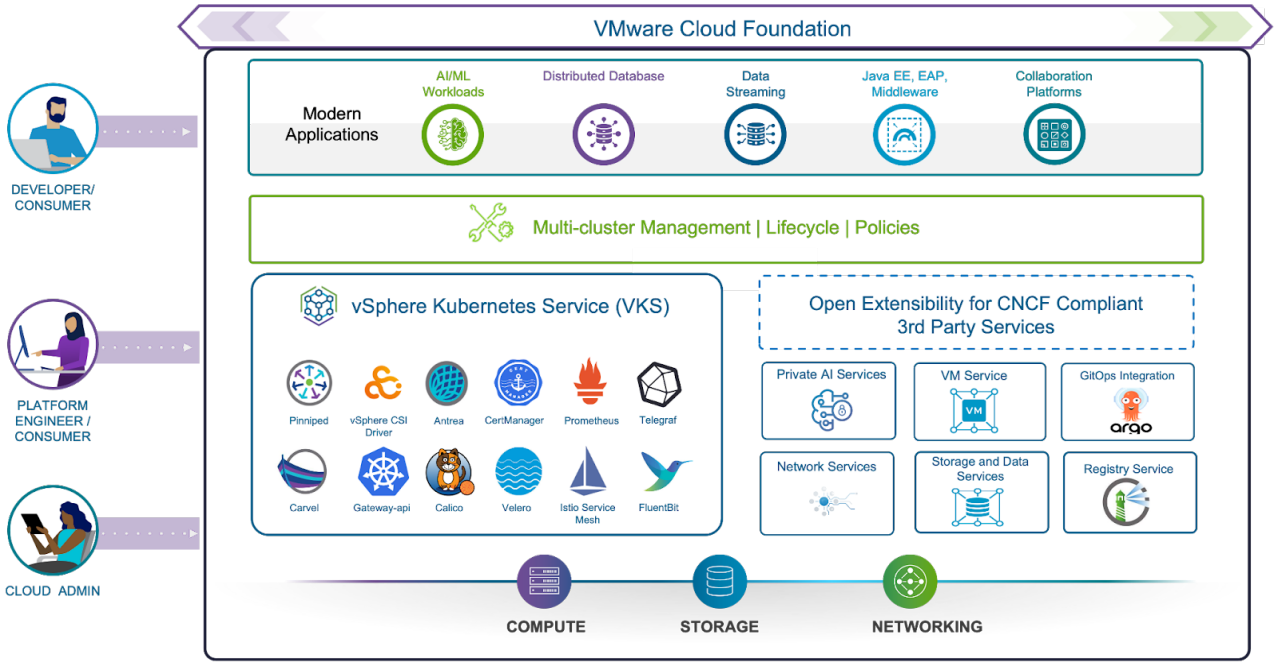

Figure 1: VMware Cloud Foundation with vSphere Kubernetes Service

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

https://github.com/vmware/vks-consumption-models/tree/main/observability-consumption-model

Solution Architecture

Core Infrastructure

This paper is based on a VCF 9.0 environment, integrating compute, networking, and storage to provide an holistic private cloud platform. For component details, visit:
https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vcf-9-0-and-later/9-0/release

Figure 2: Core Infrastructure

Component	Version	Notes
VCF	9.0.0	3x Hosts — Intel(R) Xeon(R) Gold 5218R CPU @ 2.10GHz 1023.62 GB Memory
Supervisor	v1.30.10	build 24845085

This core infrastructure provides the foundation on which the remainder of the solution is built. The platform used throughout this paper is a VCF 9.0 environment with vSphere Supervisor enabled, running across three hosts. This provides the underlying compute, storage, and networking services that support the Kubernetes clusters and platform components described in this implementation.

Kubernetes Environment

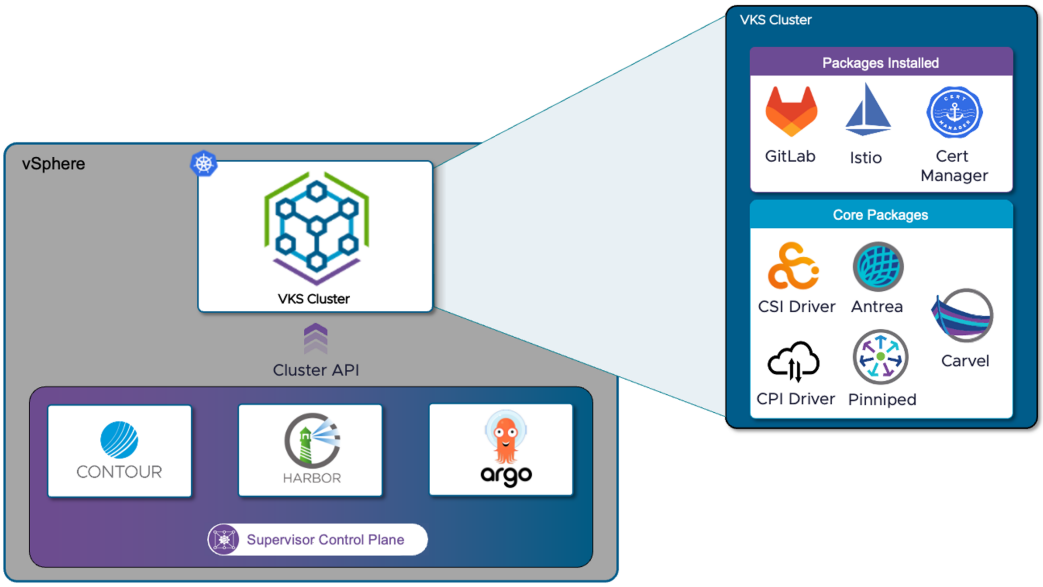

Figure 3: Supervisor Control Plane and vSphere Kubernetes Service

At its core, vSphere provides a consistent, highly available Supervisor control plane that leverages ESX hosts directly to provide essential services. 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). These VKS clusters are fully CNCF-conformant and include core packages such as Antrea for networking and the Cloud Storage Interface driver to provide storage. Moreover, Broadcom also provides a set of standard open-source packages for VKS, distributed in Carvel format and validated against supported VKS cluster versions. For further details see the release notes: https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/release-notes/vks-standard-packages-release-notes.html

Component	Version	Notes
vSphere Kubernetes Service	3.5	Control plane Service
Harbor	2.13.1+vmware.1-vks.1	Control plane Service
Contour	1.32.0+vmware.1-vks.1	Control plane Service
Supervisor Management Proxy	0.4.0	Control plane Service

VKS Cluster Configuration

A single VKS cluster was deployed using the configuration manifest below. 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 *guaranteed-large* profile, with 100% reservation (of the allocated 4 vCPU and 16 GiB of RAM per node). In addition, each node was associated with an extra 800 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 16GB RAM vSAN RAID 5	VM Class: "guaranteed large" Storage Class: "vsan-esa-default-policy-raid5"
Kubernetes Worker Nodes	3 Replicas — 4x vCPU 16GB RAM vSAN RAID 5 + Extra 800Gi Volume	VM Class: "guaranteed large" Storage Class: "vsan-esa-default-policy-raid5"
Kubernetes Release	v1.33.3+vmware.1-fips	
OS Image	Photon	
Cluster Domain	cluster.local	
CIDR Blocks	Pods: 192.168.156.0/20 Services: 10.96.0.0/12	

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/v1beta1
kind: Cluster
metadata:
  name: kubernetes-cluster-kmnr
  namespace: supervisor-namespace
spec:
  clusterNetwork:
    pods:
      cidrBlocks:
        - 192.168.156.0/20
    services:
      cidrBlocks:
        - 10.96.0.0/12
    serviceDomain: cluster.local
  topology:
    class: builtin-generic-v3.4.0
    version: v1.33.3---vmware.1-fips-vkr.1
    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=photon
    workers:
      machineDeployments:
        - class: node-pool
          name: kubernetes-cluster-kmnr-nodepool-nve9
          replicas: 3
          metadata:
            annotations:
              run.tanzu.vmware.com/resolve-os-image: os-name=photon
          variables:
            overrides:
              - name: volumes
                value:
                  - name: vol-5be3
                    mountPath: /var/lib/containerd
                    storageClass: vsan-esa-default-policy-raid5
                    capacity: 800Gi
```

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, several add-on packages were installed (such as cert-manager) while the monitoring bundle 'kube-prometheus-stack' was installed using Helm.

For details on how to install add-on packages, 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.html

In addition, reference documentation for the packages can be found here: https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/managing-vsphere-kuberenetes-service-clusters-and-workloads/installing-standard-packages-on-tkg-service-clusters/standard-package-reference.html

Package	Version	Notes
Antrea	2.3.1	VKS core package
Gateway API	1.2.1	VKS core package
Guest-cluster auth-service	1.4.2	VKS core package
Metrics Server	0.7.2	VKS core package
Pinniped	0.39.0	VKS core package

Secretgen Controller	0.19.1	VKS core package
vSphere CPI	1.33.0	VKS core package
vSphere CSI	3.5.0	VKS core package
Istio	1.27.1	VKS add-on package
Cert-Manager	1.18.2	VKS add-on package
Telegraf	1.35.4	VKS add-on package
Fluent Bit	4.0.8	VKS add-on package

Solution Validation

Monitoring

Monitoring remains a foundational part of observability: metrics provide the structured, high-level view needed to understand system health, capacity, and performance over time. In Kubernetes, where workloads are distributed, elastic, and frequently rescheduled, monitoring must adapt dynamically to changes in pods, services, and infrastructure in order to remain accurate, timely, and operationally useful.

Prometheus serves as the *de facto* industry standard time-series database for real-time, high-resolution monitoring in Kubernetes. In this model, applications publish statistics to an endpoint (usually '/metrics'), and Prometheus then *scrapes* (ingests) this data. Typically, organizations then use **Grafana** to observe and visualize these metrics.

At the core of this stack lies the **Prometheus Operator** which manages both *Prometheus* and *Alertmanager* as Kubernetes-native resources, i.e. using Custom Resource Definitions (CRDs). Thus, instead of maintaining static scrape targets, rules are declared via CRDs that the operator then acts on, significantly reducing operational complexity as the cluster scales.

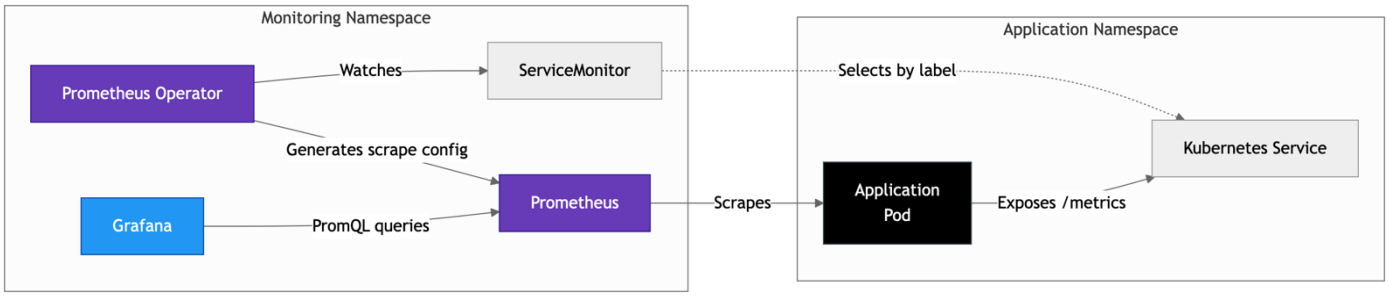

Figure 4: Overview of Kubernetes monitoring architecture using Prometheus and Grafana

To ensure comprehensive observability within the VKS cluster, we deploy the **Prometheus Community Stack** (kube-prometheus-stack). This widely adopted Helm chart provides a complete monitoring layer, integrating Prometheus for metrics, Grafana for visualization, and Node Exporter for hardware telemetry.

First, we ensure that the relevant VKS addon packages have been installed. For example, we can see details for Istio, confirming that version 1.27.1 is installed:

```
# Verify installation of Istio
kubectl -n vmware-system-tkg describe $(kubectl get pkgi -A -o name | grep istio)

Name:         kubernetes-cluster-kmnr-istio
Namespace:    vmware-system-tkg
Labels:       addon.kubernetes.vmware.com/addon-name=istio
              addons.kubernetes.vmware.com/addoninstall-namespace=supervisor-namespace
              cluster.x-k8s.io/cluster-name=kubernetes-cluster-kmnr
              kubernetes.vmware.com/managed-by=vks
Annotations:  addon.kubernetes.vmware.com/cluster-addon-name: kubernetes-cluster-kmnr-istio
              addons.kubernetes.vmware.com/addoninstall-name: istio-kubernetes-cluster-kmnr
              cluster.x-k8s.io/cluster-namespace: supervisor-namespace
API Version:  packaging.carvel.dev/v1alpha1
Kind:         PackageInstall
Metadata:
  Creation Timestamp:  2026-01-20T12:11:45Z
  Finalizers:
    finalizers.packageinstall.packaging.carvel.dev/delete
  Generation:        1
  Resource Version:  170973743
  UID:               f4b69124-0210-407a-9acc-efaf861e2040
```

```
Spec:
  Package Ref:
    Ref Name:  istio.kubernetes.vmware.com
    Version Selection:
      Constraints:       1.27.1+vmware.1-vks.1
  Service Account Name:  tanzu-cluster-bootstrap-sa
  Values:
    Secret Ref:
      Name:  kubernetes-cluster-kmnr-istio-values
Status:
  Conditions:
    Status:                True
    Type:                  ReconcileSucceeded
  Friendly Description:    Reconcile succeeded
  Last Attempted Version:  1.27.1+vmware.1-vks.1
  Observed Generation:     1
  Version:                 1.27.1+vmware.1-vks.1
Events:                    <none>
```

We then proceed to install the 'kube-prometheus-stack' Helm chart.

Note: We must set the Pod Security Admission (PSA) policy to *privileged* as Node Exporter needs direct access to the underlying cluster nodes (i.e. the VKS VMs) to harvest hardware and OS metrics

```
# Add the Prometheus Community repo
helm repo add prometheus-community \
  https://prometheus-community.github.io/helm-charts

# Create namespace and set permissions
kubectl create ns monitoring
kubectl label --overwrite ns monitoring \
  pod-security.kubernetes.io/enforce=privileged

# Install the operator includes (Prometheus & Grafana)
helm install prometheus prometheus-community/kube-prometheus-stack \
  --namespace monitoring
```

High-Security Alternative If we want to keep permissions more restrictive (for security), we can disable the Node Exporter component. Since Node Exporter requires access to the underlying host system, removing it allows us to enforce the standard baseline security policy instead of the more permissive privileged policy.

```
# Create namespace with standard 'baseline' security
kubectl create ns monitoring
kubectl label --overwrite ns monitoring \
  pod-security.kubernetes.io/enforce=baseline

# Install the stack without Node Exporter
helm install prometheus prometheus-community/kube-prometheus-stack \
  --namespace monitoring \
  --set nodeExporter.enabled=false
```

After installation, we can confirm a number of pods have been created

```
# Show running pods
kubectl -n monitoring get pods -o custom-columns=NAME:.metadata.name,\
STATUS:.status.phase,\
READY:'.status.containerStatuses[*].ready'

NAME                                                   STATUS    READY
alertmanager-prometheus-kube-prometheus-alertmanager-0   Running   true,true
prometheus-grafana-6687f5b74f-sbdlx                    Running   true,true,true
prometheus-kube-prometheus-operator-7bcc9b675d-zwqmk   Running   true
prometheus-kube-state-metrics-8457d8c49f-95th4         Running   true
```

```
prometheus-prometheus-kube-prometheus-prometheus-0       Running   true,true
prometheus-prometheus-node-exporter-5629r                Running   true
prometheus-prometheus-node-exporter-jj4tx                Running   true
prometheus-prometheus-node-exporter-lqn7p                Running   true
prometheus-prometheus-node-exporter-nksdc                Running   true
prometheus-prometheus-node-exporter-pngrk                Running   true
prometheus-prometheus-node-exporter-qjtjq                Running   true
```

Further, we can see the services created and associated ports (we can use this information to later target the internal ports for our gateway).

```
# Show services
kubectl -n monitoring get svc -o custom-columns=NAME:.metadata.name,\
TYPE:.spec.type,\
IP:.spec.clusterIP,\
PORT:'.spec.ports[*].port'

NAME                                    TYPE        IP              PORT
alertmanager-operated                   ClusterIP   None            9093,9094,9094
prometheus-grafana                      ClusterIP   10.104.67.6     80
prometheus-kube-prometheus-alertmanager ClusterIP   10.96.143.228   9093,8080
prometheus-kube-prometheus-operator     ClusterIP   10.110.95.31    443
prometheus-kube-prometheus-prometheus   ClusterIP   10.109.221.198  9090,8080
prometheus-kube-state-metrics           ClusterIP   10.108.75.231   8080
prometheus-operated                     ClusterIP   None            9090
prometheus-prometheus-node-exporter     ClusterIP   10.105.184.122  9100
```

While the Helm chart automates the deployment of the monitoring stack, it is critical to understand the security model it enforces.

Operationally, the deployment establishes a dedicated programmatic identity (a ServiceAccount) for the monitoring processes, assigning it a specific set of global permissions (ClusterRole).

To verify this architecture, we can inspect the service accounts created by the Helm release. Note that each component (Grafana, Alertmanager, Prometheus) is assigned a distinct identity to limit the blast radius of a potential compromise.

