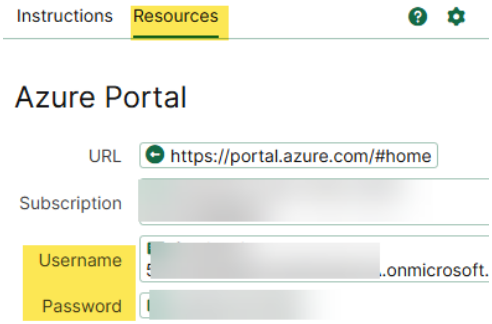
Module 1 – Initial Setup

This section covers start-to-finish, AKS setup via az CLI (PowerShell) for East US 2, with a tainted system pool and a separate user pool, plus monitoring to a Log Analytics workspace.

Step 0 : Login & (optional) subscription



`az login -> Choose "Work or School Account" -> Set the correct subscription -> Enter the username/password found in the Resources tab. 'Apply to All Accounts' if asked.`

Step 1.1 : Set Variables

```
$id    = "55244003"    < Get this Id from the Instructions tab
$loc   = "eastus2"
$rg    = "azure-rg"
$law   = "aks-$id-law"
$aks   = "aks-$id"
```

Step 1.2 : Set VM SKUs

Get the list of available VM SKU's with 2 cores in the selected region or use the default below

```
az vm list-sizes --location $loc --query "[?numberOfCores == ``2``].{Name:name}" -o table

$VM_SKU  = "Standard_A2m_v2"
```

Step 1.3 : Register Providers and Confirm Registration. Add Extensions

```
az provider register --namespace Microsoft.Storage
az provider register --namespace Microsoft.Compute
az provider register --namespace Microsoft.Network
az provider register --namespace Microsoft.Monitor
az provider register --namespace Microsoft.Insights
az provider register --namespace Microsoft.ManagedIdentity
az provider register --namespace Microsoft.OperationalInsights
az provider register --namespace Microsoft.OperationsManagement
az provider register --namespace Microsoft.KeyVault
az provider register --namespace Microsoft.ContainerRegistry
az provider register --namespace Microsoft.ContainerService
az provider register --namespace Microsoft.Kubernetes

az provider show --namespace Microsoft.Storage --query "registrationState" -o tsv
az provider show --namespace Microsoft.Compute --query "registrationState" -o tsv
az provider show --namespace Microsoft.Network --query "registrationState" -o tsv
az provider show --namespace Microsoft.Monitor --query "registrationState" -o tsv
az provider show --namespace Microsoft.Insights --query "registrationState" -o tsv
az provider show --namespace Microsoft.ManagedIdentity --query "registrationState" -o tsv
az provider show --namespace Microsoft.OperationalInsights --query "registrationState" -o tsv
```

```
az provider show --namespace Microsoft.OperationsManagement --query "registrationState" -o tsv
az provider show --namespace Microsoft.KeyVault --query "registrationState" -o tsv
az provider show --namespace Microsoft.ContainerRegistry --query "registrationState" -o tsv
az provider show --namespace Microsoft.ContainerService --query "registrationState" -o tsv
az provider show --namespace Microsoft.Kubernetes --query "registrationState" -o tsv
```

Step 2 : Create Resource group

```
az group create -n $rg -l $loc
```

Step 3 : Create Log Analytics workspace

```
az monitor log-analytics workspace create -g $rg -n $law -l $loc
```

```
$lawId = az monitor log-analytics workspace show -g $rg -n $law --query id -o tsv
$lawId
```

Step 4 : Create AKS cluster

```
az aks create --nodepool-name syspool --node-count 2 --generate-ssh-keys --node-vm-size
$VM_SKU --nodepool-taints "CriticalAddonsOnly=true:NoSchedule" --name $aks --resource-
group $rg --enable-addons monitoring --workspace-resource-id $lawId
```

Step 5 : Add a user node pool

```
az aks nodepool add `
  -g $rg --cluster-name $aks `
  -n userpool --mode User `
  --node-count 1 `
  --node-vm-size $VM_SKU
```

Step 6 : Get kubeconfig & quick validation

```
az aks get-credentials -g $rg -n $aks --overwrite-existing

# Check pools & taints
kubectl get nodes -o custom-
columns=NAME:.metadata.name,POOL:.metadata.labels.agentpool,TAINTS:.spec.taints

# Expectation:
# - syspool nodes include:  [ {key=CriticalAddonsOnly, value=true, effect=NoSchedule} ]
# - userpool nodes: no taints by default
```

Step 7 : Verify monitoring is wired to your workspace

```
az aks show -g $rg -n $aks --query "azureMonitorProfile.containerInsights" -o jsonc
```

Step 8 : (Optional) Basic smoke test

```
# Create a simple namespace and deployment on the user pool
kubectl create ns demo
kubectl -n demo create deployment hello --image=nginx:stable-alpine
kubectl -n demo expose deployment hello --port=80 --type=LoadBalancer
kubectl -n demo get svc hello –watch
$ip = kubectl -n demo get svc hello -o jsonpath='{.status.loadBalancer.ingress[0].ip}' 2>$null
Invoke-WebRequest "http://$ip/"
kubectl delete ns demo
```

Step 9 : (Stop! only do this at the end of this Module)

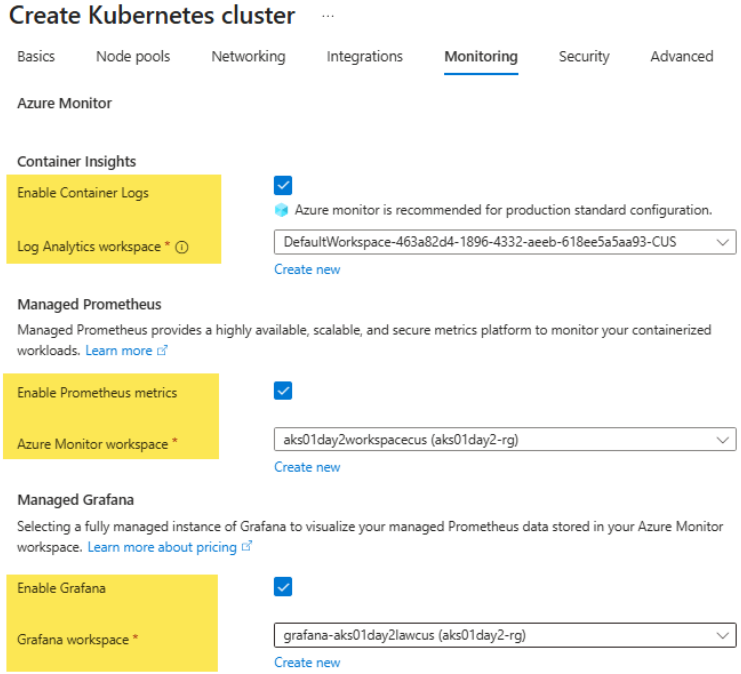
```
az group delete -n $rg --yes --no-wait
```

Step 10 : One-flow enablement: Prometheus + Container Insights + Grafana

Goal: Use the portal onboarding to enable Managed Prometheus (metrics), Container insights (logs), and Managed Grafana (visualization) for your AKS cluster in one pass.

Steps

1. Azure Portal → open your **AKS** → **Insights** (or **Monitoring** blade) → **Enable monitoring**.
2. Choose **Managed Prometheus** (metric collection), **Container insights** (log collection), and **Managed Grafana** (visualization). Either select existing workspaces (LA + Azure Monitor workspace) or allow the wizard to create them. **Create/Review/Enable**. Microsoft Learn



3. (If you prefer CLI/Terraform/ARM/Policy, the same article lists commands and templates.) Microsoft Learn

Validate (success = "green check")

- AKS → Monitoring → **Insights** loads without "enable" prompts (tiles populate). Microsoft Learn
- In AKS → Monitoring → **Metrics**, you can select the AKS resource and plot at least one platform metric (e.g., **Node** CPU). Microsoft Learn
- An **Azure Managed Grafana** workspace exists and opens. Microsoft Learn

Troubleshooting

- If onboarding failed or resources are mismatched by region, re-run the wizard and align regions for AKS + workspaces. The enablement doc has per-method notes. Microsoft Learn

Step 11 : Route control-plane logs (Activity + Resource logs) to Log Analytics

Goal: Ensure Activity log and AKS resource logs land in Log Analytics for correlation and RCA.

Steps - Activity log (subscription-level)

1. Azure Portal → AKS → Activity log → Export Activity log.
2. **Add diagnostic setting**: choose **Log Analytics workspace** as the destination. Save. Microsoft Learn

Steps - Resource logs (AKS cluster)

1. AKS → Monitoring → **Diagnostic settings** → **Add diagnostic setting**.
2. Select **Resource log categories** you'll need (for example, kube-apiserver, kube-audit, kube-audit-admin, autoscaler, controller-manager, etc. and see the **supported categories** reference).
3. Destination: **Send to Log Analytics workspace** (use the same LA workspace you chose for CI) and prefer **Resource-specific tables**. Save. Microsoft Learn

Validate

- In your **Log Analytics workspace**, run below queries , to confirm subscription events are arriving for Activity Logs and Resource Logs.

```
AzureActivity | take 5
KubeEvents | take 5
KubePodInventory | take 5
```

- If these return, LA is ingesting CI data; your AKS resource logs may appear under other table names, just confirm **some** new AKS tables are active.)
- Confirm new AKS resource-specific tables appear (e.g., **KubeEvents**, **KubeNodeInventory,** **KubePodInventory**) after some minutes of activity. Microsoft Learn

Why this matters

- Activity log = **what changed**; AKS **resource logs** = component diagnostics (API server, scheduler, autoscaler, etc.). Both route via **diagnostic settings**. Microsoft Learn

Step 12 : Light up Container Insights (Triage views + Live Data)

Goal: Use Azure Monitor's Container Insights to rapidly triage AKS cluster, node, and workload health-including live tailing of logs, events, and metrics, without needing kubectl.

Steps

1. Open Container Insights

- Navigate to your **AKS cluster** in the Azure Portal.
- In the left-hand menu, under **Monitor**, click **Insights** (or under **Containers** in the Azure Monitor menu).
- If it's not enabled, click **Enable Container Insights** to onboard. This will provision a Data Collection Rule and start sending data. Microsoft Learn / Also check DCR for CI

2. Explore Health Dashboards & Reports

- You'll see multiple tabs:
  - **Cluster/Overview**: Multi-cluster health summary, including counts of healthy, warning, or critical clusters. Microsoft Learn / Microsoft Learn
  - **Nodes**: Per-node health, CPU/memory usage, and Kubernetes conditions.
  - **Controllers**: Workload status across Deployments, StatefulSets, ReplicaSets (rollouts, restarts).
  - **Containers (Pods)**: Drill into individual pod/container-level metrics and statuses.
  - **Metric ratings**:
    - **Min:** The lowest recorded CPU usage among nodes over the selected time range.
    - **Avg:** The mean (average) CPU usage across all nodes.
    - **50th (Median):** The middle value (half the nodes are using more, half less). This is useful to see the "typical" node load, without skew from outliers.
    - **90th percentile:** 90% of the nodes are at or below this usage, 10% are higher. Helps spot "heavier" consumers.
    - **95th percentile:** Even stricter: 95% of the nodes are below this, 5% above. Often used in capacity planning to capture high-load cases without being thrown off by extreme spikes.
    - **Max:** The highest recorded CPU usage across nodes (the outlier).
- To dive deeper, use the prebuilt **Workbooks** such as **Cluster Optimization**, **Node Monitoring**, or **Resource Monitoring**. Microsoft Learn / Microsoft Learn

3. Use Live Data (Logs, Events, Metrics)

- Select any resource (Container, Node, Controllers/Deployment, etc.) in one of the tabs.
- From its Overview pane, click:
  - **Live Logs** to stream container stdout/stderr like kubectl logs.
  - **Live Events** to receive real-time Kubernetes events (similar to kubectl get events). Microsoft Learn / GitHub
  - **Live Metrics** to view CPU/memory metrics live (like kubectl top). Azure Documentation+3Microsoft Learn+3Microsoft Learn+3
- Live Metrics won't work if you have the **Managed Prometheus visualizations** enabled. GitHub+11Microsoft Learn+11Microsoft Learn+11

Step 13 : Validate platform metrics in Metrics Explorer

Goal: Prove you can chart common AKS metrics and use dimensions/splits for diagnosis.
Steps
1. Navigate to Metrics Explorer
   - In the **Azure Portal**, go to **Monitor → Metrics**.
   - **Set the scope** to your **AKS cluster resource** (subscription, resource group, AKS instance).
   - The Metrics Explorer will automatically show available AKS platform metrics (e.g., CPU, memory, network).
2. Select & Plot a Platform Metric with Splits
   - Under **Metric Namespace**, select **Container Service**.
   - Choose a metric like **CPU Usage Percentage** (node_cpu_usage_percentage) or **Memory Working Set Bytes**.
     These metrics support dimensions node and nodepool.
   - Set the **Time Range** (e.g., last 1-6 hours).
   - In **Aggregation**, select appropriate statistics (e.g., *Average* or *Maximum*).
   - Click **Split by** and choose **node** to display a separate line per node. This helps identify hot nodes.
3. (Optional) Include Control Plane Metrics

To visualize API server or etcd performance:
   - Ensure the AKS cluster **Control Plane Metrics (Preview)** feature is enabled via Managed Prometheus integration.
   - In Metrics Explorer, select metrics like **API Server CPU Usage Percentage** (apiserver_cpu_usage_percentage) or **Inflight Requests**.
   - Use dimension breakdowns if available, then plot using time series for analysis.
4. Save or Share Your Chart
   - Once satisfied, click **Pin to dashboard** to add this chart to an Azure dashboard.