```
kubectl -n monitoring get sa -l app.kubernetes.io/instance=prometheus

NAME                                    SECRETS   AGE
prometheus-grafana                      0         148m
prometheus-kube-prometheus-alertmanager 0         148m
prometheus-kube-prometheus-operator     0         148m
prometheus-kube-prometheus-prometheus   0         148m
prometheus-kube-state-metrics           0         148m
prometheus-prometheus-node-exporter     0         148m
```

These identities consume specific cluster roles defined by the chart:

```
kubectl get clusterroles -l app.kubernetes.io/instance=prometheus

NAME                                   CREATED AT
prometheus-grafana-clusterrole         2026-02-09T12:25:11Z
prometheus-kube-prometheus-operator    2026-02-09T12:25:11Z
prometheus-kube-prometheus-prometheus  2026-02-09T12:25:11Z
prometheus-kube-state-metrics          2026-02-09T12:25:11Z
```

By inspecting the Prometheus role in detail, we can confirm the *principle of least privilege*. The output below demonstrates that the system is granted only read-only access (verbs: get, list, watch) to the necessary API groups for service discovery, with no permissions to modify the environment.

```
kubectl describe clusterrole prometheus-kube-prometheus-prometheus

Name:         prometheus-kube-prometheus-prometheus
Labels:       app.kubernetes.io/instance=prometheus
              app.kubernetes.io/managed-by=Helm

PolicyRule:
  Resources                       Non-Resource URLs    Resource Names  Verbs
  ---------                       -----------------    --------------  -----
  endpoints                       []                   []              [get list watch]
  nodes/metrics                   []                   []              [get list watch]
  nodes                           []                   []              [get list watch]
  pods                            []                   []              [get list watch]
  services                        []                   []              [get list watch]
  endpointslices.discovery.k8s.io []                   []              [get list watch]
  ingresses.networking.k8s.io     []                   []              [get list watch]
                                  [/metrics/cadvisor]  []              [get]
                                  [/metrics]           []              [get]
```

Since the monitoring stack runs in an isolated network namespace, we must explicitly define how external traffic reaches the dashboards. We will use the built-in Kubernetes Gateway CRD to create a gateway of class 'Istio' and then configure an *HTTPRoute* to forward traffic from the Gateway to the specific ports we identified previously.

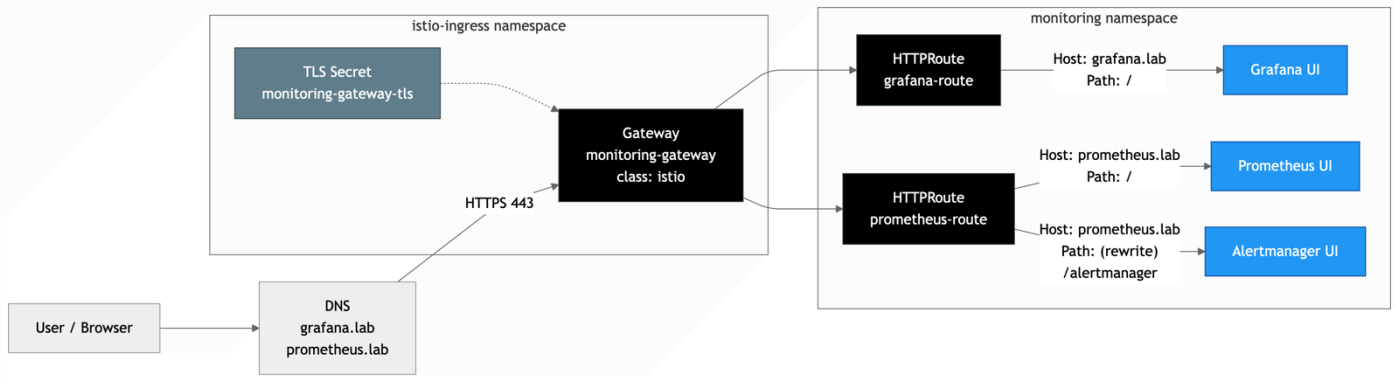

Figure 5: Istio ingress gateway and http routes to services

To secure our Ingress Gateway connections, we use cert-manager configured with an enterprise-trusted *ClusterIssuer*. Rather than relying on self-signed certificates, this architecture integrates directly with Active Directory Certificate Services (ADCS) to automatically provision and rotate CA-backed certificates. Detailed instructions for configuring this ADCS integration and the prerequisite 'Web Server' derived template can be found in Appendix B.

```
# Create ns for Istio ingress gw
NS=istio-ingress
kubectl create ns $NS
kubectl label --overwrite ns $NS \
  pod-security.kubernetes.io/enforce=baseline


# Apply the Certificate Request
kubectl apply -f - <<EOF
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: monitoring-gateway-cert
  namespace: istio-ingress
spec:
  secretName: monitoring-gateway-tls
  commonName: "*.lab"
```

```
  dnsNames:
    - "*.lab"
  issuerRef:
    group: adcs.certmanager.csf.nokia.com
    kind: ClusterAdcsIssuer
    name: adcs-cluster-issuer
EOF
```

Finally, we apply the Gateway configuration. This references the secret *monitoring-gateway-tls* that cert-manager just created. We also enforce the Namespace Trust Policy *gateway-trusted* to prevent unauthorized routes.

```
# Label the monitoring namespace so the Gateway trusts it
kubectl label ns monitoring access=gateway-trusted --overwrite

# Apply the Gateway Listener
kubectl apply -f - <<EOF
kind: Gateway
apiVersion: gateway.networking.k8s.io/v1
metadata:
  name: monitoring-gateway
  namespace: istio-ingress
spec:
  gatewayClassName: istio
  listeners:
  - name: https
    protocol: HTTPS
    port: 443
    hostname: "*.lab"
    tls:
      mode: Terminate
      certificateRefs:
      - name: monitoring-gateway-tls
    allowedRoutes:
      namespaces:
        from: Selector
        selector:
          matchLabels:
            access: gateway-trusted
EOF
```

The routing to the backend services can then be defined via the manifest below, using *HTTPRoute*.

Note for alert manager: by default, when a user requests *https://domain.com/alertmanager*, the ingress controller forwards that full path to the backend service. However, the alert manager application **expects** to receive requests at its root (i.e. to '/'), and if it receives a request for '/alertmanager', it will return a 404 error (as it does not have an internal handler for that sub-path).

To solve this without modifying the application's internal configuration, we utilize the **URLRewrite Filter** provided by the Gateway API. The Gateway takes the request for '/alertmanager' and issues as re-write to '/'. Thus any and all requests for alert manager are forwarded to the backend correctly. This decoupled approach allows us to expose multiple backend tools under a single domain namespace without requiring complex reconfiguration of the applications themselves.

Note: The hostnames for Grafana and Prometheus will need to be resolvable; these should be added to DNS

```
kubectl apply -f - <<EOF
# 1. Grafana Route -> Service Port 80
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: grafana-route
  namespace: monitoring
```

```
spec:
  parentRefs:
  - name: monitoring-gateway
    namespace: istio-ingress
    sectionName: https
  hostnames:
  - "grafana.lab"
  rules:
  - matches:
    - path:
        type: PathPrefix
        value: /
    backendRefs:
    - name: prometheus-grafana
      port: 80
---

# 2. Prometheus & Alert Manager Route -> Service Ports 9090 & 9093
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: prometheus-route
  namespace: monitoring
spec:
  parentRefs:
  - name: monitoring-gateway
    namespace: istio-ingress
    sectionName: https
  hostnames:
  - "prometheus.lab"
  rules:

  # Rule A: Alertmanager (Sub-path with Rewrite)
  # Traffic to '/alertmanager' is rewritten to '/' and sent to port 9093
  - matches:
    - path:
        type: PathPrefix
        value: /alertmanager
    filters:
    - type: URLRewrite
      urlRewrite:
        path:
          type: ReplacePrefixMatch
          replacePrefixMatch: /
    backendRefs:
    - name: prometheus-kube-prometheus-alertmanager
      port: 9093

  # Rule B: Prometheus (Root Path)
  # Traffic to '/' is sent to port 9090
  - matches:
    - path:
        type: PathPrefix
        value: /
    backendRefs:
    - name: prometheus-kube-prometheus-prometheus
      port: 9090
EOF
```

We can then verify the gateway was reconciled correctly and an IP address was assigned

```
kubectl -n istio-ingress get gateway monitoring-gateway

NAME                 CLASS     ADDRESS        PROGRAMMED   AGE
```

```
monitoring-gateway   contour   10.163.44.47   True         3m57s
```

Further, we verify that the routing has been accepted by the controller by inspecting the parent status of the route. This confirms that the Gateway has recognized the configuration and successfully attached it to the HTTPS listener.

```
kubectl -n monitoring get httproutes -o \
  jsonpath='{.items[*].status.parents}' | jq

[
  {
    "conditions": [
      {
        "lastTransitionTime": "2026-02-10T11:25:54Z",
        "message": "References resolved",
        "observedGeneration": 2,
        "reason": "ResolvedRefs",
        "status": "True",
        "type": "ResolvedRefs"
      },
      {
        "lastTransitionTime": "2026-02-10T11:25:54Z",
        "message": "Accepted HTTPRoute",
        "observedGeneration": 2,
        "reason": "Accepted",
        "status": "True",
        "type": "Accepted"
      }
    ],
    "controllerName": "projectcontour.io/gateway-controller",
    "parentRef": {
      "group": "gateway.networking.k8s.io",
      "kind": "Gateway",
      "name": "monitoring-gateway",
      "namespace": "istio-ingress",
      "sectionName": "https"
    }
  }
]
[
  {
    "conditions": [
      {
        "lastTransitionTime": "2026-02-10T11:25:57Z",
        "message": "References resolved",
        "observedGeneration": 2,
        "reason": "ResolvedRefs",
        "status": "True",
        "type": "ResolvedRefs"
      },
      {
        "lastTransitionTime": "2026-02-10T11:25:57Z",
        "message": "Accepted HTTPRoute",
        "observedGeneration": 2,
        "reason": "Accepted",
        "status": "True",
        "type": "Accepted"
      }
    ],
    "controllerName": "projectcontour.io/gateway-controller",
    "parentRef": {
      "group": "gateway.networking.k8s.io",
      "kind": "Gateway",
      "name": "monitoring-gateway",
      "namespace": "istio-ingress ",
```

```
      "sectionName": "https"
    }
  }
]
```

We can then test access via *curl*: first, we retrieve the external IP address assigned to the gateway, then issue requests to verify both the primary dashboard (Grafana) and the rewritten backend path (Alertmanager). These should return a '200 OK' message to indicate the ingress stack is fully operational.

```
# Get the Gateway's External IP
GATEWAY_IP=$(kubectl get gateway -n istio-ingress monitoring-gateway \
  -o jsonpath='{.status.addresses[0].value}')

# Check Grafana access
curl -ik \
  --resolve grafana.lab:443:$GATEWAY_IP https://grafana.lab/login

# Check Prometheus access & URL Re-write
curl -ik \
  --resolve prometheus.lab:443:$GATEWAY_IP https://prometheus.lab/alertmanager
```

Opening a browser to each (as defined above, to https://prometheus.lab and https://grafana.lab) shows the respective user interfaces

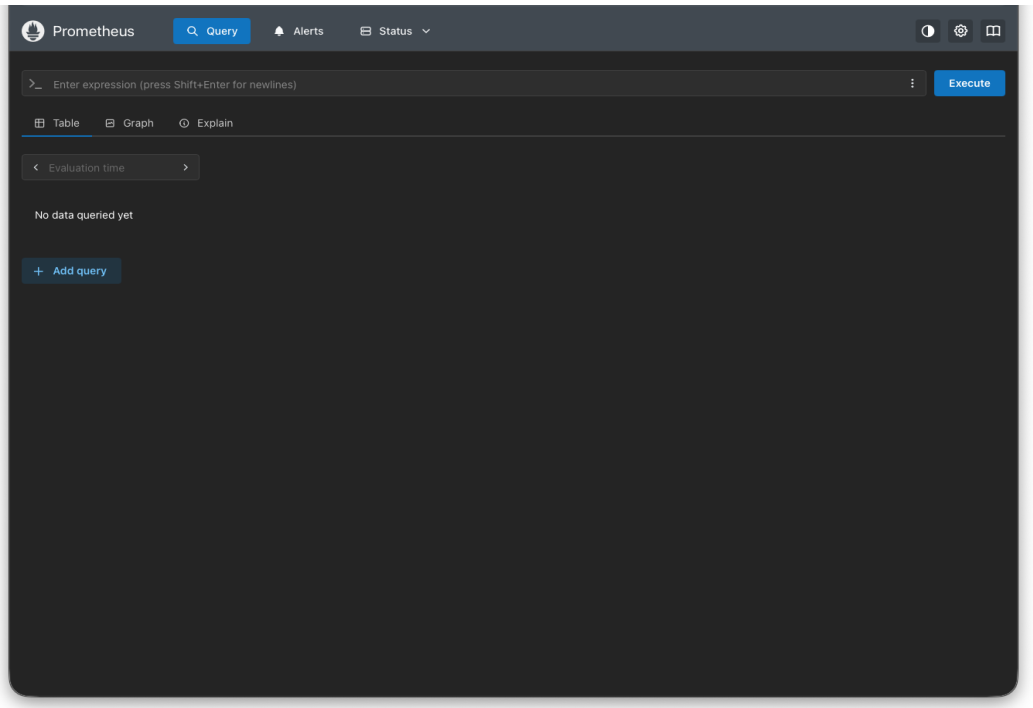

Figure 6: Prometheus UI

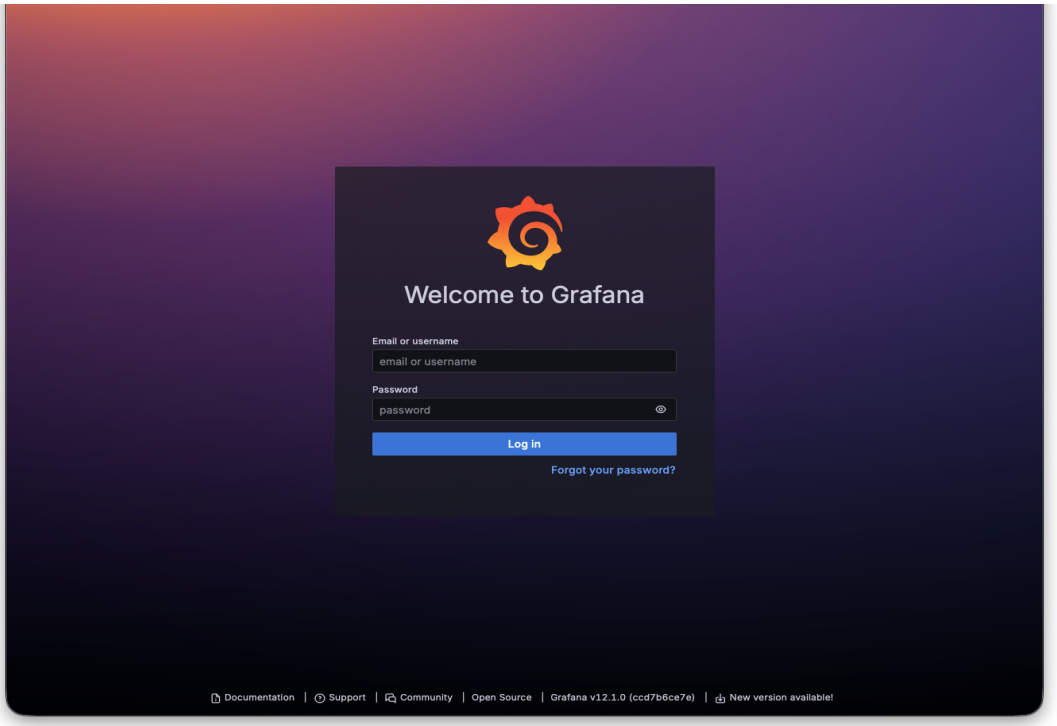

Figure 7: Grafana UI

On install, Grafana creates a randomized admin password. We can retrieve this by probing the secret in the monitoring namespace

```
kubectl -n monitoring get secret prometheus-grafana \
  -o jsonpath="{.data.admin-password}" | base64 --decode ; echo
```

To confirm the stack is correctly scraping secure node endpoints, we will inspect the **Kubelet** dashboard.

1. **Log in** to Grafana with the admin password obtained earlier
2. Click the **Dashboards** icon in the left sidebar.
3. Navigate to **Kubernetes / Kubelet**.

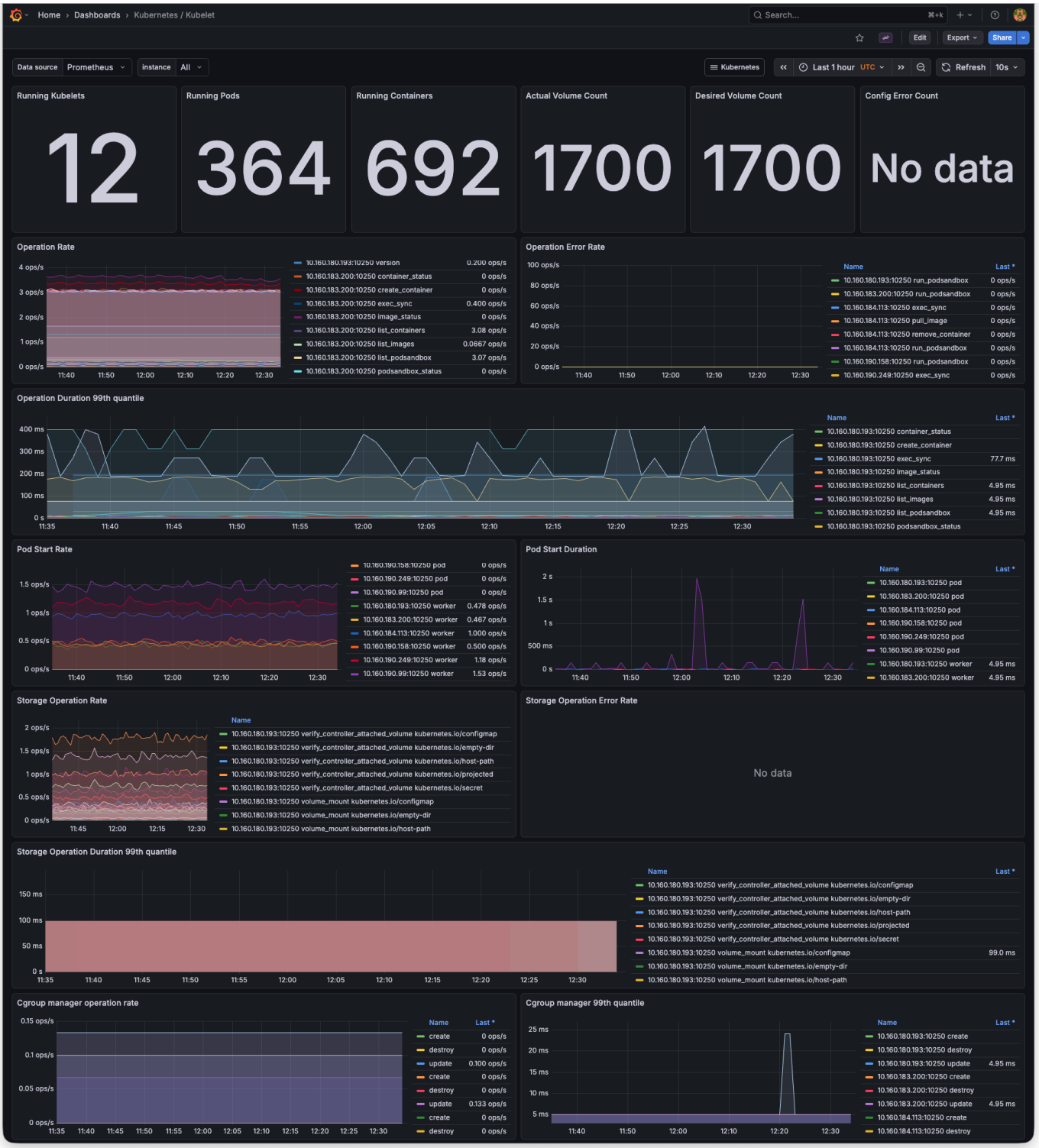

Figure 8: Grafana dashboard showing Kubelet metrics

From this dashboard, we notice that:

- The **"Running Kubelets: 12"** panel confirms that Prometheus has discovered and is successfully scraping all 12 nodes in the cluster.
- The **"Operation Rate"** and **"Pod Start Rate"** graphs are populated with live data. This proves that Prometheus is authorized to read the Kubelet's internal runtime statistics (e.g., kubelet_runtime_operations_total).

- The **"Config Error Count: No data"** and empty **"Storage Operation Error Rate"** panels are positive indicators. In this context, "No data" means "Zero Errors" thus confirming the nodes are healthy.

Thus, we have successfully deployed a production-grade observability stack. Here we can clearly see that the VKS cluster statistics are captured from Prometheus and can be visualized in real-time using Grafana. In the next stage, we will deploy a sample application to demonstrate how to leverage this monitoring for production apps.

Instead of pushing data, cloud-native applications simply *expose* their metrics over a standard HTTP endpoint (typically to '`/metrics`'). This endpoint serves a plain-text list of key-value pairs representing the application's internal state (such as request counts, memory usage, or active threads). Prometheus then *scrapes* (polls) this endpoint at a defined interval to ingest the data.

To automate this process in a dynamic cluster, we utilize the Custom Resource *ServiceMonitor*. This instructs Prometheus which Services to watch, which port to scrape, and how frequently to pull the data.

In the following steps, we will deploy a sample application that demonstrates this. We will then configure a ServiceMonitor to pull data into our observability stack. We make use of the Prometheus example app (see https://github.com/brancz/prometheus-example-app).

From the documentation:

```
Usage is simple, on any request to / the request will result in a 200 response code. This
increments the counter for this response code. Similarly the /err endpoint will result in a
404 response code, therefore increments that respective counter. Duration metrics are also
exposed for any request to /.

Prometheus metrics are exposes via requests to /metrics.
```

We can then deploy this within the namespace 'my-app'

```
# Create the application namespace & set permissions
kubectl create ns my-app
kubectl label --overwrite \
  ns my-app pod-security.kubernetes.io/enforce=baseline


# Deploy the application and service
kubectl apply -f - <<EOF
apiVersion: apps/v1
kind: Deployment
metadata:
  name: prom-example-app
  namespace: my-app
  labels:
    app: prom-example-app
spec:
  replicas: 2
  selector:
    matchLabels:
      app: prom-example-app
  template:
    metadata:
      labels:
        app: prom-example-app
    spec:
      containers:
      - name: prom-example-app
        image: fabxc/instrumented_app
        ports:
        - name: web
          containerPort: 8080
---
kind: Service
```

```
apiVersion: v1
metadata:
  name: prom-example-app
  namespace: my-app
  labels:
    app: prom-example-app
spec:
  selector:
    app: prom-example-app
  ports:
  - name: web
    port: 8080
EOF
```

Next, we define the service monitor. Note, this configuration sits inside the monitoring namespace (rather than with the application), providing separation between application and monitoring configuration.

```
kubectl apply -f - <<EOF
apiVersion: monitoring.coreos.com/v1
kind: ServiceMonitor
metadata:
  name: prom-example-app-monitor
  namespace: prometheus-operator
  labels:
    release: prometheus
spec:
  # Explicitly select the tenant namespace
  namespaceSelector:
    matchNames:
    - my-app
  selector:
    matchLabels:
      app: prom-example-app
  endpoints:
  - port: web
    interval: 15s
EOF
```

To test, we create a temporary pod that generates load (until aborted) via a loop of HTTP requests. We let this run for at least two minutes.

```
kubectl run -i --tty load-generator \
  --rm \
  --image=registry.k8s.io/e2e-test-images/busybox:1.29-2 \
  --restart=Never \
  --namespace=my-app \
  -- /bin/sh -c
     "while sleep 0.01; do wget -q -O- http://prom-example-app:8080; echo; done"
```

We can then verify that Prometheus is scraping the application data defined in *ServiceMonitor*. This can be seen on the Prometheus UI by navigating to **Status** > **Targets** and searching for '*prom-example-app*'. Here, we can see that there are two endpoints (as we had defined two replicas) and both are shown as 'up'.

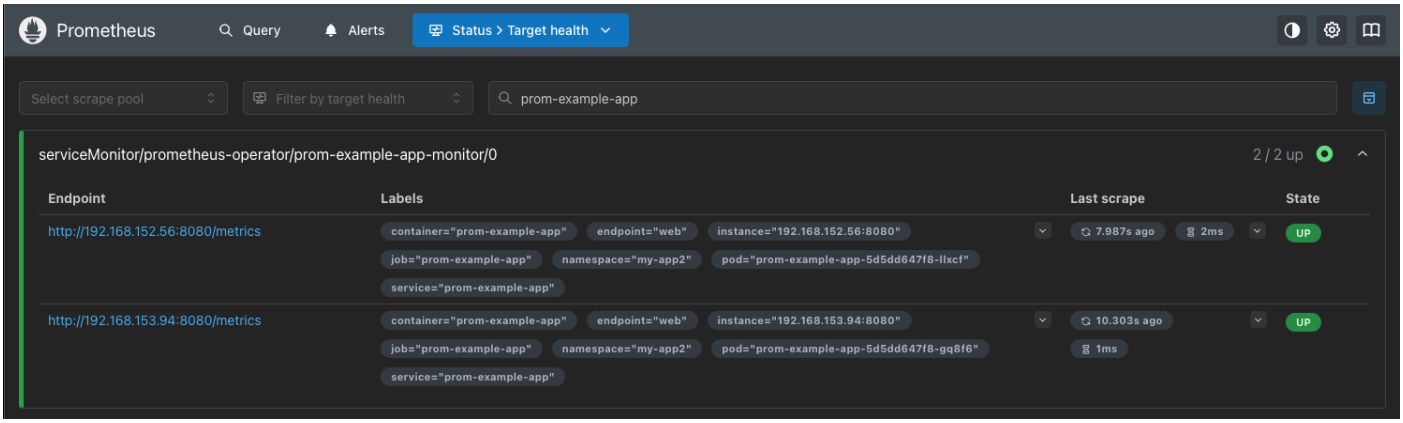

Figure 9: Service monitor endpoints

On the Grafana UI, we can see the data visually by running a custom query. Navigating to the **Explore** tab, we can run a query to see the total http request: '`rate(http_requests_total{job="prom-example-app"}[5m])`'

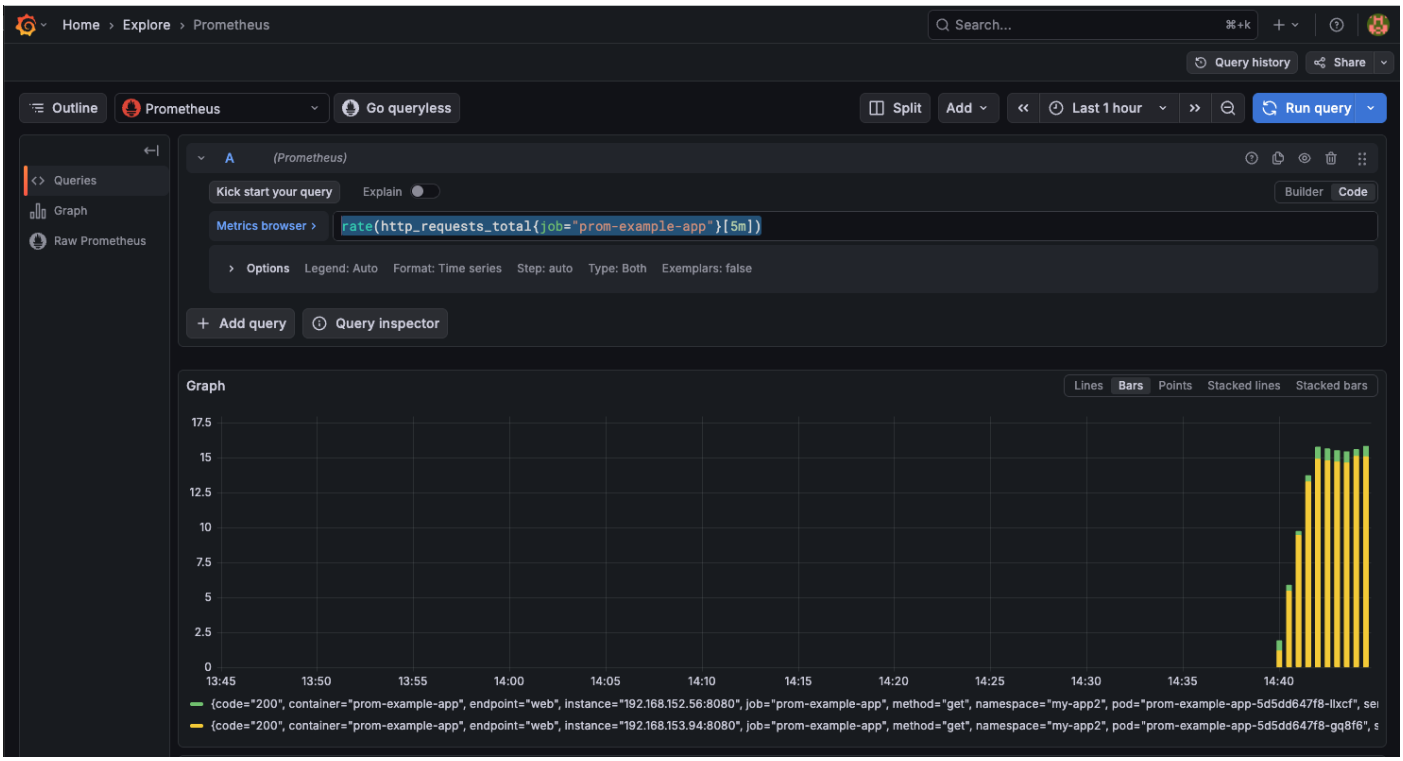

Figure 10: Grafana query on the number of http requests

Service Mesh Observability

Here, we demonstrate how to leverage Istio to surface application-level traffic metrics from the mesh data plane. Rather than relying on the app to expose a metrics endpoint, we can collect consistent metrics from Istio's Envoy sidecars.

First, we ensure that Istio's system pods are up and running

```
kubectl -n istio-system get pods | grep istio

istio-cni-node-f59vw             1/1     Running   0          21d
istio-cni-node-lppwn             1/1     Running   0          21d
istio-cni-node-xprgl             1/1     Running   0          21d
istio-support-678f869466-hfwwt   1/1     Running   0          21d
istiod-6c974957b7-gnskn          1/1     Running   0          21d
```

Next, we make use of the example app (the prometheus-example-app from the previous section) and label the namespace for Istio to inject its sidecars

```
# Label for Istio & restart the app pods
kubectl label namespace my-app istio-injection=enabled
kubectl rollout restart deployment/prom-example-app -n my-app
```

This will inject an additional container *'istio-proxy'* into each pod (this is the Istio *sidecar*). We run a quick sanity check to make sure the Istio containers are ready

```
# Check that the Istio sidecar is ready
NS=my-app
kubectl -n $NS get pods -o name | \
  xargs -I {} kubectl -n $NS logs {} -c istio-proxy |  \
  awk '{print $4,$5,$6,$7,$8}' | \
  grep ready

proxy is ready
proxy is ready
```

The Prometheus Operator requires explicit configuration to scrape metrics from the Istio sidecars. We apply a *PodMonitorresource* that targets the dedicated Envoy metrics port (15090). This ensures that the platform captures mesh traffic data independent of the application's own metric endpoints.

```
kubectl apply -f - <<EOF
apiVersion: monitoring.coreos.com/v1
kind: PodMonitor
metadata:
  name: istio-sidecars
  namespace: prometheus-operator
  labels:
    release: prometheus
spec:
  # Select all pods with the Istio sidecar injected
  selector:
    matchLabels:
      security.istio.io/tlsMode: istio
  namespaceSelector:
    matchNames:
    - my-app2
  # Target the standard Envoy metrics port
  podMetricsEndpoints:
  - path: /stats/prometheus
    port: http-envoy-prom   # Port 15090
    interval: 15s
EOF
```

As before, we generate synthetic traffic between a client pod and the demo app, which will allow us to observe the metrics generated by the mesh proxies.

```
# Deploy a stable client pod (sleeper) to act as the traffic source
kubectl run -n my-app2 debug-sleeper \
  --image=registry.k8s.io/e2e-test-images/busybox:1.29-2 \
  --restart=Never \
  -- /bin/sh -c "sleep 3600"

# Wait for the sidecar to be fully initialized (2/2 Ready)
echo "Waiting for sidecar injection..."
kubectl wait --for=condition=Ready pod/debug-sleeper -n my-app2 --timeout=60s