Step 14 : Validate Prometheus metrics and Grafana dashboards

Goal: Ensure Prometheus metrics from your AKS cluster are being **ingested into an Azure Monitor workspace** and can be visualized through Azure's Metrics Explorer (via PromQL) and Azure Managed Grafana dashboards.

Part A - Confirm Prometheus Metrics in Azure Monitor

1. In the Azure Portal, navigate to your **Azure Monitor Workspace**.
2. Go to **Metrics** (Metrics Explorer).
3. Set **Scope** to your Monitor Workspace.
4. Click **Add metric** → choose **"Add with Editor"** (PromQL query interface).
   - Optionally, use the **Builder** view to inspect available metrics. Metrics-Explorer / PromQL / Metric Charts
5. Enter a basic PromQL query, for example: kube_pod_info, or any known metric from kube-state-metrics. Then click **Run**.
6. Confirm that a time series chart renders successfully.

Part B - Use Prometheus Explorer Workbook (Optional)

- Navigate to **Workbooks** on the Monitor Workspace page.
- Choose the **Prometheus Explorer** workbook.
- Input a PromQL query, for e.g., `rate(container_cpu_usage_seconds_total[5m])` → gives you CPU cores used per second by container/pod/node.
  - rate() (or irate()) expects a **counter** (monotonically increasing value).
  - Visualize results via **Graph**, **Grid**, or **Dimensions** tabs. Prometheus Workbooks

Part C - Explore Prometheus Dashboards in Grafana

- In Grafana, go to **Dashboards → All Dashboards**.
- Open a prebuilt dashboard such as:
  - Kubernetes / Compute Resources / Cluster
  - Kubernetes / Compute Resources / Pod
- Observe metrics like CPU, memory, pod restarts, kubelet status, or API server latency-depending on dashboard layout.

Step 15 : Application Insights | Auto-Instrumentation test

Goal
Verify that your application deployed to AKS is sending telemetry (requests, availability, live metrics) to Application Insights using the **auto-instrumentation** feature with OpenTelemetry. The steps below result in auto-instrumentation by injecting the Azure Monitor OpenTelemetry Distro into application pods to generate telemetry. For more on autoinstrumentation and its benefits, see the official documentation.

Step 15.1: Create or Select an Application Insights Resource

- In the Azure Portal, navigate to **Application Insights → Create**.
- Use a naming convention like aks-<env> and select the **Log Analytics Workspace** for log integration.

Step 15.2: Prepare Azure CLI & Feature Flags

1. Install the preview extension:
   ```
   az extension add --name aks-preview
   ```

2. Register the preview feature:
   ```
   az login
   az feature register --namespace "Microsoft.ContainerService" --name
   "AzureMonitorAppMonitoringPreview"
   ```

3. Check status (this may take hours to move from *Registering → Registered*):
   ```
   az feature list -o table --query "[?contains(name,
   'Microsoft.ContainerService/AzureMonitorAppMonitoringPreview')].{Name:name,State:prope
   rties.state}"
   ```

4. Once Registered, re-register the provider:
   ```
   az provider register --namespace "Microsoft.ContainerService"
   az provider show --namespace "Microsoft.ContainerService" --query "registrationState"
   ```

Step 15.3: Enable App Monitoring on AKS Cluster

```
az aks update `
  --resource-group <RESOURCE_GROUP> `
  --name <CLUSTER_NAME> `
  --enable-azure-monitor-app-monitoring
```

Step 15.4: Create and Apply the Instrumentation Custom Resource

The **Instrumentation** CRD defines where telemetry should go. You need one per namespace or scenario.

Save below to **instrumentation.yaml**

```
apiVersion: monitor.azure.com/v1
kind: Instrumentation
metadata:
  name: cr1
  namespace: default
spec:
  settings:
    autoInstrumentationPlatforms: []
  destination:
    applicationInsightsConnectionString: "<Replace with Instrumentation key from App Insights>"
```

Apply this file:

```
kubectl apply -f .\instrumentation.yaml
kubectl describe Instrumentation -n default
```

Step 15.5: Deploy the Sample App with Injection Annotation

Save below to **hello-ai.yaml**

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: hello-ai
  labels:
    app: hello-ai
spec:
  replicas: 1
  selector:
    matchLabels:
      app: hello-ai
  template:
    metadata:
      labels:
        app: hello-ai
      annotations:
        # Tells the injector to use Instrumentation CR named "cr1"
        instrumentation.opentelemetry.io/inject-nodejs: "cr1"
    spec:
      containers:
      - name: aks-helloworld
        image: mcr.microsoft.com/azuredocs/aks-helloworld:v1
        ports:
        - containerPort: 80
---
apiVersion: v1
kind: Service
metadata:
  name: hello-ai
  labels:
    app: hello-ai
spec:
  type: LoadBalancer
  selector:
    app: hello-ai
  ports:
  - port: 80
    targetPort: 80
```

```
# Replace the existing app if it exists and the service with this manifest
kubectl delete deploy hello-ai --ignore-not-found
```

```
kubectl delete svc hello-ai --ignore-not-found
```

```
# Apply to default ns as that's where CR is
kubectl apply -f .\hello-ai.yml -n default
k get all -n default

kubectl rollout restart deployment/hello-ai -n default
kubectl rollout status deployment/hello-ai -n default
```

Step 15.6: Confirm Pods are enabled for Auto-Instrumentation

Inspect pod's annotations and environment variables to confirm the injection happened. This should result in an annotation like `instrumentation.opentelemetry.io/inject-nodejs: cr1` and Environment variables or NODE_OPTIONS flags referencing the OpenTelemetry/Monitor shim.

```
$pod = kubectl get pods -l app=hello-ai -n default -o name
kubectl get $pod -n default -o yaml | findstr /i "otel monitor azure azmon"
```

results in below:

```
- name: OTEL_RESOURCE_ATTRIBUTES
       value: InstrumentationKey=….
    image: mcr.microsoft.com/azuredocs/aks-helloworld:v1
```

```
kubectl get $pod -o yaml -n default | findstr /I "instrumentation.opentelemetry.io/inject-nodejs: cr1"
```

results in below:

```
instrumentation.opentelemetry.io/inject-nodejs: cr1
```

Also verify the **mutating webhook** is installed and active:

```
kubectl get mutatingwebhookconfigurations | Select-String -Pattern "monitor|otel"
```

Step 15.7: Generate Load

```
$ip = (kubectl get svc hello-ai -n default -o jsonpath='{.status.loadBalancer.ingress[0].ip}')
while ([string]::IsNullOrEmpty($ip)) { Start-Sleep 5; $ip = (kubectl get svc hello-ai -n default -o
jsonpath='{.status.loadBalancer.ingress[0].ip}') }

1..30 | % {
  try {
    $resp = Invoke-WebRequest "http://$ip" -UseBasicParsing
    $txt = ($resp.Content -replace '\s+', ' ' -replace '<[^>]+>', '')
    if ($txt.Length -gt 80) { $txt = $txt.Substring(0,80) + "..." }
    Write-Host ("[OK] " + (Get-Date -Format HH:mm:ss) + " - " + $txt)
  } catch { Write-Host ("[ERR] " + (Get-Date -Format HH:mm:ss) + " - " + $_.Exception.Message) }
  Start-Sleep 1
}
```

Should give:

```
[OK] xx -   Welcome to Azure Container Instances!   h1 { color: darkblue; font-family:aria...
```

Step 15.8: Verify in Application Insights

1. Overview → See Request Rate, Failures, and Performance populate.
2. Live Metrics Stream → Observe live requests.
3. Application Map → See service components.

Step 15.9: Optional – Availability Test

1. In Application Insights → **Availability**.
2. Create a Standard Test.
3. Fill in the required details:
    1. Test name: e.g., **hello-ai-health-test**
    2. Test type: URL ping test
    3. URL: `http://<external-ip>/`
        - Replace <external-ip> with the LoadBalancer IP from your AKS service using
          `kubectl get svc hello-ai -o jsonpath='{.status.loadBalancer.ingress[0].ip}'`
4. Test frequency: every 5 or 10 minutes (default 5m is fine).
5. Locations: choose multiple test locations for resiliency.

6. Monitor uptime results.

At this point, your AKS app is auto-instrumented via the preview add-on, telemetry is flowing into Application Insights, and you can confirm via both metrics and live diagnostics.

Step 15.10: Clean up the bad deploy (if it exists)

```
kubectl delete deploy hello-ai --ignore-not-found
kubectl delete svc hello-ai --ignore-not-found
```

Step 16 : Validate previous steps for this app

Go through earlier steps and confirm findings on the above run deployment

Step 17 : Sanity checks

Validate if below is successful:

A. Logs (change + behavior)

- **Activity log** is exported to **Log Analytics** and returns rows:
  ```
  AzureActivity
  | take 5
  ```
- **AKS logs** are arriving (either resource-specific tables or AzureDiagnostics) and you can read a few rows from typical CI tables:
  ```
  KubeEvents | take 5
  KubePodInventory | take 5
  ```

B. Metrics (fast health)

- **Metrics Explorer** shows at least one meaningful chart (e.g., **Node CPU** split by node, or API server latency if available).
- **Azure Monitor workspace (Prometheus)** returns at least one Prometheus time series.

C. Dashboards (first pane)

- **Azure Managed Grafana** is connected to your **AMW** and at least one **AKS dashboard** (Nodes/Pods/Kubelet/API server) shows live data.

D. (Optional) App telemetry

- **Application Insights** shows request/availability signals for a sample app.

Artifacts to save (screenshots or exports)

- Activity log query results; CI **Live logs** window; one **Metrics Explorer** chart; one **Grafana** dashboard; (optional) App Insights availability chart.

Module 2 – AKS Application setup and Test

Lab 2.0 - Install aks-store-demo

This section installs the **aks-store-demo** using the repo's **all-in-one** manifest.

```
# 1) Create a clean namespace (repo examples use 'pets')
kubectl create ns pets

# 2) Deploy everything (MongoDB, RabbitMQ, services, UIs, and virtual load)
kubectl apply -n pets `
  -f https://raw.githubusercontent.com/Azure-Samples/aks-store-demo/main/aks-store-all-in-one.yaml

# Watch until all Pods are Ready
kubectl config set-context --current --namespace=pets
kubectl get pods
```

It deploys the below services:

- store-front Service **type: LoadBalancer** (port 80 → container 8080)
- store-admin Service **type: LoadBalancer** (port 80 → 8081)
- order-service (ClusterIP:3000, /health)
- makeline-service (ClusterIP:3001, /health)
- product-service (ClusterIP:3002, /health)
- mongodb (StatefulSet, ClusterIP:27017)
- rabbitmq (StatefulSet, ClusterIP:5672 AMQP & 15672 mgmt; default user/pass **username/password**)
- virtual-customer & virtual-worker Deployments with default "100 orders/hour" env vars.

Lab 2.1 - Validate key services

1) External UIs (store-front & store-admin)

```
# Get the public IPs provisioned by LoadBalancer Services
kubectl -n pets get svc store-front, store-admin

# When EXTERNAL-IP shows, set variables
$FRONT_IP = kubectl -n pets get svc store-front -o jsonpath="{.status.loadBalancer.ingress[0].ip}"
$ADMIN_IP = kubectl -n pets get svc store-admin -o jsonpath="{.status.loadBalancer.ingress[0].ip}"

Write-Output "front: http://$FRONT_IP/   admin: http://$ADMIN_IP/"
```

For health check run, Invoke-WebRequest http://$FRONT_IP/health which should return OK; same for admin at Invoke-WebRequest http://$ADMIN_IP/health which should return OK.

2) Core microservices Internal Check (ClusterIP health)

```
# Order service
kubectl -n pets run tmp --rm -it --restart=Never --image=busybox `
       -- /bin/sh -c "wget -qO- http://order-service.pets.svc.cluster.local:3000/health"

# Makeline service
kubectl -n pets run tmp2 --rm -it --restart=Never --image=busybox `
       -- /bin/sh -c "wget -qO- http://makeline-service.pets.svc.cluster.local:3001/health"

# Product service
kubectl -n pets run tmp3 --rm -it --restart=Never --image=busybox `
       -- /bin/sh -c "wget -qO- http://product-service.pets.svc.cluster.local:3002/health"
```

Each should respond healthy with **"status":"ok",** since those paths are defined in probes.

Lab 2.2 – Check existing resource limit

Run below to validate existing resource limits

```
$deploys = "store-front","order-service","makeline-service","product-service"
$jsonpath = @"
{range .spec.template.spec.containers[*]}{.name}{' => requests: '}{.resources.requests}{' , limits:
'}{.resources.limits}{'\n'}{end}
"@

foreach ($d in $deploys) {
  Write-Host "`nDeployment: $d"
  kubectl get deploy $d -o=jsonpath="$jsonpath"
}

kubectl get statefulset mongodb -o jsonpath="{range .spec.template.spec.containers[*]}{.name}:
requests={.resources.requests} , limits={.resources.limits}{'\n'}{end}"

kubectl get statefulset rabbitmq -o jsonpath="{range .spec.template.spec.containers[*]}{.name}:
requests={.resources.requests} , limits={.resources.limits}{'\n'}{end}"
```

Should return results below:

```
Deployment: store-front
store-front => requests: {"cpu":"1m","memory":"200Mi"} , limits: {"cpu":"1","memory":"512Mi"}