# Generate traffic loop (Mesh Client -> Application)
# Press Ctrl+C to stop
kubectl exec -it -n my-app2 debug-sleeper -- \
  /bin/sh -c \
  "while true; do wget -q -O- http://prom-example-app:8080; \
  echo -n .; sleep 0.1; done"
```

On the Prometheus UI, we would then expect to see the statistics from the pod monitor we created. Navigating to **Status** > **Target health** and searching for 'istio-sidecar' gives us the monitoring endpoints

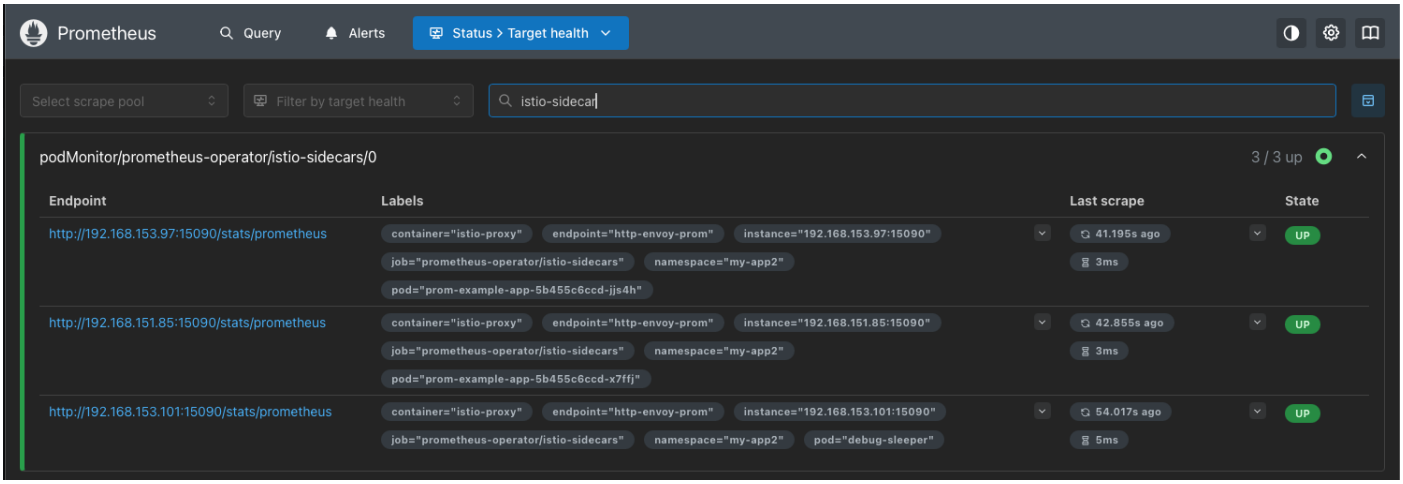

Figure 11: Istio sidecar endpoints

On the Grafana UI, we then issue queries, such as '`istio_requests_total{destination_service_name="prom-example-app"}`' which (as the name suggests) shows the total requests to the app. Below are several examples; note the query on each.

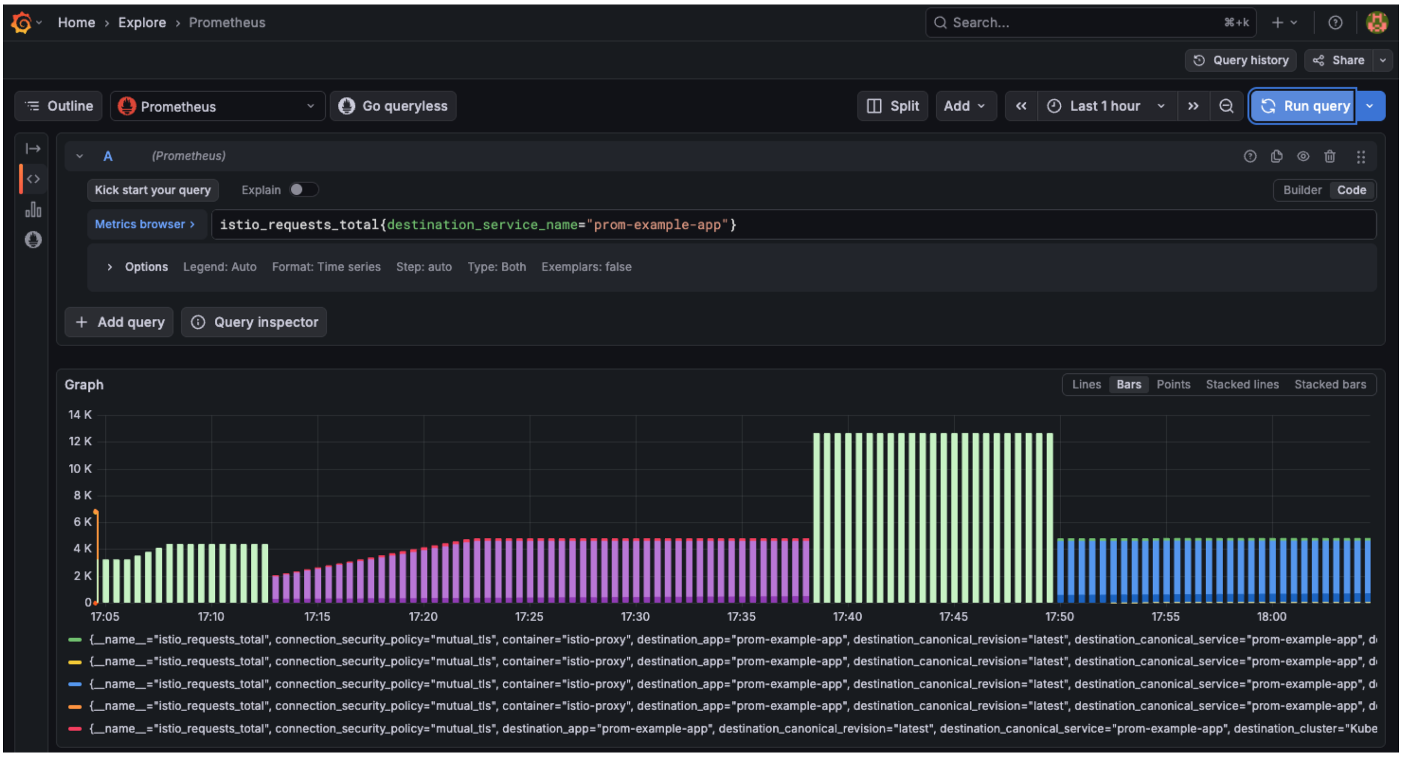

Figure 12: Grafana dashboard for the total number of Istio requests to the app over time

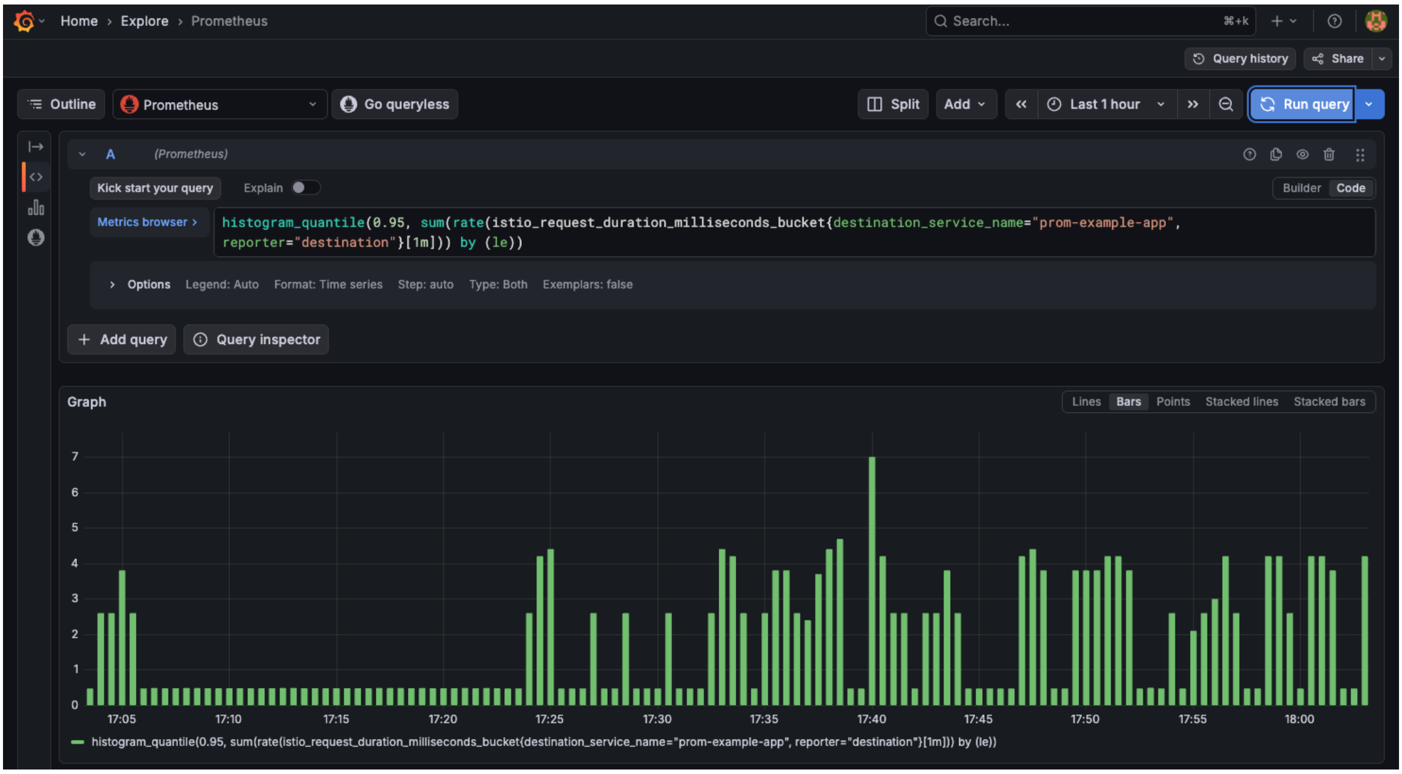

Figure 13: Executing a PromQL query in the Grafana Explore view

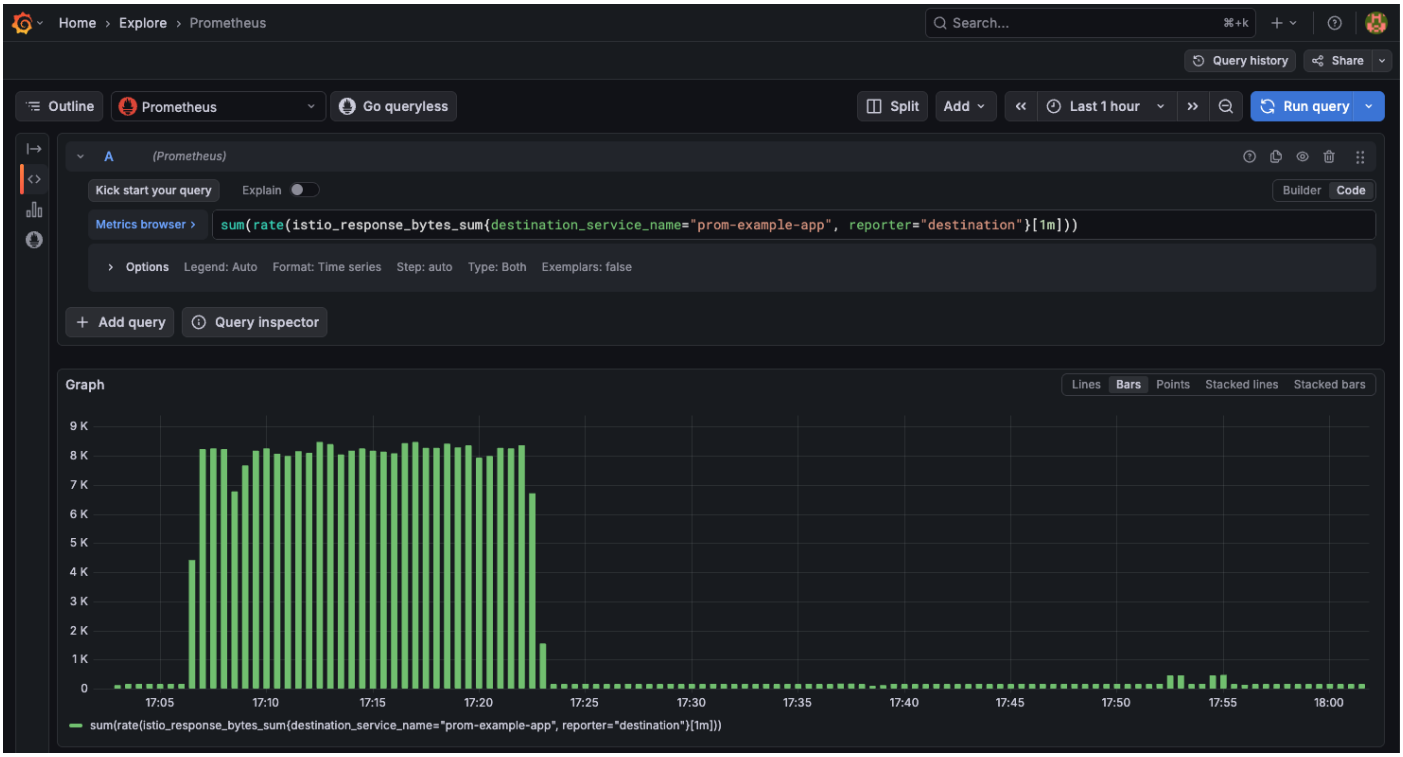

Figure 14: Analyzing response byte metrics in the Grafana Explore interface

Integration with VCF Operations

While the Prometheus Operator is indispensable for real-time, cluster-level observability, its scope is inherently limited to the Kubernetes layer. It remains blind to the underlying physical components (such as physical hosts) that support the VKS environment. To bridge this gap, we integrate **VCF Operations**, which extends visibility beyond the Kubernetes cluster boundary.

The primary value-add of this integration is the ability to visualize the infrastructure dependencies of the Kubernetes layer. For example, while Prometheus can report that a worker node is experiencing high I/O latency, it cannot discern the root cause outside the Kubernetes cluster. In contrast, by correlating VKS metrics with vSphere telemetry, VCF Operations can reveal that the underlying physical host is congested due to a "noisy neighbor": a completely separate, non-Kubernetes workload consuming shared resources. This level of full-stack root cause analysis is impossible to achieve with Prometheus alone.

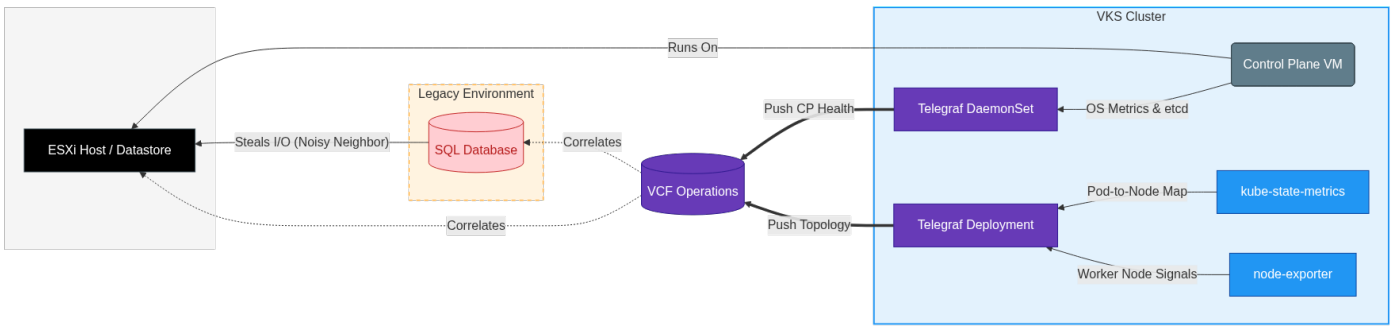

Figure 15: VCF Operations architecture for monitoring and correlating Kubernetes and legacy workloads

To integrate cluster metrics with VCF Operations, we leverage **Telegraf** as a lightweight collector alongside the monitoring stack. In this architecture, Telegraf acts purely as an egress bridge:

- **Collection:** Telegraf pulls metrics directly from the shared exporters deployed by the Prometheus stack (specifically *node exporter* and *kube state metrics*), rather than scraping individual application targets or querying the Prometheus.
- **Transmission:** Telegraf authenticates to VCF Operations and forwards this curated metric set to the central analytics engine over a controlled egress path.

To facilitate this, we update the default installation parameters of Telegraf using the manifest below:

```
# telegraf-data-values.yaml

isMetricProxyConfigured: true
domainName: cluster.local
namespace: tanzu-system-telegraf
createNamespace: true

# Silence local logging (metrics are pushed to VCF Ops)
outputPlugins:
  file:
    enabled: true
    files: ["/dev/null"]

# CONTROL PLANE HEALTH (DAEMONSET)
# Runs ONLY on Control Plane nodes (as we have node-exporter below)
cpOnlyforDaemonSet: true

inputPluginsForDaemonSet:
  # Host OS metrics (Control Plane VMs only)
  hostCPU: { enabled: true }
  hostMem: { enabled: true }
  hostDisk: { enabled: true }
  hostNet: { enabled: true }

  # Etcd / Control Plane specific metrics
  etcd_metrics:
    enabled: true

# WORKER & CLUSTER VISIBILITY (DEPLOYMENT)
# Scrapes the Prometheus Stack Services for Cluster-wide data.
inputPluginsForDeployment:

  # Scrapes the 'kube-state-metrics' service provided by the Prom Stack.
  # Maps Pods -> Nodes so VCF Ops can build the topology.
  kube_state_metrics:
    enabled: true

  # Scrapes the 'prometheus-node-exporter' service.
  # Gives VCF Ops the hardware stats for all worker nodes.
  node_exporter_metrics:
    enabled: true
```

Subsequently, we update the Telegraf install:

```
# Find the Telegraf package and namespace
read -r NAMESPACE PKG_NAME <<< \
  $(kubectl get pkgi -A --no-headers | awk '/telegraf/{print $1, $2}')

# Update Telegraf
vcf package installed update "$PKG_NAME" \
  --values-file telegraf-data-values.yaml \
```

```
--namespace "$NAMESPACE"
```

The screenshot below shows the VCF Operations dashboard with data collected by Telegraf for this cluster.

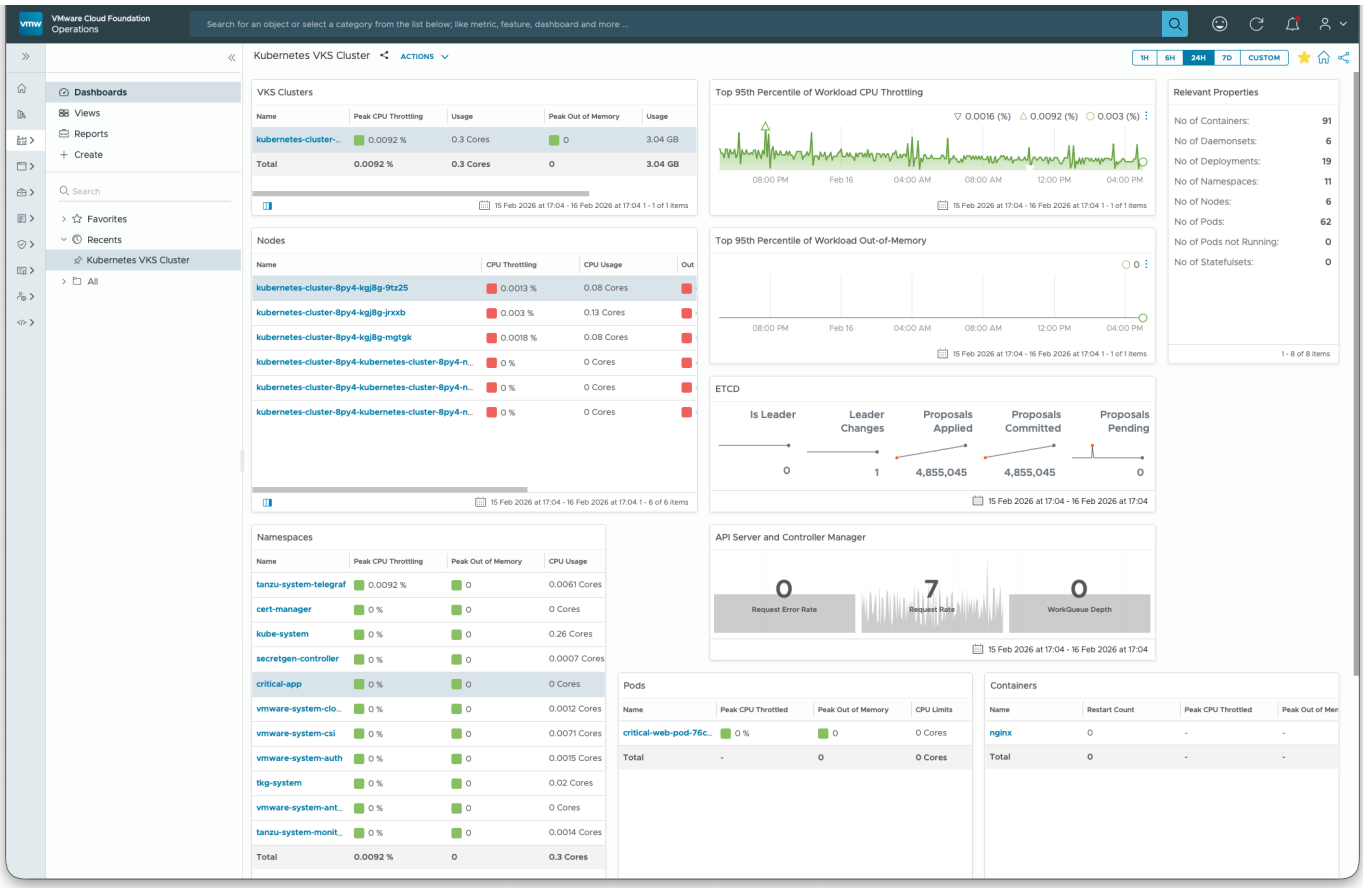

Figure 16: Kubernetes VKS Cluster overview dashboard within VMware Cloud Foundation Operations

Noisy Neighbor Scenario

Here, we expand on our statement above: that VCF Operations can be used to correlate Kubernetes workloads with *traditional* (non-Kubernetes) workloads running on the same cluster. While Kubernetes is highly efficient at managing resources within its own cluster boundaries, it operates with a fundamental blind spot: the scheduler assumes the resources presented to its worker nodes are *guaranteed*. When a Kubernetes cluster shares physical infrastructure with other workloads (such as VMs), it becomes susceptible to the "noisy neighbor" effect: where VM workloads saturate the hypervisor, starving the Kubernetes nodes of physical compute cycles.

To demonstrate how VCF Operations bridges this observability gap, we deployed a highly sensitive, CPU-bound workload pod (details of this pod can be found in Appendix C). The relevant VKS node VM was then *pinned* to the physical host. We then deployed several VMs which would generate CPU load using the 'stress' package. Finally we configured a strict DRS rule (VM-Host affinity) to keep the VKS VM and VM workloads on the same host to force a sustained contention scenario. Note: under usual circumstances, DRS would spread the workload by live migrating VMs, via vMotion, across the hosts.

First, an Ubuntu cloud VM was configured; here we used govc (see https://github.com/vmware/govmomi/releases) to configure and deploy the VMs. We are then able to clone this VM and define a script to run over all clones

```
# Clone the Ubuntu image
seq 1 10 | xargs -P0 -I {} govc vm.clone -vm ubuntu ubuntu-clone{}


# Define the run script
cat << EOF  > run_all.sh
#!/bin/bash
export input=\$1
govc find -type m -name 'ubuntu-clone*' | \
  xargs -P0 -I \
  '{}' bash -c 'ssh -o "StrictHostKeyChecking=no" ubuntu@\$(govc vm.ip {}) "\$input"'
EOF
```

The Kubernetes workload pod ran in two phases. In the first phase, it calibrated results to form a baseline and then subsequently ran in a loop measuring and assessing the time taken for the calculation to take place

```
kubectl -n critical-app logs -l app=victim -f

[Phase 1] Calibrating to underlying hardware...
✅ HEALTHY:  ▏          1.35s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.37s
✅ HEALTHY:  ▏          1.37s
✅ HEALTHY:  ▏          1.37s
✅ HEALTHY:  ▏        1.35s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.37s
✅ HEALTHY:  ▏          1.37s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏        1.35s

[Phase 1 Complete] Hardware Profile Locked:
  - Baseline:      1.36s per batch
  - Alert Trigger: >1.63s
==================================================

[Phase 2] Executing Continuous Processing...
✅ HEALTHY:  ▏        1.34s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏        1.35s
✅ HEALTHY:  ▏          1.36s
✅ HEALTHY:  ▏          1.35s
✅ HEALTHY:  ▏        1.35s
```

Once the workload was running, the stress test was initiated

```
# Start the stress test
./run_all.sh 'stress --cpu 16 --timeout 600s'

stress: info: [4994] dispatching hogs: 16 cpu, 0 io, 0 vm, 0 hdd
stress: info: [4600] dispatching hogs: 16 cpu, 0 io, 0 vm, 0 hdd
stress: info: [4567] dispatching hogs: 16 cpu, 0 io, 0 vm, 0 hdd
stress: info: [4653] dispatching hogs: 16 cpu, 0 io, 0 vm, 0 hdd
stress: info: [4996] dispatching hogs: 16 cpu, 0 io, 0 vm, 0 hdd
stress: info: [4648] dispatching hogs: 16 cpu, 0 io, 0 vm, 0 hdd
stress: info: [4606] dispatching hogs: 16 cpu, 0 io, 0 vm, 0 hdd
stress: info: [5026] dispatching hogs: 16 cpu, 0 io, 0 vm, 0 hdd
stress: info: [4660] dispatching hogs: 16 cpu, 0 io, 0 vm, 0 hdd
stress: info: [4586] dispatching hogs: 16 cpu, 0 io, 0 vm, 0 hdd
```

The effects of the CPU contention can then be seen on the workload pod

```
✅ HEALTHY:  1.36s
✅ HEALTHY:  1.38s
🚨 CRITICAL: 1.84s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.28s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.69s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.67s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.66s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.67s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.67s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.66s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.64s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.64s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.65s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.66s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.67s (CPU STEAL LIKELY)
🚨 CRITICAL: 2.67s (CPU STEAL LIKELY)
```

Using the VCF Operations dashboard, we analyzed the environment across three distinct layers:

1. **The 'Victim' (Kubernetes Worker Node):** By querying the vCenter adapter's view of the worker node VM, we observed a spike in *CPU Ready (%).* This metric explicitly confirmed that the node's guest OS was ready to execute instructions but was being *forced to wait* by the hypervisor.
2. **The Root Cause (ESX Host):** Correlating the node's CPU Ready metric with its underlying physical host revealed a simultaneous, sustained spike in *Host CPU Contention (%)* and *Usage (%).*
3. **The Culprits (Topology Mapping):** Using VCF Operations' object relationship mapping, we could easily identify the ten legacy VMs consuming the host's capacity, thus proving that the Kubernetes degradation was caused by an external infrastructure event.

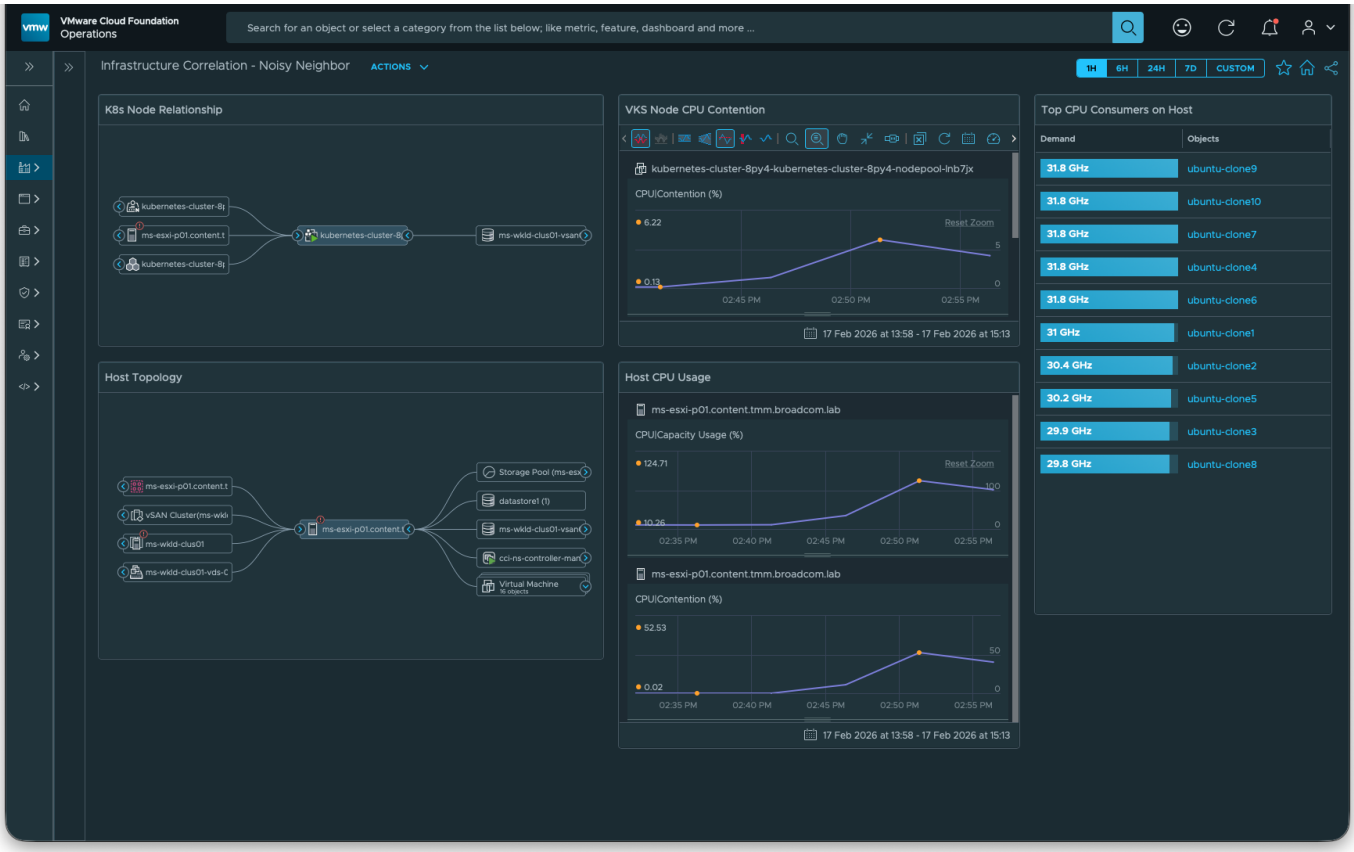

Figure 17: Identifying a "noisy neighbor" issue by correlating VKS node CPU contention with overall host CPU usage and top consumers

Logging

Effective observability requires more than metrics: logs provide the detailed, time-ordered context needed to understand deployments, failures, and user-impacting incidents. In Kubernetes, where workloads are dynamic and pods are frequently rescheduled, log data must be captured and aggregated outside the workload lifecycle to remain searchable and operationally useful.

To support this model, **Grafana Loki** is deployed as the centralized log store for Kubernetes logs, with **Fluent Bit** running as a lightweight node-level collector on each worker node. Fluent Bit tails container stdout/stderr at the node, enriches each record with Kubernetes metadata (for example namespace, pod, container, and selected labels), and forwards the stream into Loki. Loki then serves as the primary backend for Grafana-based log exploration, enabling fast, label-driven queries and drill-down troubleshooting close to the workload.

In parallel, Fluent Bit can also forward the same enriched log stream into **VCF Operations for Logs**, allowing Kubernetes logs to be analyzed alongside the wider infrastructure log estate without introducing a second collection agent. This dual-destination approach preserves a clear separation of concerns: Loki with Grafana remains the Kubernetes-native workflow for application and platform teams, while VCF Operations for Logs provides the operations-facing view for broader correlation and cross-domain troubleshooting.

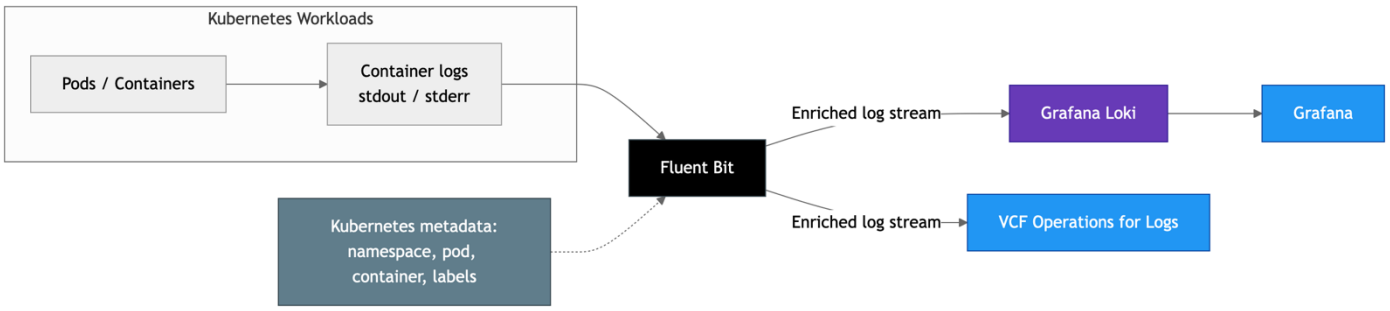

Figure 18: Kubernetes logging and enrichment architecture using Fluent Bit

Loki

We deploy Loki in *Distributed* (microservices) mode, backed by an S3-compatible object store (we use MinIO, see Appendix D for details). Loki's ingestion and query path components (distributor, ingester, querier, query-frontend, and index-gateway) run as separate services, allowing the system to scale horizontally and to isolate read and write workloads.

Note: this design maps directly to Grafana Enterprise Logs (GEL), Grafana's commercial offering based on Loki. GEL provides enterprise support and additional operational and governance capabilities while remaining compatible with the Loki ingestion and query model.

The manifest below describes the configuration:

```
# loki-values.yaml

# Define image registry to avoid dockerhub limits
global:
  image:
    registry: my.internal.registry

# Loki microservices mode
deploymentMode: Distributed

loki:
  # Loki authorization is advised in production
  # Typically via external proxy (OIDC/OAuth2)
```

```
  auth_enabled: false

  commonConfig:
    replication_factor: 2

  storage:
    # For 'microservices' (production) shared
    # storage, typically an S3 datastore, is required
    type: s3
    bucketNames:
      chunks: loki-chunks
      ruler: loki-ruler
      admin: loki-admin
    s3:
      endpoint: http://minio.minio.svc.cluster.local:9000
      s3ForcePathStyle: true
      insecure: true

      # Credentials are injected via env vars from a Secret
      accessKeyId:
      secretAccessKey:

  limits_config:
    allow_structured_metadata: true
    volume_enabled: true

  ruler:
    enable_api: true

# Inject MinIO credentials into all Loki pods
extraEnvFrom:
  - secretRef:
      name: loki-s3-creds

# Disable non-distributed deployment modes
singleBinary: { replicas: 0 }
backend: { replicas: 0 }
read: { replicas: 0 }
write: { replicas: 0 }

# Microservice components
ingester: { replicas: 2, maxUnavailable: 1 }
querier: { replicas: 2, maxUnavailable: 1 }
distributor: { replicas: 2, maxUnavailable: 1 }
queryFrontend: { replicas: 1 }
queryScheduler: { replicas: 1 }
indexGateway: { replicas: 1 }

gateway:
  enabled: true
```

This can then be installed with Helm in the usual way

```
# Add Grafana charts repo
helm repo add grafana https://grafana.github.io/helm-charts


# Install/upgrade Loki
helm upgrade --install loki grafana/loki \
  -n loki --create-namespace \
  -f loki-values.yaml \
  --wait --timeout 20m
```

Fluent-bit

Fluent Bit provides the log shipping layer that connects Kubernetes workloads to Loki. Deployed as a DaemonSet, it tails container stdout/stderr on each node, enriches entries with Kubernetes context (namespace, pod, container and selected labels), and forwards the resulting stream to Loki over HTTP. This ensures logs arrive with consistent metadata and label structure, enabling efficient indexing in Loki.

First, we ensure that the Fluent-bit VKS addon package is healthy

```
# Confirm fluent-bit package is installed
kubectl -n vmware-system-tkg describe \
  $(kubectl get pkgi -A -o name | grep fluent-bit) | grep -A5 'Status'

Status:
  Conditions:
    Status:                True
    Type:                  ReconcileSucceeded
  Friendly Description:    Reconcile succeeded
  Last Attempted Version:  4.0.8+vmware.1-vks.1
  Observed Generation:     1
  Version:                 4.0.8+vmware.1-vks.1


# Check runtime
kubectl get pods -A -l app=fluent-bit

NAMESPACE              NAME                 READY   STATUS    RESTARTS   AGE
tanzu-system-logging   fluent-bit-f4bqt     1/1     Running   0          8m58s
tanzu-system-logging   fluent-bit-hzj9r     1/1     Running   0          9m5s
tanzu-system-logging   fluent-bit-jwj5s     1/1     Running   0          9m8s
tanzu-system-logging   fluent-bit-p9zxs     1/1     Running   0          8m56s
tanzu-system-logging   fluent-bit-q67n5     1/1     Running   0          9m2s
tanzu-system-logging   fluent-bit-vsxjv     1/1     Running   0          9m
```

Next, we configure and customize Fluent-bit (for details see https://techdocs.broadcom.com/us/en/vmware-cis/vcf/vsphere-supervisor-services-and-standalone-components/latest/managing-vsphere-kuberenetes-service-clusters-and-workloads/installing-standard-packages-on-tkg-service-clusters/standard-package-reference/fluent-bit-package-reference.html)

```
# Create a secret with VCF/logs Certificate
kubectl -n tanzu-system-logging create secret generic tls-ca-cert \
  --from-file=tls_http.crt=vcf-ops-logs-ca.pem


# Create the Fluent Bit values file
# Outputs to both Loki & VCF Ops/Logs
VCF_LOGS_HOST=ops-logs.lab
VKSCLUSTER=vks-kubernetes-cluster

cat << EOF > fluent-bit-values.yaml
fluent_bit:
  config:
    outputs: |
      [OUTPUT]
        Name          http
        Match         *
        Host          ${VCF_LOGS_HOST}
        Port          9543
        URI           api/v2/events
        Format        json
        Header        Content-Type application/json
        tls           on
        tls.debug     4
        tls.verify    on
```

```
        tls.ca_file   /etc/ssl/certs/tls_http.crt
        Retry_Limit   False