Deployment: order-service
order-service => requests: {"cpu":"1m","memory":"50Mi"} , limits: {"cpu":"100m","memory":"256Mi"}

Deployment: makeline-service
makeline-service => requests: {"cpu":"1m","memory":"6Mi"} , limits: {"cpu":"5m","memory":"20Mi"}

Deployment: product-service
product-service => requests: {"cpu":"1m","memory":"1Mi"} , limits: {"cpu":"2m","memory":"20Mi"}

mongodb: requests={"cpu":"5m","memory":"75Mi"} , limits={"cpu":"25m","memory":"1Gi"}

rabbitmq: requests={"cpu":"10m","memory":"128Mi"} , limits={"cpu":"250m","memory":"256Mi"}
```

Lab 2.3 – Tighten pod **resource limits** (force contention)

These patches make pods easier to throttle/OOM (which then shows up clearly in Container Insights/Logs).

Deployments (store-front, order-service, makeline-service, product-service)

```
# CPU: request 100m / limit 200m; Mem: request 128Mi / limit 256Mi
kubectl -n pets set resources deploy/store-front --requests=cpu=100m,memory=128Mi --
limits=cpu=200m,memory=256Mi
kubectl -n pets set resources deploy/order-service --requests=cpu=100m,memory=128Mi --
limits=cpu=200m,memory=256Mi
kubectl -n pets set resources deploy/makeline-service --requests=cpu=100m,memory=128Mi --
limits=cpu=200m,memory=256Mi
kubectl -n pets set resources deploy/product-service --requests=cpu=100m,memory=128Mi --
limits=cpu=200m,memory=256Mi

# Roll them to pick up any unchanged templates
kubectl -n pets rollout restart deploy/store-front deploy/order-service deploy/makeline-service
deploy/product-service
kubectl -n pets rollout restart deploy/virtual-customer deploy/virtual-worker
```

StatefulSets (MongoDB & RabbitMQ) - Use slightly higher memory to avoid instant collapse, but still tight

```
# MongoDB
kubectl -n pets patch statefulset mongodb -p `
'{"spec":{"template":{"spec":{"containers":[{"name":"mongodb","resources":{"requests":{"cpu":"200m"
,"memory":"256Mi"},"limits":{"cpu":"400m","memory":"512Mi"}}}]}}}}'

# RabbitMQ
kubectl -n pets patch statefulset rabbitmq -p `
'{"spec":{"template":{"spec":{"containers":[{"name":"rabbitmq","resources":{"requests":{"cpu":"200m
","memory":"256Mi"},"limits":{"cpu":"400m","memory":"512Mi"}}}]}}}}'
```

After these changes, watch for **CPU throttling**, **probe failures**, and **OOMKilled** under higher load.

Validate that Pods are fully up

```
kubectl get pods
```

Lab 2.4 – Add **Horizontal Pod Autoscalers** (so you see replica changes)

```
# Scale on CPU to demonstrate controller/pod churn in Insights
kubectl -n pets autoscale deploy/store-front --cpu-percent=50 --min=1 --max=2
kubectl -n pets autoscale deploy/order-service --cpu-percent=50 --min=1 --max=2
kubectl -n pets autoscale deploy/makeline-service --cpu-percent=50 --min=1 --max=2
kubectl -n pets autoscale deploy/product-service --cpu-percent=50 --min=1 --max=2
kubectl -n pets get hpa

# for manual scale down replicas to 1
kubectl scale deployment store-front -n pets --replicas=1
kubectl patch hpa store-front -n pets --type=merge --patch
'{\"spec\":{\"minReplicas\":1,\"maxReplicas\":2}}'

# to delete
kubectl delete hpa store-front product-service order-service makeline-service -n pets
```

Lab 2.5 – Increase realistic load through services

The virtual-customer and virtual-worker pods in aks-store-demo simulate placing orders and processing them:

- **virtual-customer** → continuously POSTs new orders to **order-service** (which then enqueues to RabbitMQ).
- **virtual-worker** → dequeues from RabbitMQ, pushes each order through **makeline-service**, and writes to **mongodb**.

So by turning up their rate, you naturally stress:

- **order-service** CPU/memory (handling requests, writing to queue).
- **makeline-service** CPU (order processing logic).
- **mongodb** CPU/memory/disk (persisting orders).

```
# CHECK DEFAULTS
kubectl -n pets get deploy virtual-customer -o jsonpath='{.spec.template.spec.containers[0].env}'
{"name":"ORDERS_PER_HOUR","value":"100"}]

kubectl -n pets get deploy virtual-worker -o jsonpath='{.spec.template.spec.containers[0].env}'
{"name":"ORDERS_PER_HOUR","value":"100"}]

kubectl -n pets exec deploy/virtual-worker -- printenv

# INCREASE LOAD
kubectl -n pets set env deploy/virtual-customer ORDERS_PER_HOUR=90000
kubectl -n pets set env deploy/virtual-worker ORDERS_PER_HOUR=90000

# CHECK SERVICE LOGS
kubectl -n pets logs deploy/virtual-customer --tail=200 -f
kubectl -n pets logs deploy/virtual-worker --tail=200 -f

# RESET on completion
kubectl -n pets set env deploy/virtual-customer ORDERS_PER_HOUR=100
kubectl -n pets set env deploy/virtual-worker ORDERS_PER_HOUR=100
```

Lab 2.6 – Stress test: Turn up k6 (open-loop / constant-arrival-rate)

What it is: A scriptable load tester (JS) that can also run in an **open-loop mode**: instead of simulating a fixed number of users, k6 schedules requests at a **target rate** (e.g., 1000 requests/second). It spins up as many virtual users as needed (up to your configured maxVUs) to maintain that arrival rate.

Why it's useful: This simulates an external client system hammering your service at a fixed request rate, independent of how slow or fast responses are. It's ideal for stressing infrastructure capacity and to *force the system to show stress* (throttling, OOM, autoscaling).

Latency in k6 (open-loop):

- Latency is still measured automatically per request (http_req_duration, plus breakdowns like http_req_connecting, http_req_waiting, etc.).
- Because requests keep coming at the set rate, if the service slows down, **latency grows** and **errors increase** instead of throughput dropping.
- This makes bottlenecks and resource exhaustion show up clearly in monitoring dashboards.

```
kubectl -n pets run k6-front --rm -it --restart=Never `
  --image=grafana/k6:0.49.0 --command -- sh -lc "
cat <<'EOF' > /tmp/test.js
import http from 'k6/http';
import { sleep } from 'k6';

export const options = {
  scenarios: {
    high_rps: {
      executor: 'constant-arrival-rate',
      rate: 1000,           // <<< requests per second
      timeUnit: '1s',
      duration: '10m',      // <<< hold long enough to show in CI charts
      preAllocatedVUs: 400,
      maxVUs: 1000
    }
  },
  thresholds: {
    http_req_failed: ['rate<0.02'],
    http_req_duration: ['p(95)<1500']
  }
};

const BASE = 'http://store-front.pets.svc.cluster.local';

export default function () {
  http.get(BASE + '/');     // you can swap to a heavier path if you like
  sleep(0.01);
}
EOF
k6 run /tmp/test.js"
```

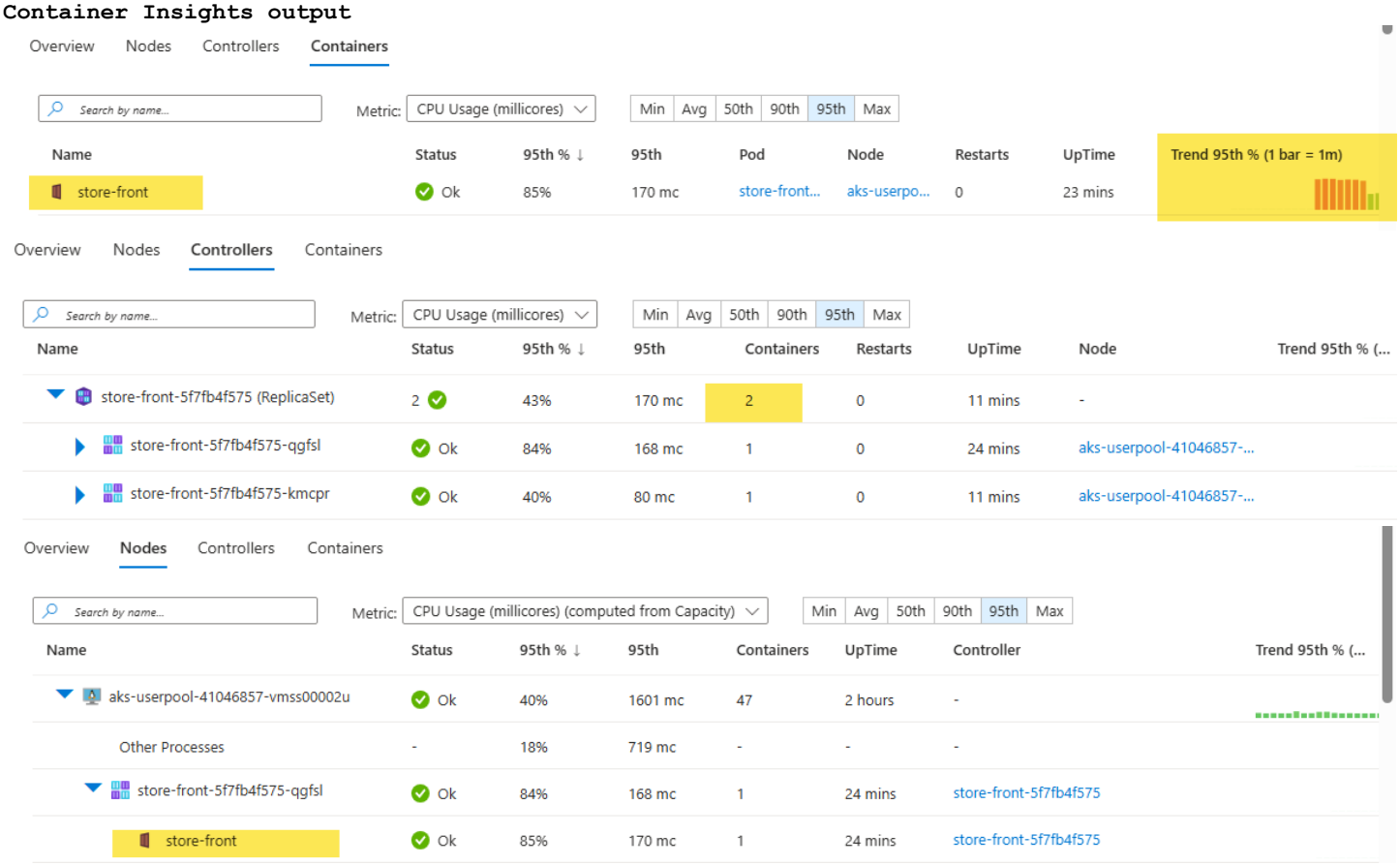
Resulting output as it increases traffic surge

```
running (04m17.2s), 0011/0400 VUs, 257154 complete and 0 interrupted iterations
high_rps   [=======>-----------] 0011/0400 VUs  04m17.2s/10m0s  1000.00 iters/s
```

Closure as seen below

```
    data_received..................: 409 MB 681 kB/s
    data_sent......................: 60 MB  100 kB/s
    http_req_blocked...............: avg=6.15µs   min=1.04µs   med=2.92µs   max=26.04ms p(90)=3.82µs   p(95)=4.35µs
    http_req_connecting............: avg=1.43µs   min=0s       med=0s       max=22.3ms  p(90)=0s       p(95)=0s
  ✓ http_req_duration..............: avg=338.75µs min=127.88µs med=222.28µs max=61.14ms p(90)=464.7µs  p(95)=703.27µs
      { expected_response:true }...: avg=338.75µs min=127.88µs med=222.28µs max=61.14ms p(90)=464.7µs  p(95)=703.27µs
  ✓ http_req_failed................: 0.00%  ✓ 0          ✗ 600001
    http_req_receiving.............: avg=72.18µs  min=9.41µs   med=39.61µs  max=44.12ms p(90)=96.23µs  p(95)=170.54µs
    http_req_sending...............: avg=19.62µs  min=5.07µs   med=12.77µs  max=22.04ms p(90)=22.56µs  p(95)=31.44µs
    http_req_tls_handshaking.......: avg=0s       min=0s       med=0s       max=0s      p(90)=0s       p(95)=0s
    http_req_waiting...............: avg=246.94µs min=101.18µs med=161.55µs max=60.54ms p(90)=329.69µs p(95)=486.94µs
    http_reqs......................: 600001 999.982065/s
    iteration_duration.............: avg=10.9ms   min=10.18ms  med=10.76ms  max=74.98ms p(90)=11.34ms  p(95)=11.7ms
    iterations.....................: 600001 999.982065/s
    vus............................: 10     min=6        max=56
    vus_max........................: 400    min=400      max=400


running (10m00.0s), 0000/0400 VUs, 600001 complete and 0 interrupted iterations
high_rps ✓ [============================] 0000/0400 VUs  10m0s  1000.00 iters/s
pod "k6-front" deleted
```

Container Insights output



Monitor HPA

`kubectl get hpa -w`

```
NAME              REFERENCE                        TARGETS         MINPODS   MAXPODS   REPLICAS   AGE
makeline-service  Deployment/makeline-service      cpu: 1%/50%     1         2         1          137m
order-service     Deployment/order-service         cpu: 1%/50%     1         2         1          137m
product-service   Deployment/product-service       cpu: 1%/50%     1         2         1          137m
store-front       Deployment/store-front           cpu: 74%/50%    1         2         2          137m
```

Lab 2.7 – Stress test: Turn up Vegeta (constant-rate pressure)

Vegeta is great for **fixed RPS**. Here we send a steady stream of GETs to the front-end; you can also point it at internal services if you expose safe endpoints. The image usage below mirrors the author's Kubernetes examples.

What it is: A tiny HTTP load tester that sends requests at a fixed rate regardless of how slow the server gets.
Why it's useful: Great for capacity tests. If your app can't keep up, latency grows and errors appear, becomes very visible in platform metrics, Container Insights, and App Insights.
How it generates load: You give it a targets file (URLs + verbs). Vegeta's "attacker" emits requests at the configured rate, think of it as an external metronome. It doesn't wait for responses to schedule the next request.

Run this to delete, in case of previous stale jobs:

```
kubectl delete job vegeta-ramp-then-hold  --ignore-not-found
```

Run this to start off the load:

```
@'
apiVersion: batch/v1
kind: Job
metadata:
  name: vegeta-ramp-then-hold
  namespace: pets
spec:
  backoffLimit: 0
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: vegeta
        image: peterevans/vegeta:6.9.1
        command: ["/bin/sh","-lc"]
        args:
        - |
          printf "GET http://store-front.pets.svc.cluster.local/\n" > /tmp/t.txt
          rm -f /tmp/results.bin

          echo "--- Phase 1: ramp to 1000 rps (steps in 20s each) ---"
          for r in 600 800 1000; do
            echo "[Phase1] $r rps for 20s"
            vegeta attack -targets=/tmp/t.txt -timeout=10s -rate=$r -duration=20s \
              | tee -a /tmp/results.bin \
              | vegeta report -type=text -every=5s
          done

          echo "--- Phase 2: steady 1100 rps for 10m ---"
          vegeta attack -targets=/tmp/t.txt -timeout=10s -rate=1100 -duration=10m \
            | tee -a /tmp/results.bin \
            | vegeta report -type=text -every=5s

          echo "=== Final summary (all phases) ==="
          vegeta report < /tmp/results.bin
'@ | Out-File vegeta-ramp-then-hold.yaml -Encoding utf8

kubectl apply -f vegeta-ramp-then-hold.yaml
sleep 5
kubectl -n pets logs -f job/vegeta-ramp-then-hold

# USE ONLY IF PODS NOT DELETED
kubectl delete job vegeta-ramp-then-hold -n pets
```

Tip: If your cluster handles this easily, bump -rate again (e.g., 2000) or run **both** k6 and Vegeta simultaneously.

Vegeta tool run to max 1100 rps and concludes with 100% success rate. An option to see failure rates is to increase RPS beyond 1100 and reduce resource limits

```
Requests      [total, rate, throughput]         660001, 1100.00, 1100.00
Duration      [total, attack, wait]             10m0s, 10m0s, 236.782µs
Latencies     [min, mean, 50, 90, 95, 99, max]  136.584µs, 253.659µs, 225.085µs, 297.063µs, 356.972µs, 764.325µs, 20.638ms
Bytes In      [total, mean]                     292380443, 443.00
Bytes Out     [total, mean]                     0, 0.00
Success       [ratio]                           100.00%
Status Codes  [code:count]                      200:660001
Error Set:
=== Final summary (all phases) ===
2025/10/02 16:13:01 extra data in buffer
```

Overview Nodes Controllers **Containers**

Name	Status	95th % ↓	95th	Pod	Node	Restarts	UpTime	Trend 95th % (...
store-front	Unk	73%	145 mc	store-front-5f7fb4f575-qgfsl	aks-userpo...	0	2 hours	

Overview Nodes **Controllers** Containers

Search by name... Metric: CPU Usage (millicores) Min Avg 50th 90th 95th Max

Name	Status	95th % ↓	95th	Containers	Restarts	UpTime	Node	Trend 95th % (1 bar = 14m)
store-front-5f7fb4f575 (ReplicaSet)	2	42%	167 mc	2	0	5 mins	-	
store-front-5f7fb4f575-qgfsl	Ok	60%	120 mc	1	0	2 hours	aks-userpool-41046857-vmss00002u	
store-front	Ok	71%	142 mc	1	0	2 hours	aks-userpool-41046857-vmss00002u	
store-front-5f7fb4f575-8wk48	Ok	42%	85 mc	1	0	5 mins	aks-userpool-41046857-vmss00002u	
store-front	Ok	45%	91 mc	1	0	5 mins	aks-userpool-41046857-vmss00002u	

Lab 2.8 – Check Container Insights

This view shows you *something went wrong* (pods restarting, high CPU). View the below for clues with below customer changes:

- Set the **Time Range** for the period under test.
- Set the **Percentile view** to 95th CPU/Memory (only 5% of time did CPU or Mem exceed 100%), shows the worst-case outliers with performance issues. *This is where stress shows up*. If some pods are running hot while others are fine, the 95th jumps. This will show the bottlenecks. 90th (only 10% of the time did CPU exceed 95%) shows how most requests/pods behave under load.

1. **Overview tab**
    - **Cluster Summary**:
        - Watch **Node CPU / Memory** climb during the Vegeta phases (0→600, 800 rps).
        - In the *Namespace CPU/Memory* charts, the pets namespace should dominate.
    - **Logs by volume**: Spikes in store-front, order-service, makeline-service, and mongodb when load is highest.
2. **Nodes tab**
    - Look for any node >70-80% CPU or memory.
    - If the **Cluster Autoscaler** is enabled, you'll see **new nodes appear** in the list as the load grows.
3. **Controllers tab**
    - For workloads with HPAs (e.g., store-front, order-service) their **Replica counts** should rise as CPU hits the target threshold.
    - If limits are too low, you'll see **Restart counts** increasing for the ReplicaSets.
4. **Containers tab**
    - Sort by **95th percentile CPU %** or **Memory %**.
    - Expect to see store-front, order-service, makeline-service, and mongodb at the top.
    - If limits are tight:
        - CPU % pegged at 100%
        - Memory close to limit → restarts.
    - Restart counts >0 are a dead giveaway that OOMKilled or probe failures happened.
5. **Live Data (Logs / Events / Metrics)**
    - **Logs**: Flood of HTTP requests from Vegeta; 5xx errors if pods restart.
    - **Events**: Unhealthy, Killing with OOMKilled, or BackOff if CrashLooping.
    - **Metrics (kubectl top equivalent)**:
        - Pod CPU/memory should spike for store-front and order-service.
        - RabbitMQ shows spikes if the order queue backs up.

Expected progression during the Vegeta test

1. **Phase 1 (ramp 0→600 rps)**
    - Pod CPU rises in Containers tab.
    - Nodes are still under control.
    - No restarts yet.
2. **Phase 2 (steady 800 rps)**
    - Sustained high CPU for app pods.
    - If resource limits are tight, you start seeing throttling.
    - HPA may kick in: Replica count of store-front / order-service goes up.
3. **Phase 3 (800 → 1100 rps)**
    - Pods exceed CPU/memory limits → Restarts in Controllers/Containers tab.
    - Events show Killing / Unhealthy.
    - Node CPU climbs: Cluster Autoscaler may add nodes.
    - Logs show 5xx errors in Live Data.

Main point: Average (Avg) looks okay, but **95th/Max** reveal the hot pods that are failing.

Lab 2.9 – Check Metrics Explorer

This can be used to prove whether the *nodes* or *control plane* were under stress when the workloads (store-front, order-service, makeline-service, mongodb) failed.

API SERVER

- **API Server CPU Usage % / Memory Usage %**
  As load grew, the API server itself was fine - CPU stayed under 10%, so root cause wasn't control-plane saturation.

CLUSTER AUTOSCALER (Preview)

- **Cluster Health / Unneeded Nodes / UnSchedulable Pods**
  Shows whether the autoscaler detected pressure and added nodes.

ETCD

- **ETCD CPU / Memory Usage %**
  Important if your demo scales pods quickly; ETCD can choke, but even during the scale storm, ETCD remained healthy and there was no datastore bottleneck.

NODES

- **CPU Usage Percentage / CPU Usage Millicores**
  Core metric to show nodes hitting high % under Vegeta ramp.
- **Memory RSS % / Memory Working Set %**
  Expose memory pressure, especially when MongoDB is stressed.
- **Disk Used Bytes / IOPS % (if available)**
  This will show MongoDB pressure spilling to disk.
- **Network In/Out Bytes**
  Show spikes when virtual-customer floods order-service with traffic.

Insights:

- As vegeta hit 1200 rps, node CPU jumped to 85%. Memory Working Set also crossed 70%, confirming the pods were competing for tight limits. The autoscaler then added a node, visible as CPU % dropping across nodes.

PODS

- **Number of Pods by Phase / Ready state**
  Spikes in **Failed** or **Pending** pods align with OOMKills and FailedScheduling events.

Insights:

- Pods shifted to Pending because there was no room left on nodes until the autoscaler added capacity.
- Workload OOM (tight limits), not API server or ETCD.

Metric Output