      [OUTPUT]
        Name          loki
        Match         kube.*
        Host          cluster-loki-gateway.loki.svc.cluster.local
        Port          80
        URI           /loki/api/v1/push
        labels        job=fluent-bit,source=vks,cluster=${VKSCLUSTER}
        label_keys    \$namespace_name,\$pod_name,\$container_name,\$host,\$stream

  daemonset:
    secretName: tls-ca-cert

namespace: tanzu-system-logging
EOF
```

We update the Fluent-bit configuration by applying the configuration manifest to the package install

```
# Switch to the supervisor context
vcf context use <supervisor namespace>


# Update the fluent-bit config
vcf addon install update fluent-bit \
  --cluster-name $VKSCLUSTER \
  -f fluent-bit-values.yaml
```

Switching back to the VKS cluster context, we restart the daemonset and ensure the Fluent-bit pods are running

```
# Restart Fluent-bit daemonset to reconcile changes
kubectl -n tanzu-system-logging rollout restart daemonset fluent-bit


# Ensure pods are healthy
kubectl -n tanzu-system-logging get pods

NAME                   READY   STATUS    RESTARTS   AGE
pod/fluent-bit-6jqjj   1/1     Running   0          9m43s
pod/fluent-bit-f66x5   1/1     Running   0          9m49s
pod/fluent-bit-gfb9t   1/1     Running   0          9m54s
pod/fluent-bit-mqphg   1/1     Running   0          9m51s
pod/fluent-bit-qg96p   1/1     Running   0          9m47s
pod/fluent-bit-tn7kg   1/1     Running   0          9m45s
```

Grafana Log Drilldown

Next, we add Loki as a data source in Grafana and use it to query and explore the logs collected by Fluent Bit:

- Navigate to Connections → Data sources
- Click Add data source and select Loki
- Configure the data source:
  - **URL**: enter the Loki gateway in the form:
    http://<loki-gateway>.<namespace>.svc.cluster.local
    (e.g. http://cluster-loki-gateway.loki.svc.cluster.local)
  - **HTTP headers** (optional): add a custom header for multi-tenancy:
    Header: *X-Scope-OrgID*
    Value: *platform*
- Click Save & test

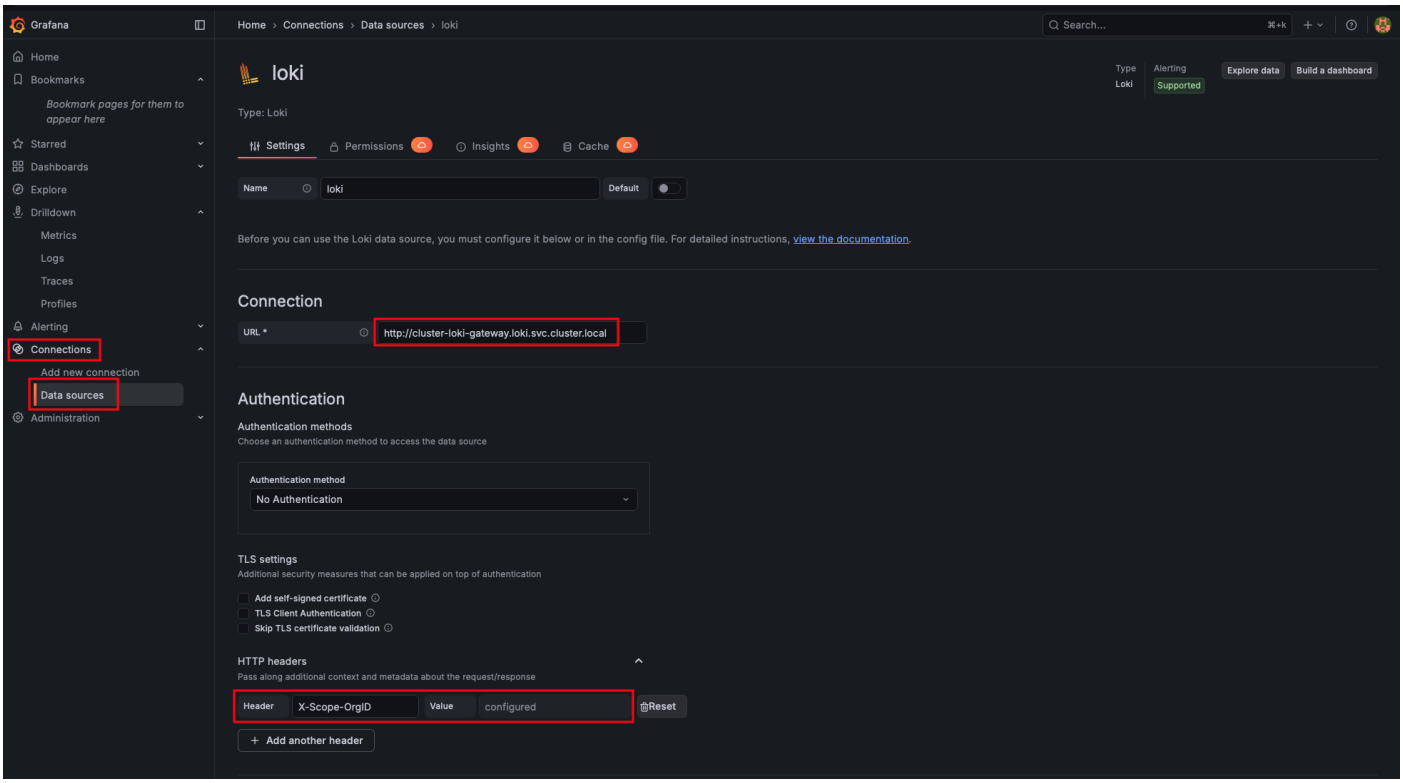

Figure 19: Configuring Loki as a data source in Grafana, highlighting the internal Kubernetes connection URL and required HTTP headers

To view collected logs, open Drilldown → Logs

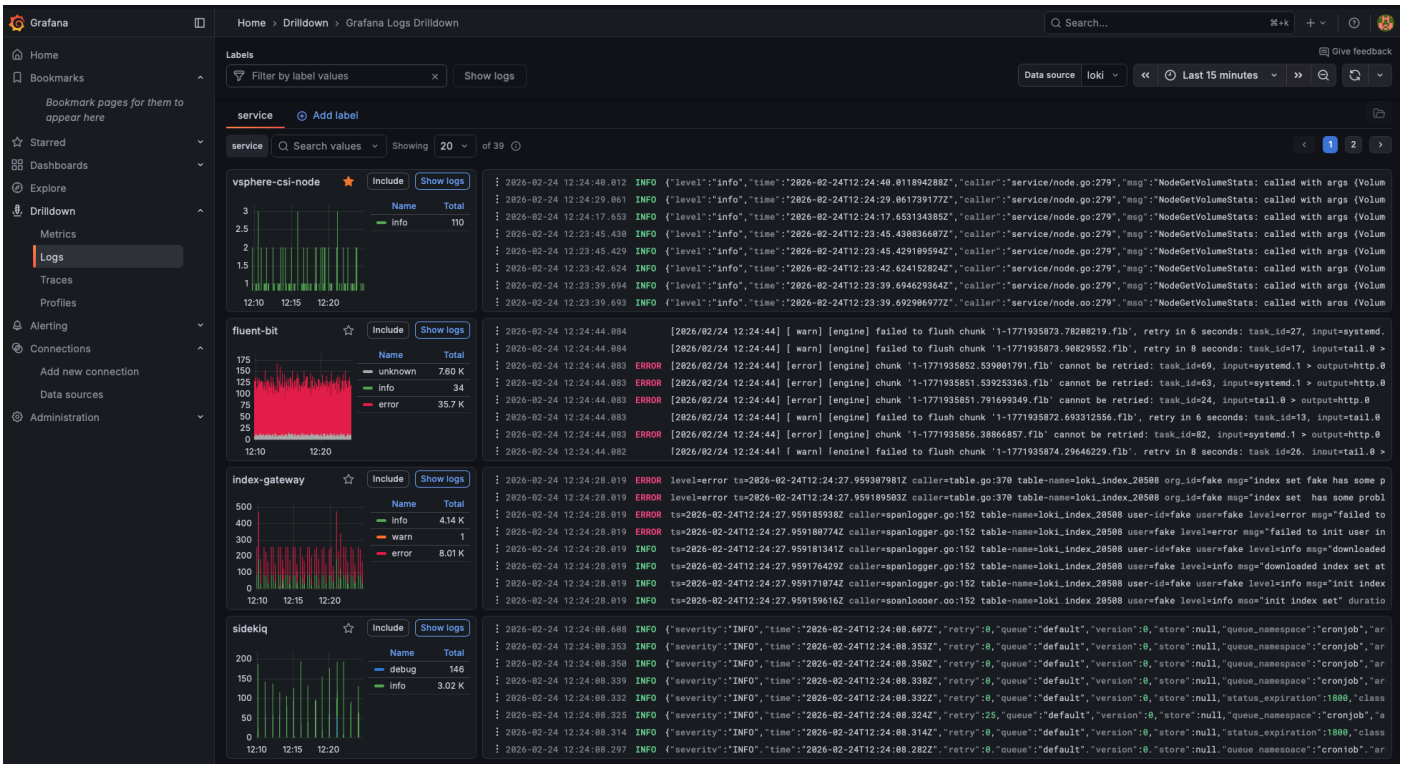

Figure 20: Grafana Logs Drilldown dashboard, displaying aggregated log streams, volume histograms, and log severity levels organized by service

VCF Operations for Logs

To make Kubernetes logs available beyond the Loki/Grafana workflow, we also onboard them into VCF Operations for Logs. Using Fluent Bit as the shared collection layer, logs are harvested from each node, tagged with Kubernetes metadata, and shipped onward to VCF Operations for indexing and correlation with broader VCF telemetry. This enables consistent operational troubleshooting across infrastructure and workloads, while keeping Loki optimized for label-driven queries and visualization in Grafana.

We can see the data ingested by navigating to ‘Explore Logs’ and filtering for ‘container’. This shows a list of events and a graph of the count

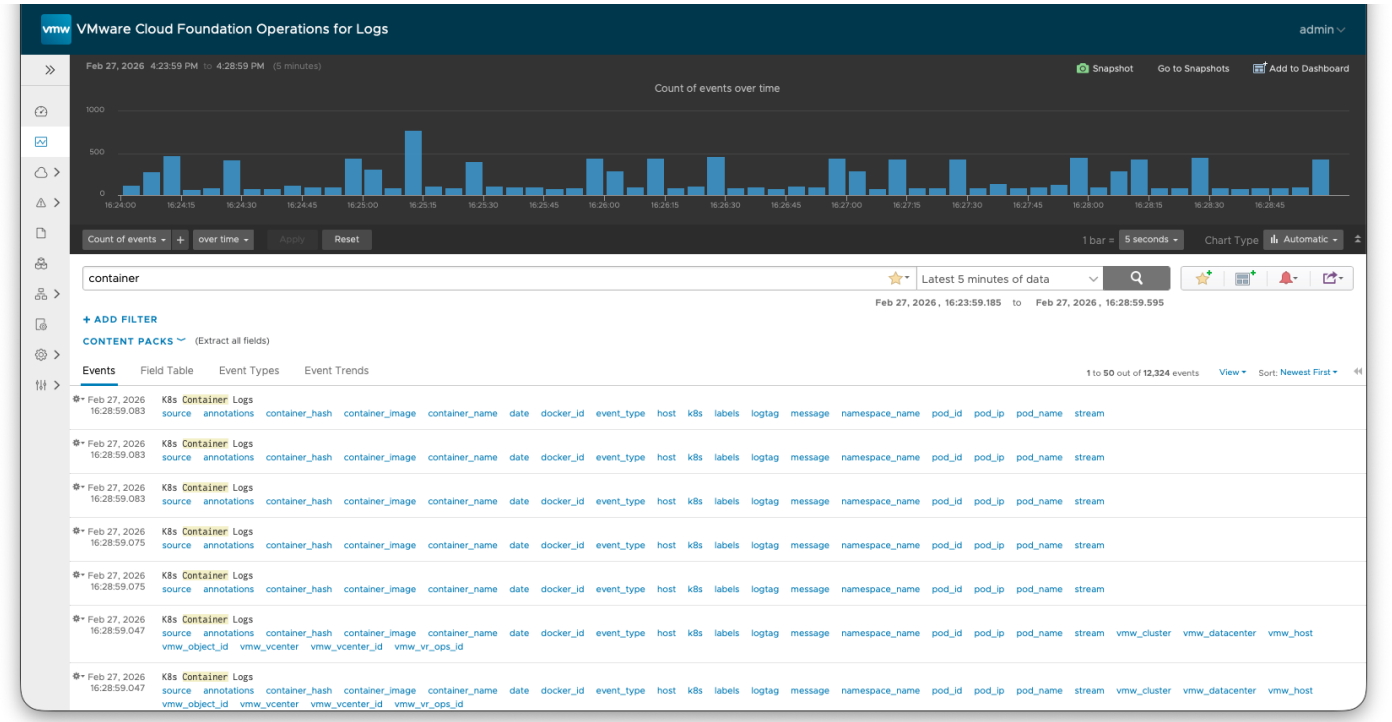

Figure 21: Log analysis and search interface in VMware Cloud Foundation Operations for Logs

The filters can be refined to focus the analysis. In this example, the Event Trends view is used to visualize the unique count of pods over time, grouped by host.

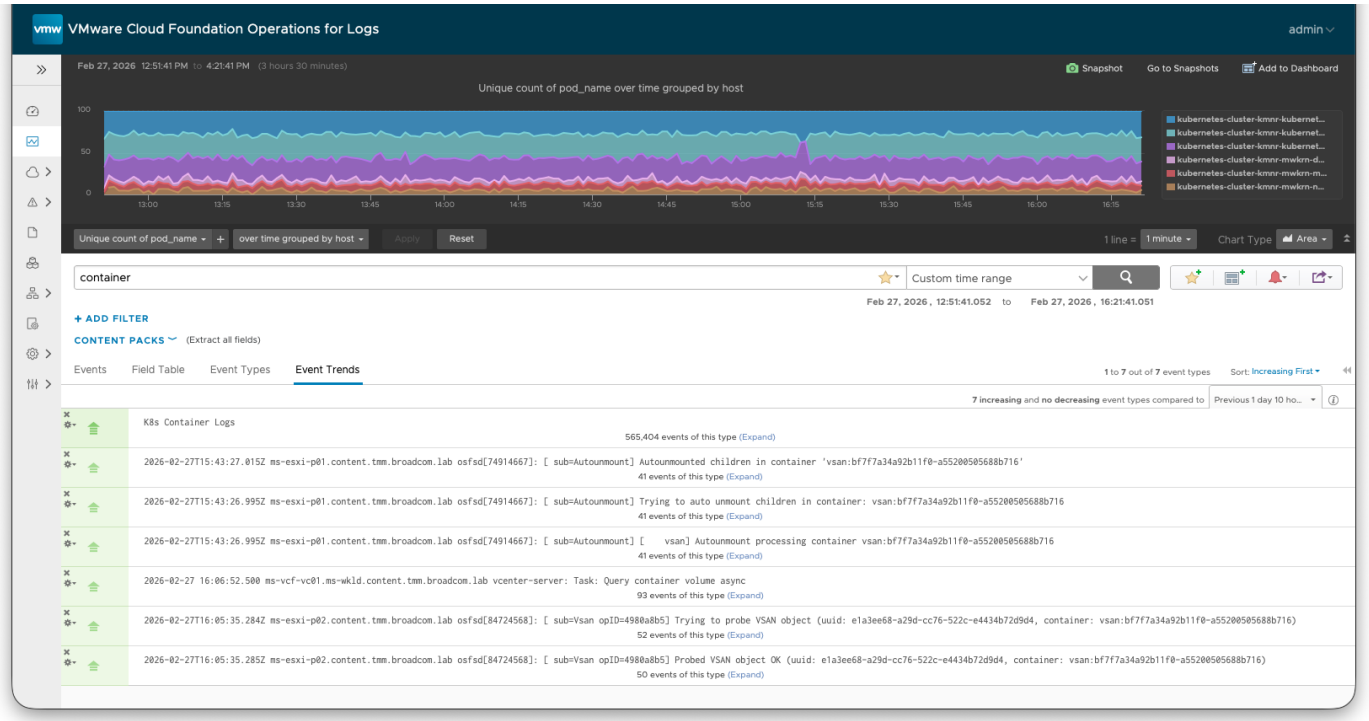

Figure 22: Event Trends analysis interface in VMware Cloud Foundation Operations for Logs

Tracing

Distributed tracing extends observability beyond metrics and logs by showing how requests move across services, revealing latency, service dependencies, and points of failure across application flows. In Kubernetes, where requests often traverse multiple ephemeral services and infrastructure boundaries, trace data must be captured and correlated across the full execution path to remain useful for performance analysis and end-to-end troubleshooting.

To support this model, we leverage **OpenTelemetry** for distributed tracing. Workloads emit trace data using OTLP to an in-cluster OpenTelemetry Collector, which provides a consistent point for receiving, processing, and forwarding traces to the selected backend. This OTLP data is then ingested by **Jaeger**, which provides the user interface for exploring end-to-end request paths, service dependencies, and latency across the application.

Additionally, Jaeger relies on a persistent storage backend to retain and index trace data. For the purposes of this whitepaper and for brevity, we configure Jaeger to connect to **OpenSearch** over HTTPS using insecure certificate verification. In production, OpenSearch should be configured with trusted certificates for its REST and transport layers, and Jaeger should validate the OpenSearch certificate chain and hostname against a trusted CA. For more information, consult the OpenSearch TLS configuration documentation and Jaeger's security guidance.

To demonstrate tracing, the **OpenTelemetry Demo** is configured to emit OTLP trace data to the in-cluster collector, providing a representative multi-service workload through which end-to-end request flows can be observed. This allows trace collection, forwarding, and visualization to be exercised against a realistic application path, showing how requests traverse multiple services and how latency and failures can be isolated across the stack. See Appendix E for the specific OpenTelemetry Demo configuration used in this implementation. Note that the demo

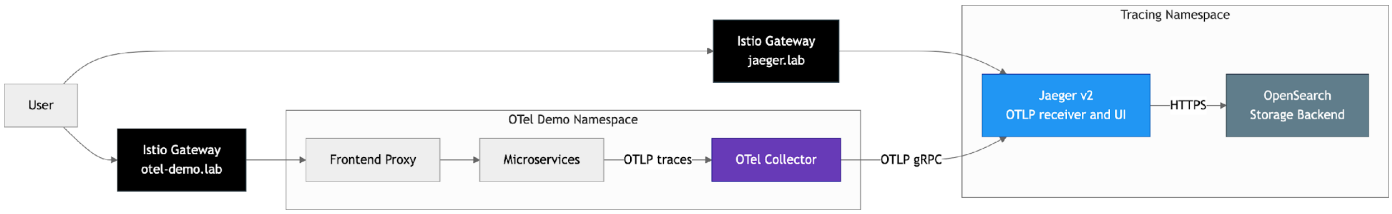

Figure 23: Distributed tracing architecture using OpenTelemetry, Jaeger v2, and OpenSearch

OpenSearch

Here, for our purposes, we deploy OpenSearch as a single node, with basic authentication. Note that this configuration uses internal certificates, production environments should be configured with trusted certificates.

The manifest below describes the configuration:

```
# opensearch-values.yaml

global:
  dockerRegistry: public.ecr.aws
clusterName: jaeger-traces
nodeGroup: master
singleNode: true
replicas: 1
extraEnvs:
  - name: OPENSEARCH_INITIAL_ADMIN_PASSWORD
    value: "C0rrugated!Trace#2026"
opensearchJavaOpts: -Xms1g -Xmx1g
resources:
  requests:
    cpu: 500m
    memory: 2Gi
  limits:
    cpu: '2'
    memory: 2Gi
volumeClaimTemplate:
  accessModes:
    - ReadWriteOnce
  storageClassName: <your-vsan-storageclass>
  resources:
    requests:
      storage: 100Gi
service:
  type: ClusterIP
sysctlInit:
  enabled: false
  image: docker/library/busybox
persistence:
  image: docker/library/busybox
```

As before, this can then be installed with Helm in the usual way

```
# Create ns for OpenSearch
NS=tracing
kubectl create ns $NS
kubectl label --overwrite ns $NS \
  pod-security.kubernetes.io/enforce=baseline


# Add OpenSearch charts repo
helm repo add opensearch https://opensearch-project.github.io/helm-charts
```

```
# Install/upgrade Opensearch
helm upgrade --install opensearch opensearch/opensearch \
  --namespace tracing \
  --version 3.5.0 \
  -f opensearch-values.yaml \
  --wait --timeout 20m
```

Jaeger

Here we deploy Jaeger v2 using the **OpenTelemetry Operator**. This provides a Kubernetes-native method for managing the tracing backend alongside the wider observability stack. In practice, the operator is used to instantiate and manage the Jaeger deployment, while also providing a consistent framework for configuration, lifecycle management, and integration with OpenTelemetry-based trace collection.

```
# Create ns for OpenTel Operator
NS=opentelemetry-operator-system
kubectl create ns $NS
kubectl label --overwrite ns $NS \
  pod-security.kubernetes.io/enforce=baseline

# Install the OpenTel Operator
kubectl apply -f \
  https://github.com/open-telemetry/opentelemetry-operator/releases/latest/download/\
  opentelemetry-operator.yaml
```

The operator is then used to deploy Jaeger v2 with the configuration shown below. Here, Jaeger uses OpenSearch as its persistent storage backend, following the simplified HTTPS configuration outlined earlier. For production deployments, OpenSearch should be configured with trusted certificates for its REST and transport layers, and Jaeger should validate the OpenSearch certificate chain and hostname against a trusted CA.

```
kubectl apply -f - <<EOF
apiVersion: opentelemetry.io/v1beta1
kind: OpenTelemetryCollector
metadata:
  name: jaeger
  namespace: tracing
spec:
  # Run Jaeger v2 as a single in-cluster deployment
  mode: deployment
  replicas: 1
  image: jaegertracing/jaeger:2.16.0

  # Expose the Jaeger UI and OTLP ingest endpoints
  ports:
    - name: jaeger-ui
      port: 16686
    - name: otlp-grpc
      port: 4317
    - name: otlp-http
      port: 4318

  config:
    service:
      # Enable storage, query/UI, and health endpoints
      extensions: [jaeger_storage, jaeger_query, healthcheckv2]

      pipelines:
        traces:
          # Receive OTLP traces and persist them via the Jaeger storage exporter
          receivers: [otlp]
```

```
        processors: [batch]
        exporters: [jaeger_storage_exporter]

    telemetry:
      resource:
        service.name: jaeger
      metrics:
        # Expose basic Prometheus metrics for the collector itself
        level: basic
        readers:
          - pull:
              exporter:
                prometheus:
                  host: 0.0.0.0
                  port: 8888
      logs:
        level: info

  extensions:
    # Health endpoint used for readiness / liveness checks
    healthcheckv2:
      use_v2: true
      http: {}

    # Jaeger query service and UI
    jaeger_query:
      http:
        endpoint: 0.0.0.0:16686
      storage:
        traces: opensearch_store

    # OpenSearch backend used for persistent trace storage
    jaeger_storage:
      backends:
        opensearch_store:
          opensearch:
            server_urls:
              - https://opensearch-cluster-master.tracing.svc.cluster.local:9200
            tls:
              # Simplified for this implementation --
              # replace with full cert validation in production
              insecure_skip_verify: true
            auth:
              basic:
                username: admin
                password: "C0rrugated!Trace#2026"
            indices:
              index_prefix: jaeger
              spans:
                shards: 1
                replicas: 0
              services:
                shards: 1
                replicas: 0
              dependencies:
                shards: 1
                replicas: 0
              sampling:
                shards: 1
                replicas: 0

  receivers:
    otlp:
      # Accept OTLP traces over both gRPC and HTTP
      protocols:
```

```
          grpc:
            endpoint: 0.0.0.0:4317
          http:
            endpoint: 0.0.0.0:4318

    processors:
      # Batch trace data before writing to storage
      batch: {}

    exporters:
      # Persist traces into the configured Jaeger storage backend
      jaeger_storage_exporter:
        trace_storage: opensearch_store
EOF
```

Finally, as before, we create an ingress gateway via Istio:

```
# Label the monitoring namespace so the Gateway trusts it
kubectl label ns tracing access=gateway-trusted --overwrite


# Apply the gateway and routes
kubectl apply -f - <<EOF
apiVersion: gateway.networking.k8s.io/v1
kind: Gateway
metadata:
  name: jaeger-gateway
  namespace: istio-ingress
spec:
  gatewayClassName: istio
  listeners:
    - name: https
      protocol: HTTPS
      port: 443
      hostname: jaeger.lab
      tls:
        mode: Terminate
        certificateRefs:
          - name: monitoring-gateway-tls
      allowedRoutes:
        namespaces:
          from: Selector
          selector:
            matchLabels:
              access: gateway-trusted
---
apiVersion: gateway.networking.k8s.io/v1
kind: HTTPRoute
metadata:
  name: jaeger
  namespace: tracing
spec:
  parentRefs:
    - name: jaeger-gateway
      namespace: istio-ingress
  hostnames:
    - jaeger.lab
  rules:
    - matches:
        - path:
            type: PathPrefix
            value: /
      backendRefs:
        - name: jaeger-collector
```

```
        port: 16686
EOF
```

As explained earlier, we make use of the OpenTelemetry demo to generate OTLP data for tracing (see Appendix E). After some time, we can see the traces from the demo populated in Jaeger

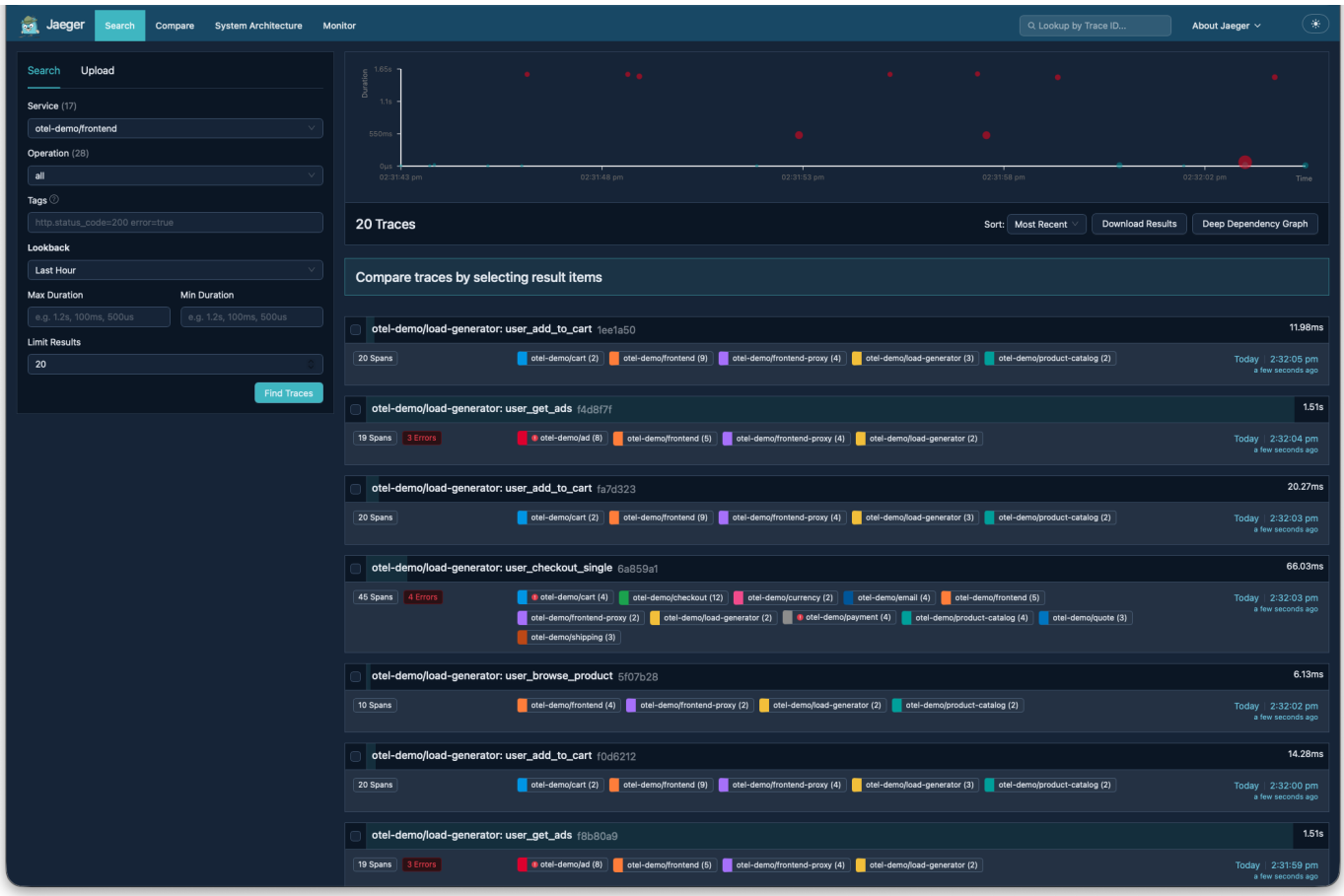

Figure 24: Jaeger UI displaying search results for distributed traces

Below, we observe the end-to-end trace for an add-to-cart transaction. Here we see the individual spans that make up the full request path across the application.

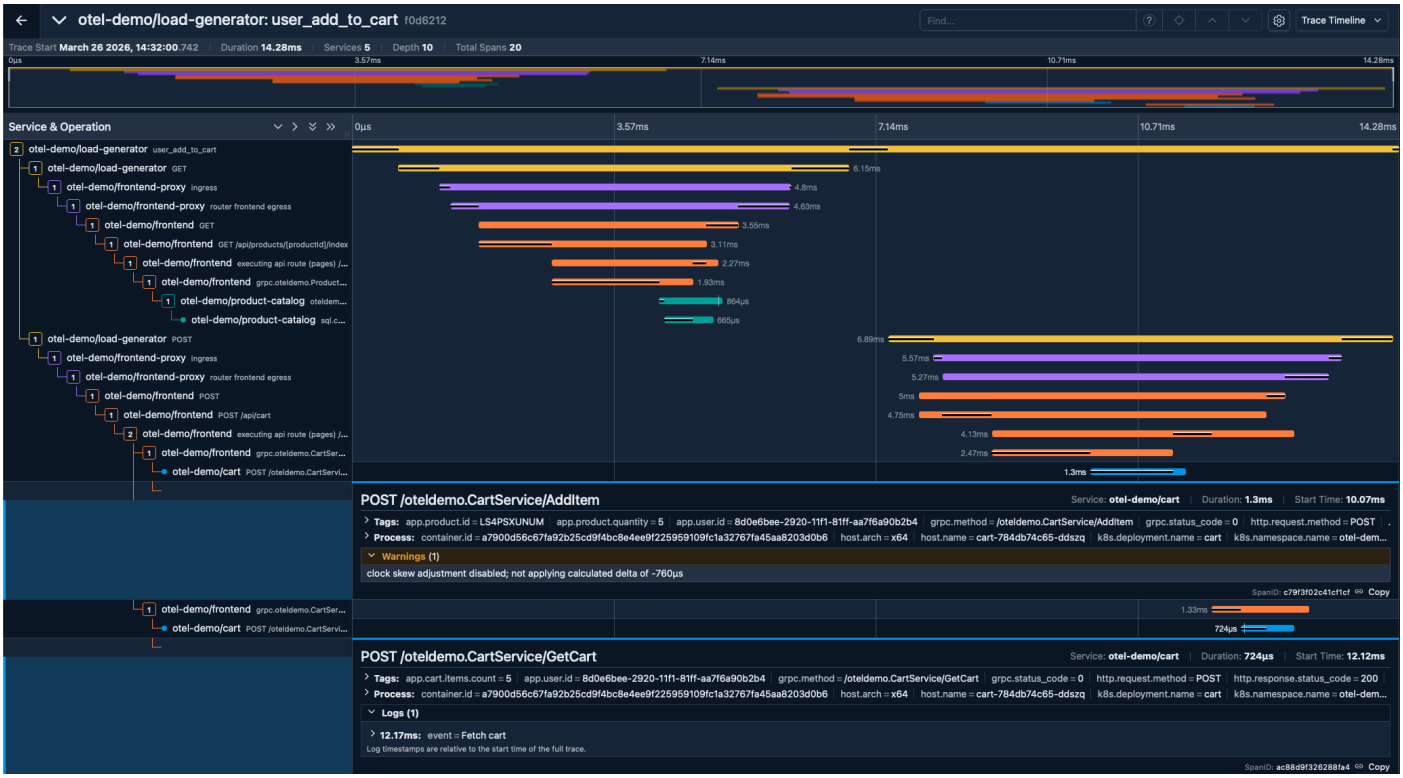

Figure 25: Jaeger UI displaying a detailed trace timeline and individual span information

The trace shown above represents a single end-to-end `'user_add_to_cart'` transaction. In Jaeger, the full transaction is recorded as a trace, while each individual operation within that request path is recorded as a span. These spans are linked through parent-child relationships, allowing the full execution path to be reconstructed across services.

In this example, the trace spans five services and twenty spans over a total duration of approximately 15 ms. The request passes through the *frontend-proxy* and *frontend* and then fans out into downstream services including *product-catalog* and *cart*. Jaeger's timeline shows the duration and nesting of each span, making it possible to see both the request hierarchy and where time is spent. The selected spans also expose technical metadata such as the HTTP and gRPC methods, status code, Kubernetes namespace and deployment, hostname, and container context, providing the detail needed to isolate latency or failure within the transaction.

Conclusion

This paper has shown that a practical, enterprise-grade observability model for Kubernetes can be built on VMware Cloud Foundation with vSphere Kubernetes Service by combining open-source, Kubernetes-native tooling with platform-level infrastructure insight. Prometheus and Grafana provide real-time visibility into cluster and service behavior, Loki and Fluent Bit deliver Kubernetes-native log aggregation and drill-down troubleshooting, and OpenTelemetry with Jaeger extends that view into end-to-end request flows across distributed applications. Together, these capabilities form a coherent observability stack that addresses the operational realities of dynamic, multi-service Kubernetes environments.

Equally important, the implementation demonstrates that observability in Kubernetes cannot be treated as an application-only concern. While metrics, logs, and traces provide deep visibility within the cluster, meaningful root-cause analysis often depends on understanding the infrastructure beneath it. By integrating VCF Operations alongside the Kubernetes-native toolchain, the solution is able to correlate workload behavior with host, storage, and network conditions, bridging the gap between platform operations and application operations. The noisy-neighbor scenario illustrates this clearly: Kubernetes telemetry alone can reveal degraded behavior, but full-stack correlation is what exposes the underlying cause.

The resulting architecture is therefore not simply a collection of monitoring tools, but a layered observability model aligned to the realities of enterprise platform operations. It combines open standards such as OTLP and OpenTelemetry with widely adopted open-source projects, while remaining grounded in the operational controls, governance, and lifecycle benefits provided by VCF and VKS. Although several implementation choices in this paper are intentionally simplified for demonstration purposes, the design patterns remain directly applicable to production environments. In that sense, the architecture provides both a validated implementation example and a reference point for organizations seeking to establish consistent, supportable observability across cloud-native applications running on private cloud infrastructure.

Appendix A: Example Client VM Configuration & Tooling

Here, we used an Ubuntu Jammy cloud VM. 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 install -m 755 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 set shorthand and autocomplete for kubectl

```
# 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 at 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
```

Appendix B: Cert-manager with ADCS Integration

The following steps outline the integration of cert-manager with Active Directory Certificate Services (ADCS); this allows certificate requests to be fulfilled by a Windows Certificate Authority (CA) server:

```
### 1. Prerequisites: Namespace & Credentials
kubectl create ns adcs-issuer
kubectl label --overwrite ns adcs-issuer pod-security.kubernetes.io/enforce=baseline

# Create secret containing user/pass for windows CA
# Ensure account has the correct rights to 'enroll' certificates (see below)
kubectl -n adcs-issuer create secret generic adcs-issuer-credentials \
  --from-literal=username='showcase\[your username]' \
  --from-literal=password='[your password]'

### 2. Install the ADCS Issuer Webhook
helm repo add djkormo-adcs-issuer https://djkormo.github.io/adcs-issuer/
helm repo update

# Inspect available versions
helm search repo djkormo-adcs-issuer/adcs-issuer --versions

# Pull default values for a version you want to pin
helm show values djkormo-adcs-issuer/adcs-issuer --version 2.2.1 > adcs-values.yaml

# Install (default values in adcs-values.yaml will server most environments)
helm upgrade --install adcs-issuer djkormo-adcs-issuer/adcs-issuer \
  --namespace adcs-issuer \
  --create-namespace \
  --version 2.2.1 \
  --values adcs-values.yaml

### 3. Configure the Cluster Issuer
# Create clusteradcsissuer: first obtain a base64 encoded certificate
# From the Windows CA & encode with 'base64 -w0 certificate.cer'
# Then add to the key 'caBundle'
# NB: ensure account used has permissions to use the "webserver" template
kubectl apply -f - <<EOF
apiVersion: adcs.certmanager.csf.nokia.com/v1
kind: ClusterAdcsIssuer
metadata:
  name: adcs-cluster-issuer
spec:
  caBundle:
  credentialsRef:
    name: adcs-issuer-credentials
  statusCheckInterval: 5m
  retryInterval: 5m
  url: "https://lvn-sc-ca01.showcase.tmm.broadcom.lab/certsrv"
  templateName: "WebServer"
EOF

### 4. Request a Certificate
# Here we request a certificate for an istio ingress gateway
# and store as the secret 'monitoring-gateway-tls'
kubectl apply -f - <<EOF
apiVersion: cert-manager.io/v1
kind: Certificate
metadata:
  name: monitoring-gateway-cert
```

```
  namespace: istio-ingress
spec:
  secretName: monitoring-gateway-tls
  commonName: "*.lab"
  dnsNames:
    - "*.lab"
  issuerRef:
    group: adcs.certmanager.csf.nokia.com
    kind: ClusterAdcsIssuer
    name: adcs-cluster-issuer
EOF

### 5. Consume the Certificate via Gateway API
# Apply the above to a gateway listener under the 'tls/certifcateRefs' key
# Only allow namespaces with the label ‘gateway-trusted’
kubectl apply -f - <<EOF
kind: Gateway
apiVersion: gateway.networking.k8s.io/v1
metadata:
  name: monitoring-gateway
  namespace: istio-ingress
spec:
  gatewayClassName: istio
  listeners:
  - name: https
    protocol: HTTPS
    port: 443
    hostname: "*.lab"
    tls:
      mode: Terminate
      certificateRefs:
      - name: monitoring-gateway-tls
    allowedRoutes:
      namespaces:
        from: Selector
        selector:
          matchLabels:
            access: gateway-trusted
EOF
```

Appendix C:
CPU Bound Pod for Workload Testing

The manifest below defines a highly sensitive CPU-bound workload (critical-web-app) pinned to a specific Kubernetes worker node. It runs as a single pod that executes a custom Python script engineered to apply predictable, deterministic compute pressure. This script performs continuous SHA-256 hashing in tight, uniform loops (10,000 iterations per work unit). The result is a workload that remains strictly compute-bound with negligible I/O, making changes in wall-clock execution time a useful proxy for CPU scheduling and contention effects.

To reduce ambiguity introduced by Kubernetes scheduling, the pod is assigned a Guaranteed QoS class by setting CPU requests equal to limits (exactly 1 core). This ensures consistent CPU entitlement and reduces variability caused by Linux CFS (Completely Fair Scheduler) time-slicing.

The application runs in two phases:

- **Phase 1: Calibration**: On startup, the pod performs a 5-second warm-up period followed by a 15-second high-frequency sampling window using a monotonic clock. It measures the baseline execution time of its work unit, applies a median filter to reduce normal OS jitter, and dynamically selects a batch size to establish a stable ~1.0-second processing baseline for the underlying node.
- **Phase 2: Monitoring**: The workload then runs continuously, measuring its own batch execution time. If throughput degrades by more than 20% relative to baseline, it logs a visible alert. Crucially, each loop also collects kernel telemetry from /proc/stat and cgroup CPU counters, allowing the script to differentiate likely causes of slowdown:
    - CFS quota enforcement (reported as throttled milliseconds), indicating local container CPU limiting; or
    - hypervisor CPU steal (reported as steal ticks), indicating the VM was runnable but not scheduled on physical CPU due to host-level contention (the “noisy neighbor” condition).

```
# critical-app.yaml

apiVersion: v1
kind: ConfigMap
metadata:
  name: victim-app-script
  namespace: critical-app
data:
  worker.py: |
    import time
    import statistics
    import hashlib

    DATA_PAYLOAD = b"vcf-ops-whitepaper-baseline"
    THR_MS_MIN = 10.0  # Ignore microscopic CFS quota blips

    def unit_of_work():
        data = DATA_PAYLOAD
        for _ in range(10000):
            data = hashlib.sha256(data).digest()
        return data

    def get_os_cpu_stats():
        # /proc/stat columns post-'cpu':
        # 0:user, 1:nice, 2:system, 3:idle,
        # 4:iowait, 5:irq, 6:softirq, 7:steal
        try:
            with open('/proc/stat', 'r') as f:
                line = f.readline().split()
```

```
                if line[0] == 'cpu':
                    ticks = [int(x) for x in line[1:]]
                    if len(ticks) > 7:
                        return sum(ticks), ticks[7]
        except Exception:
            pass
        return 0, 0

    def get_cfs_throttle_usec():
        paths = [
            "/sys/fs/cgroup/cpu.stat",              # v2
            "/sys/fs/cgroup/cpu/cpu.stat",          # v1 (common)
            "/sys/fs/cgroup/cpu,cpuacct/cpu.stat",  # v1 (older)
        ]
        for p in paths:
            try:
                with open(p, "r") as f:
                    for line in f:
                        parts = line.split()
                        if len(parts) == 2:
                            k, v = parts[0], parts[1]
                            if k == "throttled_usec":
                                return int(v)
                            elif k == "throttled_time":
                                return int(v) // 1000
            except Exception:
                continue
        return 0

    # ------------------------------------------------------------
    # PHASE 1: CALIBRATION
    # ------------------------------------------------------------
    print("\n[Phase 1] Calibrating to hardware...", flush=True)

    t_end = time.perf_counter() + 5.0
    while time.perf_counter() < t_end:
        unit_of_work()

    sample_times = []
    t_end = time.perf_counter() + 15.0

    while time.perf_counter() < t_end:
        t0 = time.perf_counter()
        unit_of_work()
        sample_times.append(time.perf_counter() - t0)

    if not sample_times:
        print("🚨 FATAL: CPU starved. No samples collected.", flush=True)
        while True:
            time.sleep(3600)

    median_time = statistics.median(sample_times)
    batch_size = max(1, int(1.0 / median_time))
    expected_duration = batch_size * median_time
    ALERT_THRESHOLD = expected_duration * 1.20

    print("\n[Phase 1 Complete] Hardware Profile Locked:", flush=True)
    print(f"  - Samples Taken: {len(sample_times)}", flush=True)
    print(f"  - Baseline:      {expected_duration:.4f}s per batch", flush=True)
    print(f"  - Alert Trigger: >{ALERT_THRESHOLD:.4f}s", flush=True)
    print("=" * 76, flush=True)
    print(flush=True)

    # Header for columnar output (76 chars wide max)
```

```
    hdr = f"{'STAT':<4}| {'UTIL':<10} | {'PERF':<13} | {'DIAGNOSIS':<11} | TELEMETRY"
    print(hdr, flush=True)
    print("-" * 76, flush=True)
    print(f"# base={expected_duration:.2f}s alert={ALERT_THRESHOLD:.2f}s "
          f"thr_min={THR_MS_MIN:.0f}ms", flush=True)

    # ------------------------------------------------------------
    # PHASE 2: MONITORING
    # ------------------------------------------------------------
    history = [expected_duration] * 3

    while True:
        start_time = time.perf_counter()
        start_tot, start_steal = get_os_cpu_stats()
        start_thr = get_cfs_throttle_usec()

        for _ in range(batch_size):
            unit_of_work()

        duration = time.perf_counter() - start_time
        end_tot, end_steal = get_os_cpu_stats()
        end_thr = get_cfs_throttle_usec()

        # Clamp deltas at 0
        tot_delta = max(0, end_tot - start_tot)
        steal_delta = max(0, end_steal - start_steal)
        thr_delta_us = max(0, end_thr - start_thr)
        thr_delta_ms = thr_delta_us / 1000.0

        steal_pct = 0.0
        if tot_delta > 0:
            steal_pct = (steal_delta / tot_delta) * 100.0

        history.pop(0)
        history.append(duration)
        rolling_avg = sum(history) / len(history)

        degradation_ratio = rolling_avg / expected_duration
        bar_len = int(degradation_ratio * 10)
        bar = "" * min(bar_len, 10)
        bar_padded = f"{bar:<10}"

        timing_str = f"{rolling_avg:4.2f}s ({degradation_ratio:4.2f}x)"
        tel = f"st:{steal_pct:04.1f}% tk:{steal_delta:03d} ms:{thr_delta_ms:03.0f}"

        if rolling_avg > ALERT_THRESHOLD:
            if steal_delta > 0 and thr_delta_ms >= THR_MS_MIN:
                diag = "[COMPLEX]"
            elif steal_delta > 0:
                diag = "[HV STEAL]"
            elif thr_delta_ms >= THR_MS_MIN:
                diag = "[CFS QUOTA]"
            else:
                diag = "[UNKNOWN]"

            out = f" ER| {bar_padded} | {timing_str:<13} | {diag:<11} | {tel}"
            print(out, flush=True)
        else:
            out = f" OK| {bar_padded} | {timing_str:<13} | {'[HEALTHY]':<11} | {tel}"
            print(out, flush=True)

        time.sleep(0.1)

---
```

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: critical-web-app
  namespace: critical-app
spec:
  replicas: 1
  selector:
    matchLabels:
      app: victim
  template:
    metadata:
      labels:
        app: victim
    spec:
      # Demo-only: Pinning to a specific Kubernetes Node (VM)
      # for reproducible contention testing.
      nodeName: kubernetes-cluster-8py4-kubernetes-cluster-8py4-nodepool-lnb7jx

      securityContext:
        runAsNonRoot: true
        seccompProfile:
          type: RuntimeDefault

      containers:
        - name: python-worker
          image: public.ecr.aws/bitnami/python:3.11
          command: ["python", "-u", "/app/worker.py"]
          resources:
            requests:
              cpu: "1"
              memory: "128Mi"
            limits:
              cpu: "1"
              memory: "128Mi"

          volumeMounts:
            - name: script-volume
              mountPath: /app
              readOnly: true

          securityContext:
            runAsUser: 1001
            allowPrivilegeEscalation: false
            capabilities:
              drop: ["ALL"]

      volumes:
        - name: script-volume
          configMap:
            name: victim-app-script
```

Appendix D:
S3 Compatible Store (MinIO)

The steps below show an example implementation of S3-compatible store, using MinIO, deployed on the VKS cluster. This is suitable for a lab environment but should not be used in production.

```
# Add MinIO Helm chart
# Note: MinIO charts (charts.min.io) are being
# replaced by helm.min.io which use Alstor
helm repo add minio https://charts.min.io/

# Create namespace & set permissions
NS=minio
kubectl create ns $NS
kubectl label --overwrite ns $NS pod-security.kubernetes.io/enforce=baseline

# Create secrets for MinIO root user and Loki user
miniorootpw=<pass for minio root>
lokiuserpw=<pass for loki user>

kubectl apply -f - << EOF
apiVersion: v1
kind: Secret
metadata:
  name: minio-root-credentials
  namespace: minio
type: Opaque
stringData:
  rootUser: minioadmin
  rootPassword: "$miniorootpw"
---
apiVersion: v1
kind: Secret
metadata:
  name: minio-loki-credentials
  namespace: minio
type: Opaque
stringData:
  password: "$lokiuserpw"
EOF

# create MinIO manifest
cat << EOF > custom-minio-values-5.4.0.yaml
mode: distributed
replicas: 2
pools: 1
drivesPerNode: 2

existingSecret: minio-root-credentials

persistence:
  enabled: true
  storageClass: vsan-esa-default-policy-raid5
  accessMode: ReadWriteOnce
  size: 100Gi   # per pod. With 4 pods this is 400Gi raw.

# limit resources
resources:
  requests:
    cpu: 100m
    memory: 512Mi
```

```
  limits:
    cpu: "1"
    memory: 2Gi

# Buckets Loki commonly expects (chunks/ruler/admin).
buckets:
  - name: loki-chunks
    policy: none
    purge: false
    versioning: false
  - name: loki-ruler
    policy: none
    purge: false
    versioning: false
  - name: loki-admin
    policy: none
    purge: false
    versioning: false

# Loki access key
users:
  - accessKey: loki
    existingSecret: minio-loki-credentials
    existingSecretKey: password
    policy: readwrite   # built-in policy
EOF

# Install using Helm
helm install minio minio/minio \
  -n minio \
  --version 5.4.0 \
  -f custom-minio-values-5.4.0.yaml
```

Appendix E: OpenTelemetry Demo Configuration

The manifest below describes the Helm configuration used for the OpenTelemetry demo, to show Jaeger tracing. For more details on the demo, visit https://opentelemetry.io/ecosystem/demo/

```
# otel-demo-values.yaml

# OpenTelemetry Demo override values used for the Jaeger tracing demonstration.
#
# Purpose:
# - Disable bundled observability backends
# - Use external Jaeger deployment as the trace backend
# - Run OpenTelemetry Collector as a Deployment rather than a
#   DaemonSet (avoiding hostPath / hostPort requirements)
# - Enrich telemetry with Kubernetes metadata
# - Prefix service names with the Kubernetes namespace for clearer Jaeger output

jaeger:
  enabled: false
prometheus:
  enabled: false
grafana:
  enabled: false
opensearch:
  enabled: false

# Set internal registry mirrors to avoid dockerhub limits
# replace 'my.internal.registry' as appropriate
x-images:
  postgres: &postgres_repo my.internal.registry/postgres
  valkey: &valkey_repo my.internal.registry/valkey/valkey
  otelcol: &otelcol_repo my.internal.registry/otel/opentelemetry-collector-contrib

x-otel:
  trace_processors: &trace_processors
    - k8sattributes
    - resource/service_namespace
    - transform/service_name
    - memory_limiter
    - batch

  # Prefix service.name with the Kubernetes namespace
  service_name_statement: &service_name_statement
    - set(attributes["service.name"], Concat([attributes["k8s.namespace.name"],
attributes["service.name"]], "/")) where not IsMatch(attributes["service.name"], "^.*/.*$")

components:
  # Disable flagd to simplify the demo and avoid extra non-essential image pulls
  flagd:
    enabled: false

  # Override Docker Hub sourced images to use the internal registry
  postgresql:
    imageOverride:
      repository: *postgres_repo
  valkey-cart:
    imageOverride:
      repository: *valkey_repo

opentelemetry-collector:
```

```yaml
# Use a Deployment rather than a DaemonSet to avoid Pod Security issues
mode: deployment
replicaCount: 1

image:
  repository: *otelcol_repo

presets:
  # Keep the collector focused on application OTLP telemetry for the tracing demo
  hostMetrics:
    enabled: false
  kubeletMetrics:
    enabled: false
  clusterMetrics:
    enabled: false
  kubernetesEvents:
    enabled: false

config:
  receivers:
    otlp:
      # Receive OTLP traffic from the demo services over gRPC and HTTP
      protocols:
        grpc:
          endpoint: ${env:MY_POD_IP}:4317
        http:
          endpoint: ${env:MY_POD_IP}:4318

  processors:
    # Enrich telemetry with Kubernetes context for correlation and filtering
    k8sattributes:
      auth_type: serviceAccount
      extract:
        metadata:
          - k8s.namespace.name
          - k8s.pod.name
          - k8s.pod.uid
          - k8s.deployment.name
          - k8s.replicaset.name
          - k8s.statefulset.name
          - k8s.node.name
          - k8s.container.name
      pod_association:
        - sources:
            - from: connection

    # Set a stable service.namespace for the demo environment
    resource/service_namespace:
      attributes:
        - key: service.namespace
          value: otel-demo
          action: upsert

    # Rewrite service.name for clearer grouping in Jaeger.
    transform/service_name:
      error_mode: ignore
      trace_statements:
        - context: resource
          statements: *service_name_statement
      metric_statements:
        - context: resource
          statements: *service_name_statement
      log_statements:
        - context: resource
          statements: *service_name_statement
```

```
    memory_limiter:
      check_interval: 5s
      limit_percentage: 80
      spike_limit_percentage: 25

    batch: {}

  exporters:
    # Keep debug output for simple troubleshooting
    debug: {}

    # Export traces to the external Jaeger deployment in the tracing namespace
    otlp/jaeger-external:
      endpoint: jaeger-collector.tracing.svc.cluster.local:4317
      tls:
        # Lab setting: insecure transport to the in-cluster Jaeger endpoint
        insecure: true

  connectors:
    # Generate span-derived metrics inside the collector
    spanmetrics: {}

  service:
    pipelines:
      traces:
        receivers: [otlp]
        processors: *trace_processors
        exporters: [otlp/jaeger-external, debug, spanmetrics]

      metrics:
        receivers: [otlp, spanmetrics]
        processors: *trace_processors
        exporters: [debug]

      logs:
        receivers: [otlp]
        processors: *trace_processors
        exporters: [debug]
```

vmware® by Broadcom

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