```
k get hpa -w
NAME              REFERENCE                        TARGETS          MINPODS   MAXPODS   REPLICAS   AGE
makeline-service  Deployment/makeline-service      cpu: 1%/50%      1         2         1          137m
order-service     Deployment/order-service         cpu: 1%/50%      1         2         1          137m
product-service   Deployment/product-service       cpu: 1%/50%      1         2         1          137m
store-front       Deployment/store-front           cpu: 74%/50%     1         2         2          137m
store-front       Deployment/store-front           cpu: 81%/50%     1         2         2          138m
store-front       Deployment/store-front           cpu: 1%/50%      1         2         1          151m
```

Observed Metrics During Traffic Spikes

1. **CPU Usage %**
    - Average CPU usage climbs steadily (baseline ~8–10%) with sharp spikes during heavy traffic.
    - Suggests workloads are CPU-sensitive; sudden traffic surges drive high CPU demand.
2. **Disk Usage %**
    - Slowly rising (~9% → ~11%) with little fluctuation.
    - Disk isn't the immediate bottleneck, but sustained growth will require capacity planning.
3. **Network In Bytes**
    - Relatively stable until traffic surges, where it increases significantly.
    - Suggests input-heavy workloads (e.g., API calls, requests to pods).
4. **Network Out Bytes**
    - Shows sharp peaks during traffic surges (outgoing responses, data transfer).
    - Correlates strongly with CPU/network activity.

Likely Outcomes Under Heavy Traffic

- **Increased Latency:** Higher CPU demand may slow response times.
- **Potential Pod Restarts/Throttling:** If CPU limits are hit, AKS may throttle or evict pods.
- **Network Bottlenecks:** Spikes in Network Out could saturate bandwidth for node pools.
- **Scalability Concerns:** Without autoscaling, users may experience degraded performance or dropped requests.

Optimization & Mitigation Steps

1. Scaling Strategies

- Configure **Horizontal Pod Autoscaler** to scale pods based on CPU% and/or network I/O thresholds. Example: scale up when CPU > 60%.
- Enable **Cluster Autoscaler** on node pools to add/remove nodes automatically during high traffic.

2. Resource Management

- **Right-size Pod Requests/Limits** by reviewing CPU/memory requests and ensure pods have enough headroom without starving other workloads.
- **Separate System vs. App Workloads** using dedicated node pools for system services vs. customer workloads to reduce resource contention.

3. Networking Optimizations

- **Load Balancer Tuning** by using Azure Load Balancer or Application Gateway with autoscaling rules to distribute spikes evenly.
- **CNI Optimization:** If using Azure CNI, validate IP address capacity as heavy traffic may exhaust subnet IPs.
- **Caching & Compression**: Reduce outbound traffic volume by caching responses or compressing payloads where possible.

4. Observability & Alerts

- **Set Alerts:** CPU > 70% sustained; Network Out spikes > baseline threshold; Disk usage > 80%
- **Dashboards:** Track per-pod and per-node metrics to identify hotspots.

5. Long-term Improvements

- **Implement Traffic Shaping / Rate Limiting:** Protect cluster from sudden surges.
- **Leverage CDN / Edge Services:** Offload static content to Azure CDN to reduce backend load.
- **Service Mesh (e.g., Istio):** Use this AKS add-on to optimize routing and resilience under network load.

Lab 2.10 – Prometheus and Grafana Dashboards

When heavy traffic hits, **Grafana's out-of-the-box AKS dashboards** let you correlate spikes in **CPU, memory, network, and API server activity** at every layer. These give you deep pod-, workload-, and cluster-level visibility that complements Container Insights and Metrics Explorer.

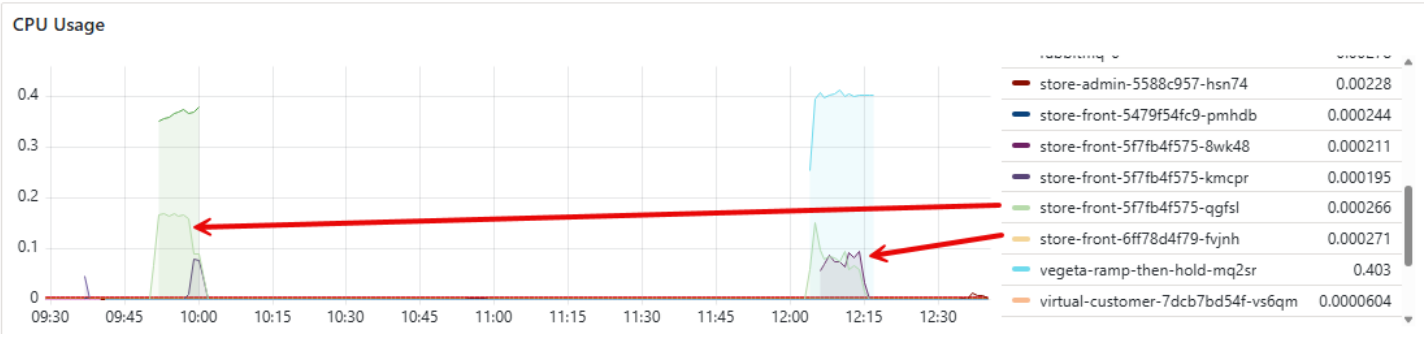
~~Turn on Advanced CNS on the cluster. Enable L7 if policies are in effect. Enable this configmap and follow steps 1-4, to visualize metrics in managed Grafana.~~

Kubernetes | API Server

- Good to prove control plane is *not* the bottleneck.
- Expected:
  - CPU/Memory for API server low and steady.
  - Latency for apiserver requests flat, confirming problem is at workload layer.

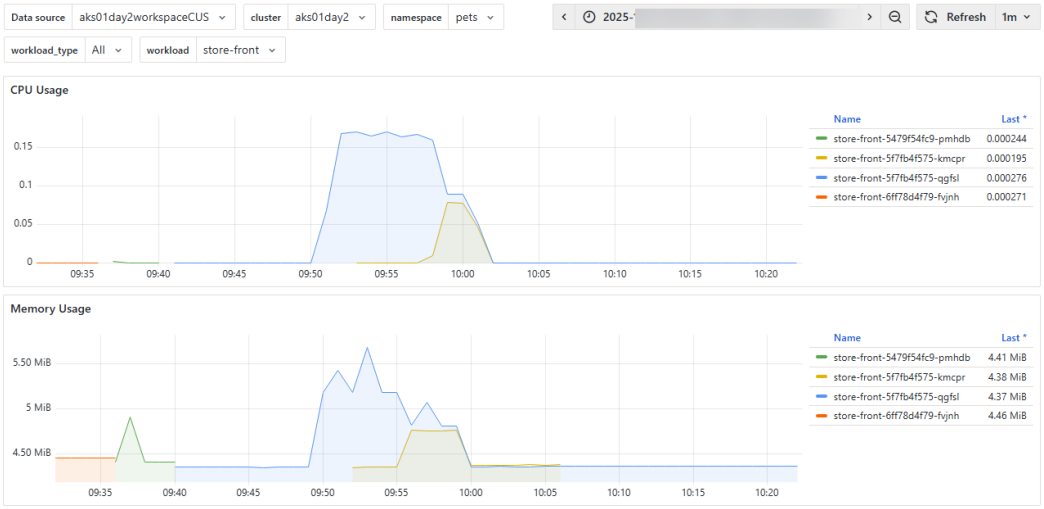
Kubernetes | Compute Resources | Namespace (Pods)

- Filter namespace="pets".
- View **CPU / Memory usage percentiles** per pod.
- Expected in your demo:
  - store-front, order-service, makeline-service → CPU pegged, memory climbing.
  - mongodb → memory spikes, possible OOM.



Kubernetes | Compute Resources | Workload

- Breaks down usage by Deployment/StatefulSet.
- Expected:
  - store-front service, order-service, makeline-service, mongodb, rabbitmq → CPU and memory climbs with load
  - HPA scaling visible as **replicas count** climbing during load.



Kubernetes | Networking | Pod Flows (Workload/Namespace)

- Show how much traffic Vegeta/k6 generate.
- Expected:
  - store-front → huge spike in incoming traffic.
  - Traffic flowing to order-service → queue → makeline-service → mongodb.
- If cluster is overloaded:
  - Look for **Drops** or **DNS errors** dashboards showing packet loss.

Data Source: aks01day2workspaceCUS | Cluster: aks01day2 | Nodes: All | Refresh

Namespace: pets | Dashboards: Network Observability | Documentation

NOTE: requires Advanced Container Networking Services (standard offering) to be enabled. See aka.ms/acns for more info.

Additionally, some panels below will be missing data when using the Minimum Ingestion Profile.

Top Namespaces

Top Source Namespaces (Outgoing Traffic)

Top Destination Namespaces (Incoming Traffic)

Namespace Snapshot (pets)

Pods with Outgoing Traffic (past 10 min...	Max Outgoing ...	Min Outgoing ...	Pods with Incoming Traffic (past 10 min...	Max Incoming ...	Min Incoming ...
9	296 p/s	1.66 p/s	5	94.0 p/s	500 mp/s

Pods with Outgoing Drops (past 1...	Max Outgoing ...	Min Outgoing ...	Pods with Incoming Drops (past 10...	Max Incoming ...	Min Incoming ...
No data	No data	No data	No data	No data	No data

Flows (in pets)

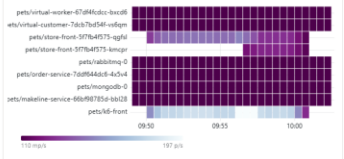
Outgoing Traffic by Trace Type

Incoming Traffic by Trace Type

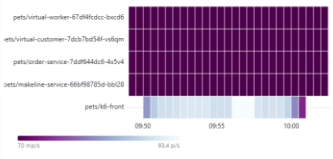
Outgoing Traffic by Verdict

Incoming Traffic by Verdict

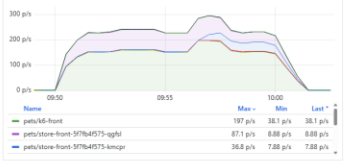
Heatmap of Outgoing Traffic for Top Source Pods



Heatmap of Incoming Traffic for Top Destination Pods



Stacked (Total) Outgoing Traffic by Source Pod

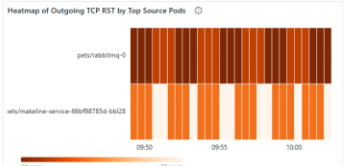


Stacked (Total) Incoming Traffic by Destination Pod

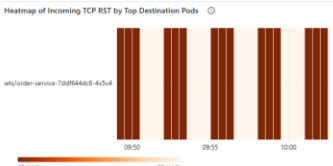
TCP (in pets)

Pods with Outgoing RST (past 1...	Max Outgoing RST Pac...	Min Outgoing ...	Pods with Incoming RST (past 1...	Max Incoming RST Pac...	Min Incoming ...
2	100 mp/s	70 mp/s	1	30 mp/s	20 mp/s

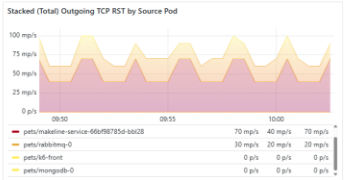
Heatmap of Outgoing TCP RST by Top Source Pods



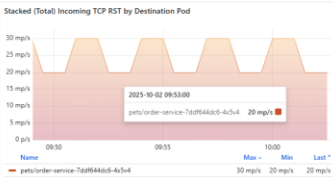
Heatmap of Incoming TCP RST by Top Destination Pods



Stacked (Total) Outgoing TCP RST by Source Pod

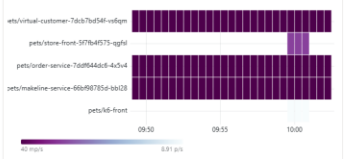


Stacked (Total) Incoming TCP RST by Destination Pod

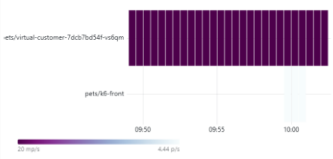


Pods with Outgoing FIN (past 1...	Max Outgoing FIN Pac...	MIN Outgoing ...	Pods with Incoming FIN (past 1...	Max Incoming FIN Pac...	Min Incoming ...
5	11.5 p/s	160 mp/s	2	4.48 p/s	20 mp/s

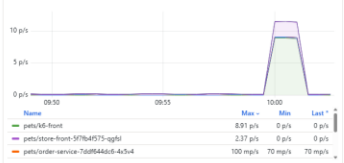
Heatmap of Outgoing TCP FIN by Top Source Pods



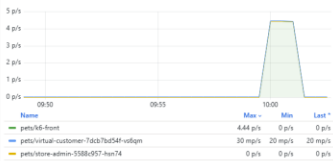
Heatmap of Incoming TCP FIN by Top Destination Pods



Stacked (Total) Outgoing TCP FIN by Source Pod



Stacked (Total) Incoming TCP FIN by Destination Pod



Namespace-Level Overview

- **Top Source Namespaces (Outgoing Traffic)**: p/s = packets/sec
  - Most traffic originates from kube-system (~35.9 p/s) — expected since system pods handle networking/DNS.
  - pets namespace contributes ~2.27 p/s, moderate relative to system components.
- **Top Destination Namespaces (Incoming Traffic)**:
  - kube-system is receiving ~7.89 p/s, suggesting lots of service discovery or API calls.
  - pets receives ~680 mp/s (much higher scale traffic), meaning intra-namespace traffic dominates.

Snapshot for pets Namespace

- **Outgoing traffic (past 10m)**:
  - 9 pods active, with a max of 296 p/s.
  - Min is very low (1.66 p/s), showing bursty patterns.
- **Incoming traffic (past 10m)**:
  - 5 pods receiving, with a max of 94 p/s and a huge min of 500 mp/s (likely sustained baseline).
- **Drops**: None recorded → suggests no packet loss, stable networking.

Traffic Characteristics

- **Flows by Trace Type**:
  - Clear segmentation between trace:to-endpoint and trace:to-stack.
  - Spikes are visible but remain within expected ranges (~100 p/s).
- **Verdict (Forwarded vs. Dropped)**:
  - Nearly all are **forwarded** (healthy cluster networking).
- **Heatmaps**:
  - Outgoing traffic is dominated by pets/store-front and pets/order-service.
  - Incoming mostly handled by pets/order-service and pets/makeline-service.
  - This validates the typical "frontend → order → makeline/product" call chain.

TCP-Level Signals

- **RST Packets**:
  - Outgoing RST spikes (up to 100 mp/s) from pets/rabbitmq and pets/makeline-service.
  - One pod (order-service) showing inbound RST (30 mp/s).
  - Frequent RSTs usually mean connections are being closed/reset rapidly (could be retries, connection churn, or misconfigured clients).
- **FIN Packets**:
  - Outgoing FIN ~11.5 p/s (normal TCP closes).
  - Incoming FIN ~4.48 p/s.
  - Distribution aligns with expected graceful connection termination, no abnormal spikes.

Key Observations

1. **Traffic Hotspots**
   - store-front is the main traffic source.
   - order-service and makeline-service are primary traffic sinks.
   - rabbitmq shows heavy TCP RST behavior → worth monitoring queue connection churn.
2. **No Drops**
   - No outgoing/incoming packet drops reported, network is stable. This is since "Pods with Outgoing Drops / Incoming Drops" panels both show **"No data"** for max/min values
3. **RST Spikes**
   - Outgoing RST near 100 mp/s is unusually high. Could indicate:
     - Clients disconnecting abruptly.
     - Application mismanaging connections.

        - RabbitMQ load/connection pressure.
4. **Steady FIN activity**
    - Graceful terminations happening in parallel → balanced with RST traffic.

Recommendations

- **Tune HPA / scaling for store-front**: high outgoing traffic → ensure backend services can scale proportionally.
- **Investigate RabbitMQ RST patterns**: check client connection pool configs, broker logs, and whether RST aligns with load test bursts.
- **Instrument order-service and makeline-service** deeper: high inbound traffic → validate resource requests/limits so pods don't throttle under load.
- **Baseline FIN/RST ratio**: If RST stays consistently high, may need to optimize connection reuse (keep-alive, pooling).

Lab 2.11 – PromQL pack

This **PromQL troubleshooting and observability pack** is a curated set of queries designed to quickly detect performance issues and failures in the **pets** namespace. You can run them directly inside in Azure Monitor Workspaces > Prometheus query explorer in the Azure portal.

The focus of this pack is triage and root-cause analysis:

- **Resource hotspots**: Identify pods with the highest CPU/memory usage.
- **Limits & throttling**: Show pods nearing CPU/memory limits (risk of throttling or OOMKill).
- **Pod health**: Detect restarts, OOMKills, or readiness failures.
- **Scaling & availability**: Cross-check HPA activity, deployment replica health, and endpoint readiness.
- **Network flows**: Spot top transmitters/receivers, correlating with connection resets or latency spikes.

This will help validate whether failures are due to resource pressure, scaling gaps, or network bottlenecks, and immediately tie them back to the right service (store-front, order-service, makeline-service, or mongodb).

Detect top offenders

Top CPU (cores) by pod (5m rate)

Shows the 10 pods using the most CPU (cores consumed averaged over 5 minutes). Expect `store-front`, `order-service`, or `makeline-service` at the top during heavy load.

```
topk(10,
  sum by (pod) (
    rate(container_cpu_usage_seconds_total{namespace="pets",container!=""}[5m])
  )
)
```

CPU % of limit by pod (shows throttling risk)

Calculates CPU usage as a percentage of each pod's CPU limit. Values near or above 100% mean pods are throttled and struggling.

```
topk(10,
  100 *
  sum by (pod) ( rate(container_cpu_usage_seconds_total{namespace="pets",container!=""}[5m]) )
  /
  sum by (pod) ( kube_pod_container_resource_limits{namespace="pets",resource="cpu",unit="core"} )
)
```

Top Memory (MiB) by pod

Lists the pods consuming the most working memory (in MiB). Expect `rabbitmq and mongodb` to show up here.

```
topk(10,
  sum by (pod) (container_memory_working_set_bytes{namespace="pets",container!=""}) / 1024 / 1024
)
```

Memory % of limit by pod (OOM risk)

Query shows the top 10 pods in the pets namespace ranked by how close their memory usage is to their memory limit. If values approach 100%, pods are at high risk of OOMKill.

```
topk(10,
  100 *
  sum by (pod) (container_memory_working_set_bytes{namespace="pets",container!=""})
  /
  sum by (pod) (kube_pod_container_resource_limits{namespace="pets",resource="memory",unit="byte"})
)
```

CPU throttling ratio (% of periods throttled)

Shows how often CPU usage was throttled vs allowed. High % = pods constrained by CPU limits.

```
topk(10,
  100 *
  sum by (pod) ( rate(container_cpu_cfs_throttled_periods_total{namespace="pets"}[5m]) )
  /
  sum by (pod) ( rate(container_cpu_cfs_periods_total{namespace="pets"}[5m]) )
)
```

Pod restarts in the last 10 minutes

Counts container restarts over the past 10 minutes. Expect spikes when pods OOMKill or crashloop.

```
topk(10,
  increase(kube_pod_container_status_restarts_total{namespace="pets"}[10m])
)
```

Pods currently OOMKilled (last termination reason)

Flags pods whose last exit was `OOMKilled`. Useful to pinpoint which services hit memory limits.

```
max by (pod) (
  kube_pod_container_status_last_terminated_reason{namespace="pets",reason="OOMKilled"} > 0
)
```

Focus on a specific service (swap the regex)

Use one of these regexes in the filters:

- `pod=~"store-front.*"`
- `pod=~"order-service.*"`
- `pod=~"makeline-service.*"`
- `pod=~"mongodb.*"`

Memory % of limit for order-service pods

Shows how close `order-service` pods are to their memory limits. >90% = near guaranteed OOMKill under load. Check with `"rabbitmq.*"`

```
100 *
sum by (pod) (container_memory_working_set_bytes{namespace="pets",pod=~"order-service.*"})
/
sum by (pod)
(kube_pod_container_resource_limits{namespace="pets",resource="memory",unit="byte",pod=~"order-
service.*"})
```

Restarts of makeline-service over last 15m

Counts how many times `makeline-service` pods restarted in the last 15 minutes. A rise confirms instability.

```
increase(kube_pod_container_status_restarts_total{namespace="pets",pod=~"makeline-service.*"}[15m])
```

CPU % of limit for mongodb

Shows mongodb CPU pressure relative to its limit. Useful to demo DB saturation leading to slow queries or OOM.

```
100 *
sum by (pod) ( rate(container_cpu_usage_seconds_total{namespace="pets",pod=~"mongodb.*"}[5m]) )
/
sum by (pod) (
kube_pod_container_resource_limits{namespace="pets",resource="cpu",unit="core",pod=~"mongodb.*"} )
```

Capacity / scheduling / endpoints

Pending / Failed pods by reason

This query shows you the number of pods in non-running states in the pets namespace, broken down by the failure reason.

```
sum by (reason) (kube_pod_status_reason{namespace="pets",reason!="Running"})
```

Below depends on HPA metrics availability

HPA: desired vs current replicas (all HPAs in ns)

Shows how many replicas the HPA wants vs how many are currently running. Great for proving autoscaler activity.

```
kube_horizontalpodautoscaler_status_desired_replicas{namespace="pets"}
kube_horizontalpodautoscaler_status_current_replicas{namespace="pets"}
```

Deployment availability (are enough replicas Serving?)

Net available pods per deployment. Drops highlight service degradation.

```
kube_deployment_status_replicas_available{namespace="pets"}
- kube_deployment_status_replicas_unavailable{namespace="pets"}
```

Service endpoints health (no ready backends = timeouts)

Lists services with pods not ready behind them. If nonzero, traffic may hit 502/connection reset.

```
sum by (service) (kube_endpoint_address_not_ready{namespace="pets"})
```

Network (useful when "connection reset / deadline exceeded" shows up)

Per-pod Rx bytes/sec (top receivers)

Shows which pods are receiving the most traffic. Expect `store-front` to dominate under Vegeta/k6.

```
topk(10, sum by (pod) (rate(container_network_receive_bytes_total{namespace="pets"}[1m])))
```

Per-pod RX bytes/sec (top transmitters)

Shows which pods are sending the most traffic. Expect `order-service` → RabbitMQ → `makeline-service` → `mongodb`.

```
topk(10, sum by (pod) (rate(container_network_transmit_bytes_total{namespace="pets"}[1m])))
```

Each query is **triage-focused**:

- **Top CPU/mem** = which pods are hottest.
- **% of limit & throttling** = why they're failing.
- **Restarts / OOMKilled** = proof of failure.
- **Deployments / endpoints** = availability impact.
- **Network** = correlates to connection resets/timeouts.

Triage Dashboard

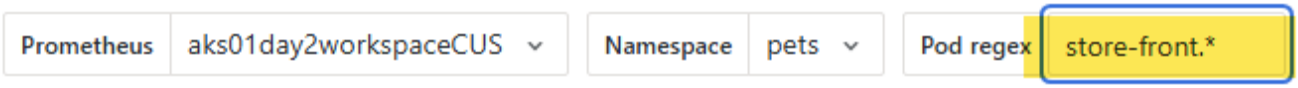
1. Save below JSON data to a file
2. Open up **Dashboards with Grafana** → New → Import → Upload JSON file → Load.

It's built for Azure Managed Prometheus (but works with any offering) and covers the following aspects:

- Top CPU / Memory (who's hottest)
- % of limit + CPU throttling (why failing)
- Restarts / OOMKilled (proof of failure)
- Deployments & Endpoints (availability impact)
- Network RX/TX (correlate with resets/timeouts)

To drop only certain pods into view, use below regex options. It defaults to .* (all pods). See highlight below.

- store-front.*
- order-service.*|makeline-service.*
- mongodb.*|order-service.*




```
{
  "title": "AKS Triage (Pods/Workloads) -  with Pod Regex",
  "tags": ["triage","aks","prometheus"],
  "timezone": "browser",
  "schemaVersion": 38,
  "version": 2,
  "templating": {
    "list": [
      {
        "name": "datasource",
        "type": "datasource",
        "query": "prometheus",
        "label": "Prometheus",
        "current": {},
        "hide": 0
      },
      {
        "name": "ns",
        "type": "query",
        "label": "Namespace",
        "datasource": "${datasource}",
        "query": "label_values(kube_pod_info, namespace)",
        "current": { "text": "pets", "value": "pets" },
        "includeAll": false,
        "refresh": 1
      },
      {
        "name": "podre",
        "type": "textbox",
        "label": "Pod regex",
        "query": ".*",
        "current": { "text": ".*", "value": ".*" }
      }
    ]
  },
  "panels": [
    { "type": "row", "title": "Top offenders (who is hottest?)", "collapsed": false, "gridPos":
{"h":1,"w":24,"x":0,"y":0} },

    {
      "type": "timeseries",
      "title": "Top CPU by Pod (cores, 5m rate)",
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "topk(10, sum by (pod) (
rate(container_cpu_usage_seconds_total{namespace=\"$ns\",container!=\"\",pod=~\"$podre\"}[5m]) ))",
          "legendFormat": "{{pod}}"
        }
      ],
      "gridPos": { "h":8, "w":12, "x":0, "y":1 }
```

```
    },
    {
      "type": "timeseries",
      "title": "Top Memory by Pod (MiB)",
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "topk(10, sum by (pod) (
container_memory_working_set_bytes{namespace=\"$ns\",container!=\"\",pod=~\"$podre\"} ) / 1024 / 1024)",
          "legendFormat": "{{pod}}"
        }
      ],
      "gridPos": { "h":8, "w":12, "x":12, "y":1 }
    },

    { "type": "row", "title": "% of limit & throttling (why they’re failing)", "collapsed": false, "gridPos":
{"h":1,"w":24,"x":0,"y":9} },

    {
      "type": "timeseries",
      "title": "CPU as % of Limit (per Pod)",
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "topk(10, 100 * sum by (pod) (
rate(container_cpu_usage_seconds_total{namespace=\"$ns\",container!=\"\",pod=~\"$podre\"}[5m]) ) / sum by (pod)
( kube_pod_container_resource_limits{namespace=\"$ns\",resource=\"cpu\",unit=\"core\",pod=~\"$podre\"} ))",
          "legendFormat": "{{pod}}"
        }
      ],
      "gridPos": { "h":8, "w":12, "x":0, "y":10 }
    },
    {
      "type": "timeseries",
      "title": "Memory as % of Limit (per Pod)",
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "topk(10, 100 * sum by (pod) (
container_memory_working_set_bytes{namespace=\"$ns\",container!=\"\",pod=~\"$podre\"} ) / sum by (pod) (
kube_pod_container_resource_limits{namespace=\"$ns\",resource=\"memory\",unit=\"byte\",pod=~\"$podre\"} ))",
          "legendFormat": "{{pod}}"
        }
      ],
      "gridPos": { "h":8, "w":12, "x":12, "y":10 }
    },
    {
      "type": "timeseries",
      "title": "CPU Throttling Ratio (% throttled periods)",
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "topk(10, 100 * sum by (pod) (
rate(container_cpu_cfs_throttled_periods_total{namespace=\"$ns\",pod=~\"$podre\"}[5m]) ) / sum by (pod) (
rate(container_cpu_cfs_periods_total{namespace=\"$ns\",pod=~\"$podre\"}[5m]) ))",
          "legendFormat": "{{pod}}"
        }
      ],
      "gridPos": { "h":8, "w":24, "x":0, "y":18 }
    },

    { "type": "row", "title": "Proof of failure (restarts / OOMKilled)", "collapsed": false, "gridPos":
{"h":1,"w":24,"x":0,"y":26} },

    {
      "type": "timeseries",
      "title": "Pod Restarts (increase over 10m)",
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "topk(10,
increase(kube_pod_container_status_restarts_total{namespace=\"$ns\",pod=~\"$podre\"}[10m]))",
          "legendFormat": "{{pod}}"
        }
      ],
      "gridPos": { "h":8, "w":12, "x":0, "y":27 }
    },
    {
      "type": "stat",
```

```
      "title": "Pods currently OOMKilled (last terminated reason)",
      "options": { "reduceOptions": { "calcs": ["lastNotNull"] }, "orientation": "horizontal" },
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "sum( max by (pod) (
kube_pod_container_status_last_terminated_reason{namespace=\"$ns\",reason=\"OOMKilled\",pod=~\"$podre\"} > 0 )
)"
        }
      ],
      "gridPos": { "h":8, "w":12, "x":12, "y":27 }
    },

    { "type": "row", "title": "Availability impact (deployments / endpoints)", "collapsed": false, "gridPos":
{"h":1,"w":24,"x":0,"y":35} },

    {
      "type": "timeseries",
      "title": "Deployments: Spec vs Available Replicas",
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "kube_deployment_spec_replicas{namespace=\"$ns\"}",
          "legendFormat": "{{deployment}} spec"
        },
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "kube_deployment_status_replicas_available{namespace=\"$ns\"}",
          "legendFormat": "{{deployment}} available"
        }
      ],
      "gridPos": { "h":8, "w":12, "x":0, "y":36 }
    },
    {
      "type": "timeseries",
      "title": "Services with Not-Ready Endpoints",
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "sum by (service) (kube_endpoint_address_not_ready{namespace=\"$ns\"})",
          "legendFormat": "{{service}}"
        }
      ],
      "gridPos": { "h":8, "w":12, "x":12, "y":36 }
    },

    { "type": "row", "title": "Network (correlate to resets/timeouts)", "collapsed": false, "gridPos":
{"h":1,"w":24,"x":0,"y":44} },

    {
      "type": "timeseries",
      "title": "Top RX Bytes/sec by Pod",
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "topk(10, sum by (pod) (
rate(container_network_receive_bytes_total{namespace=\"$ns\",pod=~\"$podre\"}[1m]) ))",
          "legendFormat": "{{pod}}"
        }
      ],
      "gridPos": { "h":8, "w":12, "x":0, "y":45 }
    },
    {
      "type": "timeseries",
      "title": "Top TX Bytes/sec by Pod",
      "targets": [
        {
          "datasource": { "type": "prometheus", "uid": "${datasource}" },
          "expr": "topk(10, sum by (pod) (
rate(container_network_transmit_bytes_total{namespace=\"$ns\",pod=~\"$podre\"}[1m]) ))",
          "legendFormat": "{{pod}}"
        }
      ],
      "gridPos": { "h":8, "w":12, "x":12, "y":45 }
    }
  ]
}
```

Dashboard is as seen below

Lab 2.12 – KQL Runbooks

Kusto Query Language (KQL) is the query language used in **Azure Monitor Logs** and **Log Analytics workspaces** for interactive analytics on large volumes of telemetry, such as metrics, logs, and events generated by your AKS clusters.

What's covered below is a triage-oriented runbook that you can paste into the Logs blade of your AKS workspace. They use the following tables in your Log Analytics workspace: **AzureActivity, AzureDiagnostics, ContainerLogV2, KubeEvents, KubePodInventory**

Health Snapshot - Pods Readiness & Blast Radius

The KQL Summarizes the latest state in **KubePodInventory**, of every pod and the target namespace and returns a single, compact row with readiness totals and quick context (data freshness, node spread, statuses seen). It also surfaces which pods are not ready and where they're running.

How the output helps triage

- **PodsTotal / PodsReady / PodsNotReady / NotReadyPct** → Instant signal of overall health and how bad the situation is.
- **LastSeen** → Confirms data is fresh; rules out "no data" false alarms.
- **NotReadyPods** → Quick list of impacted pods for targeting.
- **NodesTouched / NotReadyNodes** → Shows **blast radius** at the node level (is this localized or widespread?).
- **StatusesSeen** → Hints at the failure mode mix (e.g., CrashLoopBackOff, Pending, etc.). Also provides the distinct names of the associated pods.
- **NotReadyDetails** → Pod + status + node in one string for fast handoff and targeted follow-up (events/logs on those pods/nodes).

```
let ns = "pets"; let lookback = 1d;
KubePodInventory
| where TimeGenerated > ago(lookback) and Namespace == ns
| summarize arg_max(TimeGenerated, *) by Name
| extend NotReady = iif(PodStatus != "Running", 1, 0)
| summarize
    PodsTotal       = count(),
    PodsReady       = count() - sum(NotReady),
    PodsNotReady    = sum(NotReady),
    NotReadyPct     = round(100.0 * sum(NotReady) / count(), 2),
    LastSeen        = max(TimeGenerated),
    NodesTouched    = dcount(Computer),
    // collect pods per specific statuses
    TerminatingPods = make_set_if(Name, PodStatus == "Terminating", 100),
    RunningPods     = make_set_if(Name, PodStatus == "Running", 100),
    FailedPods      = make_set_if(Name, PodStatus == "Failed", 100),
    // usual not-ready helpers
    NotReadyPods    = make_set_if(Name, NotReady == 1, 100),
    NotReadyDetails = make_set_if(strcat(Name, " [", PodStatus, "] on ", tostring(Computer)), NotReady == 1,
50),
    NotReadyNodes   = make_set_if(Computer, NotReady == 1, 50)
| extend StatusesSeen = bag_merge(
        iif(array_length(TerminatingPods)>0, bag_pack("Terminating", TerminatingPods), dynamic({})),
        iif(array_length(RunningPods)>0,     bag_pack("Running",     RunningPods),     dynamic({})),
        iif(array_length(FailedPods)>0,      bag_pack("Failed",      FailedPods),      dynamic({}))
    )
| project-away TerminatingPods, RunningPods, FailedPods
```

PodsTotal	PodsReady	PodsNotReady	NotReadyPct	LastSeen [UTC]	NodesTouched
34	27	7	20.59	9/14/2025, 2:02:45.000 AM	3

PodsTotal	34
PodsReady	27
PodsNotReady	7
NotReadyPct	20.59
LastSeen [UTC]	2025-09-14T02:02:45Z
NodesTouched	3
NotReadyPods	["virtual-customer-f5d4cd9f7-c2bdd","virtual-worker-865bcdf78f-w89cm","http-client-7f7495fd4f-n5j5v","vegeta-ramp-then-hold-mvcbp","virtual-customer-b8566bc57-vxmfn","virtual-w
NotReadyDetails	["virtual-customer-f5d4cd9f7-c2bdd [Terminating] on aks-userpool-41046857-vmss00001q","virtual-worker-865bcdf78f-w89cm [Terminating] on aks-userpool-41046857-vmss00001q","ht
NotReadyNodes	["aks-userpool-41046857-vmss00001q","aks-userpool-41046857-vmss00001p"]
StatusesSeen	{"Terminating":["virtual-customer-f5d4cd9f7-c2bdd","virtual-worker-865bcdf78f-w89cm","http-client-7f7495fd4f-n5j5v","virtual-customer-b8566bc57-vxmfn","virtual-worker-65957df95c-

Failure Timeline by Reason (Pods & Counts)

Builds a 1-minute timeline of **KubeEvents** in the designated namespace, grouped by **Reason** (Killing, Unhealthy, BackOff, FailedScheduling). For each minute + reason, it shows:

- **Pods**: the distinct pod names impacted in that minute
- **PodCount**: how many distinct pods were hit

How the output helps triage

- **Spot spikes fast**: Minutes with high **PodCount** highlight bursts (e.g., widespread probe failures or scheduling issues).
- **Identify failure mode**: The **Reason** column points to the likely class of problem:
  - Killing often includes OOM terminations (check messages for "OOMKilled").
  - BackOff usually indicates crash loops.
  - Unhealthy surfaces liveness/readiness probe failures.
  - FailedScheduling indicates capacity/constraints (taints, quotas, node selectors).
- **See blast radius**: The **Pods** set shows exactly which pods were affected in each burst.
- **Correlate with changes**: Align spikes with deploys, autoscaler events, or infra maintenance.
- **Prioritize action**: Start with minutes where a single reason dominates and **PodCount** is high (e.g., probe failures across many pods → check readiness endpoints or recent config).

```
let ns = "pets"; let lookback = 1d;
KubeEvents
| where TimeGenerated > ago(lookback) and Namespace == ns and isnotempty(Name)
| where Reason in ("Killing","Unhealthy","BackOff","FailedScheduling")
| summarize
    Pods = make_set(Name, 200),      // distinct pods for this reason+minute
    PodCount = dcount(Name)          // how many pods hit
  by Reason, Minute = bin(TimeGenerated, 1m)
| order by Minute asc, Reason asc
```

Reason	Minute [UTC]	Pods	PodCount
FailedScheduling	9/13/2025, 1:19:00.000 PM	["store-front-6ff78d4f79-dmc...	11
Reason	FailedScheduling		
Minute [UTC]	2025-09-13T13:19:00Z		
Pods	["store-front-6ff78d4f79-dmc5z","http-server-866b9bbc75-rgbg9","virtual-customer-f5d4cd9f7-c2bdd'		
PodCount	11		
FailedScheduling	9/13/2025, 1:21:00.000 PM	["http-client-7f7495fd4f-n5j5v",...	11
BackOff	9/13/2025, 1:23:00.000 PM	["virtual-worker-865bcdf78f-w8...	1
Unhealthy	9/13/2025, 1:23:00.000 PM	["makeline-service-6c8ffb5857-...	2
Unhealthy	9/13/2025, 1:24:00.000 PM	["mongodb-0"]	1
Unhealthy	9/13/2025, 6:02:00.000 PM	["makeline-service-6c8ffb5857-...	1

Another option would be to list pods based on OOMKilled & CrashLoopBackOff events

```
let ns = "pets"; let lookback = 12h;
KubeEvents
| where TimeGenerated > ago(lookback) and Namespace == ns and isnotempty(Name)
| where (Reason == "Killing" and Message has "OOMKilled")
    or (Message has "Back-off restarting failed container")
    or (Reason == "Unhealthy")
| extend ReasonNorm = case(
        Reason == "Killing" and Message has "OOMKilled", "OOMKilled",
        Message has "Back-off restarting failed container", "CrashLoopBackOff",
        Reason == "Unhealthy", "ProbeFailed",
        Reason)
| summarize Reasons = make_set(ReasonNorm, 100), LastSeen = max(TimeGenerated) by Pod = Name
| order by LastSeen desc
```

Pod	Reasons	LastSeen [UTC]
makeline-service-6c8ffb5857-t5r...	["ProbeFailed","CrashLoopBac...	9/14/2025, 1:29:14.000 AM
Pod: makeline-service-6c8ffb5857-t5rw4		
Reasons: ["ProbeFailed","CrashLoopBackOff"]		
0: ProbeFailed		
1: CrashLoopBackOff		
LastSeen [UTC]: 2025-09-14T01:29:14Z		
mongodb-0	["ProbeFailed"]	9/14/2025, 12:54:26.000 AM
store-admin-5588c957-4n72j	["ProbeFailed"]	9/14/2025, 12:47:40.000 AM
store-front-6ff78d4f79-dmc5z	["ProbeFailed"]	9/14/2025, 12:47:38.000 AM
makeline-service-6c8ffb5857-4gdl4	["ProbeFailed","CrashLoopBack...	9/14/2025, 12:43:05.000 AM

Controller impact - which workloads are degraded

Lists controllers (Deployments/ReplicaSets/StatefulSets) with NotReady pods. The expected output would have Controllers ordered by number of affected pods.

```
let ns = "pets"; let lookback = 12h;
KubePodInventory
| where TimeGenerated > ago(lookback) and Namespace == ns
| summarize lastSeen = arg_max(TimeGenerated, *) by Name
| where PodStatus != "Running" or PodStatus != "Running"
| summarize NotReady=count() by ControllerKind, ControllerName
| order by NotReady desc
```

ControllerKind	ControllerName	NotReady
ReplicaSet	http-client-7f7495fd4f	1
ReplicaSet	virtual-customer-f5d4cd9f7	1
ReplicaSet	virtual-worker-865bcdf78f	1
ReplicaSet	virtual-customer-b8566bc57	1
ReplicaSet	virtual-worker-65957df95c	1
Job	vegeta-ramp-then-hold	1

App-level symptoms by checking Connection/Timeout Diagnostics in Logs

KQL scans **ContainerLogV2** in the designated namespace for common connectivity/latency failures, normalizes them into an ErrorType (e.g., ECONNRESET, connection reset, timeout, deadline exceeded, context deadline exceeded), and aggregates per (PodName, ContainerName, ErrorType). For each group it returns:

- **Count** - total matching log lines (hotspot intensity)
- **LastSeen** - most recent occurrence (is it active now?)
- **Samples** - up to 3 distinct example log lines to capture context

How the output helps triage

- **Surface the worst offenders first:** Sorted by Count, so you focus on the biggest impact pods/containers.
- **Check freshness:** LastSeen tells you if the issue is ongoing vs. historical.
- **Classify failure modes instantly:** ErrorType differentiates resets vs. timeouts vs. deadlines—guides which component to investigate (upstream service, network, client timeouts).
- **Jump-start root cause:** Samples provide concrete messages to search for correlation (endpoint names, error codes, paths).
- **Actionable next steps:**
  - Drill into a group: add | where PodName == "X" and ContainerName == "Y" and ErrorType == "timeout" then | project TimeGenerated, LogMessage.
  - Correlate with pod health: join with KubeEvents (probe failures, BackOff) or check restart counts in KubePodInventory.
  - Validate infra: if ECONNRESET spikes, check upstream readiness/load balancer; if timeout/deadline exceeded rises, review client timeouts and service latency.

Tips: Increase examples with make_set(LogMessage, N) (or use make_list to keep duplicates), widen/alter the keyword list to match your app's phrasing, and extend the window (lookback) to compare patterns over time.

```
// Consolidate groups by pod, container, and error type, showing the count (plus last seen and a
few sample messages).
// Count = number of matching lines per (PodName, ContainerName, ErrorType).
// Samples gives a few representative messages (bump the 3 if you want more).
let ns = "pets"; let lookback = 1d;
ContainerLogV2
| where TimeGenerated > ago(lookback) and PodNamespace == ns
| where isnotempty(PodName)
| where LogMessage has_any ("connection reset", "deadline exceeded", "timeout", "ECONNRESET",
"context deadline exceeded")
| extend ErrorType = case(
    LogMessage has "ECONNRESET",                    "ECONNRESET",
    LogMessage has "context deadline exceeded",     "context deadline exceeded",
    LogMessage has "deadline exceeded",             "deadline exceeded",
    LogMessage has "connection reset",              "connection reset",
    LogMessage has "timeout",                       "timeout",
    "other")
| summarize
    Count = count(),
    LastSeen = max(TimeGenerated),
    Samples = make_set(LogMessage, 3)   // up to 3 distinct examples per group
  by PodName, ContainerName, ErrorType
| order by Count desc, LastSeen desc
```

PodName	ContainerName	ErrorType	Count	LastSeen [UTC]	Samples
makeline-service-6c8ffb5857-4gdl4	makeline-service	context deadline exceeded	591	9/14/2025, 12:43:00.277 AM	["2025/09/13 17:23:15 no mor
PodName	makeline-service-6c8ffb5857-4gdl4				
ContainerName	makeline-service				
ErrorType	context deadline exceeded				
Count	591				
LastSeen [UTC]	2025-09-14T00:43:00.2773344Z				
Samples	["2025/09/13 17:23:15 no more orders for you: context deadline exceeded","2025/09/13 17:23:53 no more orders for you: context deadline exceeded","2025/09/13 17:24:31 no more orders for you				
makeline-service-6c8ffb5857-w9whq	makeline-service	context deadline exceeded	179	9/13/2025, 5:07:42.916 AM	["2025/09/13 03:14:23 no mor
makeline-service-6c8ffb5857-t5rw4	makeline-service	context deadline exceeded	115	9/14/2025, 3:04:39.868 AM	["2025/09/14 02:18:30 no mor
mongodb-0	mongodb	connection reset	21	9/14/2025, 2:56:59.274 AM	["2025-09-13T22:19:25.179+0
makeline-service-6c8ffb5857-t5rw4	makeline-service	timeout	4	9/14/2025, 12:49:29.993 AM	["2025/09/14 00:48:32 failed t

Per-Minute Event Impact by Reason

KQL builds a minute-by-minute timeline of **KubeEvents** in the designated namespace for the key failure reasons: Killing, Unhealthy, BackOff, and FailedScheduling. For each minute and reason it returns:

- **Pods** - the distinct pod names hit in that minute
- **PodCount** - how many distinct pods were impacted

How the output helps triage

- **Spot bursts fast:** Minutes with high **PodCount** flag active incidents or cascading failures.
- **Classify the failure mode:** The **Reason** shows whether you're dealing with OOM kills (Killing), probe issues (Unhealthy), crash loops (BackOff), or capacity/constraint problems (FailedScheduling).
- **Assess blast radius: Pods** reveals exactly which workloads were affected, helping you prioritize owners and services.
- **Correlate with changes:** Align spikes with deploys, autoscaler activity, or infrastructure events to find triggers.
- **Guide next steps:** Drill into hot minutes/reasons to pull detailed event messages or join with logs/restarts for root-cause analysis.

```
let ns = "pets"; let lookback = 1d;
KubeEvents
| where TimeGenerated > ago(lookback) and Namespace == ns and isnotempty(Name)
| where Reason in ("Killing","Unhealthy","BackOff","FailedScheduling")
| summarize
    Pods = make_set(Name, 200),      // distinct pods for this reason+minute
    PodCount = dcount(Name)          // how many pods hit
  by Reason, Minute = bin(TimeGenerated, 1m)
| order by Minute asc, Reason asc
```

Reason	Minute [UTC]	Pods	PodCount
FailedScheduling	9/13/2025, 1:19:00.000 PM	["store-front-6ff78d4f79-dmc...	11
Reason	FailedScheduling		
Minute [UTC]	2025-09-13T13:19:00Z		
Pods	["store-front-6ff78d4f79-dmc5z","http-server-866b9bbc75-rgbg9","virtual-customer-f5d4cd9f7-c2bdd","ma		
PodCount	11		
FailedScheduling	9/13/2025, 1:21:00.000 PM	["http-client-7f7495fd4f-n5j5v",...	11
BackOff	9/13/2025, 1:23:00.000 PM	["virtual-worker-865bcdf78f-w8...	1
Unhealthy	9/13/2025, 1:23:00.000 PM	["makeline-service-6c8ffb5857-...	2
Unhealthy	9/13/2025, 1:24:00.000 PM	["mongodb-0"]	1
Unhealthy	9/13/2025, 6:02:00.000 PM	["makeline-service-6c8ffb5857-...	1

Control-Plane Diagnostics Hotspots - API Server / Kubelet / Autoscaler (last 1d)

KQL scans **AzureDiagnostics** for control-plane categories (kube-apiserver, kubelet, cluster-autoscaler), classifies messages into Symptom buckets (Throttle/429, Timeout, ConnectionRefused, Error), and aggregates per (Symptom, Category, Resource):

- **Count** - number of matching diagnostics
- **LastSeen** - most recent occurrence
- **Instances** - up to *N* example lines (timestamp + message) for quick context

How the output helps triage

- **Prioritize quickly:** Sorted by **Count** and **LastSeen** so you see the noisiest and most recent control-plane issues first.
- **Pinpoint the layer:** 'Category' tells you whether it's API server, kubelet, or autoscaler; 'Resource' shows the impacted AKS resource.
- **Action by failure mode:**
  - **Throttle/429** → API QPS backpressure; check client rates, watcher fan-out, backoff.
  - **Timeout** → slow/blocked paths; investigate control-plane latency or networking.
  - **ConnectionRefused** → endpoints down/blocked; verify endpoint health, networking, or auth.
  - **Error** → generic failures worth deeper inspection.
- **Jump-start RCA:** 'Instances' provide concrete messages to pivot into detailed logs for the same Resource/Category and timeframe.
- **Validate observability:** If results are empty, it can indicate either a healthy period *or* missing AzureDiagnostics for your AKS resources and provides useful sanity check.

```
let lookback = 1d; let N = 5;
// Control-plane diagnostics categorized by symptom keywords, with up to N instances per group
AzureDiagnostics
| where TimeGenerated > ago(lookback)
| where Category in ("kube-apiserver","kubelet","cluster-autoscaler")
| extend Message = coalesce(column_ifexists("log_s",""), column_ifexists("rawlog_s",""),
column_ifexists("message",""))  // avoid properties_s
| where isnotempty(Message)
| extend Symptom = case(
    Message has_any ("429","Too Many Requests","throttle"), "Throttle/429",
    Message has "timeout",                                  "Timeout",
    Message has "connection refused",                       "ConnectionRefused",
    Message has "error",                                    "Error",
    "Other")
| where Symptom in ("Throttle/429","Timeout","ConnectionRefused","Error")
| summarize
    Count = count(),
    LastSeen = max(TimeGenerated),
    Instances = make_set(pack("Time", TimeGenerated, "Message", Message), 3)  // use make_list(...)
if you want duplicates
  by Symptom, Category, Resource
| order by Count desc, LastSeen desc
```

Symptom	Category	Resource	Count	LastSeen [UTC]	Instances
Timeout	kube-apiserver	AKS01DAY2	5897	9/14/2025, 1:01:03.445 AM	[{"Time":"2025-09-14T00:54:24
Symptom	Timeout				
Category	kube-apiserver				
Resource	AKS01DAY2				
Count	5897				
LastSeen [UTC]	2025-09-14T01:01:03.445268Z				
Instances	[{"Time":"2025-09-14T00:54:24.5139870Z","Message":"W0914 00:53:00.284481 1 dispatcher.go:210] Failed calling webhook, failing open mutation.gatekeeper.sh: failed calling webhook \"mut				
Error	kube-apiserver	AKS01DAY2	3528	9/14/2025, 1:03:02.975 AM	[{"Time":"2025-09-14T00:54:24.5
ConnectionRefused	kube-apiserver	AKS01DAY2	421	9/13/2025, 5:10:34.006 AM	[{"Time":"2025-09-13T05:10:33.5
Throttle/429	kube-apiserver	AKS01DAY2	67	9/13/2025, 1:42:57.060 PM	[{"Time":"2025-09-13T13:18:47.9

Az Monitor - Logs Workbook

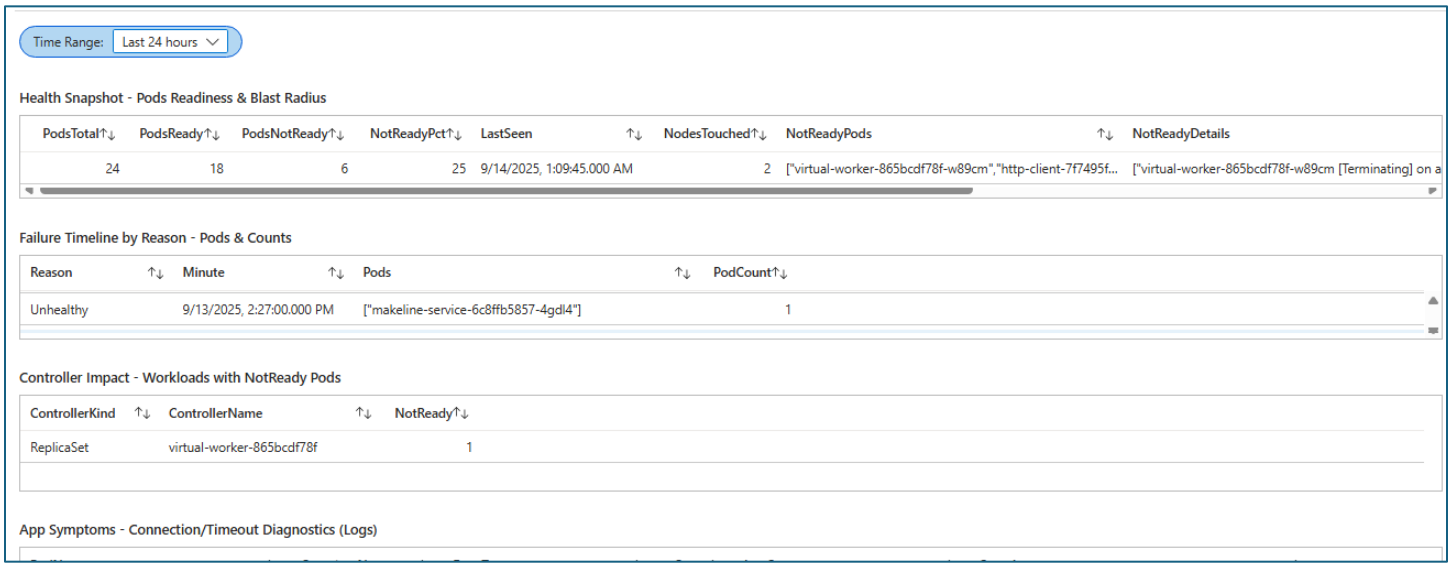
Import the below workbook JSON via **Azure Portal → Monitor → Workbooks → New → Advanced editor → paste → Apply → Save → Done Editing**.

It provides a fast triage view with:

- **Health Snapshot** (pods ready/not-ready with nodes & statuses),
- **Failure Timeline** (Killing/Unhealthy/BackOff/FailedScheduling per minute with pod counts),
- **Controller Impact** (controllers with NotReady pods),
- **App Symptoms (Logs)** (timeouts/resets grouped by Pod/Container/ErrorType with samples), and
- **Control-Plane Diagnostics** (apiserver/kubelet/autoscaler hotspots + drill-down).

Visible control: **TimeRange** (default 30m). Hidden workspace parameter **WS** is prefilled; namespace is set in each query as let ns = 'pets';.

Display



JSON

```
{
    "version": "Notebook/1.0",
    "name": "Kubernetes Triage - Single Pane",
    "items": [
        {
            "type": 9,
            "name": "parameters-0",
            "content": {
                "version": "KqlParameterItem/1.0",
                "parameters": [
                    {
                        "version": "KqlParameterItem/1.0",
                        "name": "TimeRange",
                        "label": "Time Range",
                        "type": 4,
                        "isRequired": true,
                        "value": {
                            "durationMs": 1800000
                        },
                        "typeSettings": {
                            "selectableValues": [
                                {
                                    "durationMs": 1800000
                                },
                                {
```

```
                                    "durationMs": 3600000
                                },
                                {
                                    "durationMs": 14400000
                                },
                                {
                                    "durationMs": 43200000
                                },
                                {
                                    "durationMs": 86400000
                                },
                                {
                                    "durationMs": 172800000
                                },
                                {
                                    "durationMs": 259200000
                                },
                                {
                                    "durationMs": 604800000
                                },
                                {
                                    "durationMs": 1209600000
                                },
                                {
                                    "durationMs": 2592000000
                                }
                            ],
                            "allowCustom": false
                        },
                        "id": "time-range-param",
                        "key": "time-range-param"
                    }
                ]
            },
            "styleSettings": {
                "w": 24,
                "h": 2,
                "x": 0,
                "y": 0,
                "showBorder": false,
                "borderStyle": "light thick"
            },
            "id": "parameters-block"
        },
        {
            "type": 3,
            "name": "Health Snapshot - Pods Readiness & Blast Radius",
            "content": {
                "version": "KqlItem/1.0",
                "title": "Health Snapshot - Pods Readiness & Blast Radius",
                "query": "let ns = 'pets';\nKubePodInventory\n| where Namespace == ns\n| summarize
arg_max(TimeGenerated, *) by Name\n| extend NotReady = iif(PodStatus != 'Running', 1, 0)\n|
summarize PodsTotal = count(), PodsReady = count() - sum(NotReady), PodsNotReady = sum(NotReady),
NotReadyPct = round(100.0 * sum(NotReady) / count(), 2), LastSeen = max(TimeGenerated),
NodesTouched = dcount(Computer), TerminatingPods = make_set_if(Name, PodStatus == 'Terminating',
100), RunningPods = make_set_if(Name, PodStatus == 'Running', 100), FailedPods = make_set_if(Name,
PodStatus == 'Failed', 100), NotReadyPods = make_set_if(Name, NotReady == 1, 100), NotReadyDetails
= make_set_if(strcat(Name, ' [', PodStatus, '] on ', tostring(Computer)), NotReady == 1, 50),
NotReadyNodes = make_set_if(Computer, NotReady == 1, 50)\n| extend StatusesSeen =
bag_merge(iif(array_length(TerminatingPods)>0, bag_pack('Terminating', TerminatingPods),
dynamic({})), iif(array_length(RunningPods)>0, bag_pack('Running', RunningPods), dynamic({})),
iif(array_length(FailedPods)>0, bag_pack('Failed', FailedPods), dynamic({})))\n| project-away
TerminatingPods, RunningPods, FailedPods",
                "queryType": 0,
                "resourceType": "microsoft.operationalinsights/workspaces",
                "crossComponentResources": [
                    "/subscriptions/463a82d4-1896-4332-aeeb-
618ee5a5aa93/resourcegroups/defaultresourcegroup-
cus/providers/microsoft.operationalinsights/workspaces/aks01day2lawcus"
                ],
```

```
                "timeContextFromParameter": "TimeRange",
                "visualization": "table",
                "size": 0
            },
            "styleSettings": {
                "showBorder": true,
                "borderStyle": "light thick",
                "w": 24,
                "h": 8,
                "x": 0,
                "y": 2
            },
            "id": "health-snapshot"
        },
        {
            "type": 3,
            "name": "Failure Timeline by Reason - Pods & Counts",
            "content": {
                "version": "KqlItem/1.0",
                "title": "Failure Timeline by Reason - Pods & Counts",
                "query": "let ns = 'pets';\nKubeEvents\n| where Namespace == ns and
isnotempty(Name)\n| where Reason in ('Killing','Unhealthy','BackOff','FailedScheduling')\n|
summarize Pods = make_set(Name, 200), PodCount = dcount(Name) by Reason, Minute =
bin(TimeGenerated, 1m)\n| order by Minute asc, Reason asc",
                "queryType": 0,
                "resourceType": "microsoft.operationalinsights/workspaces",
                "crossComponentResources": [
                    "/subscriptions/463a82d4-1896-4332-aeeb-
618ee5a5aa93/resourcegroups/defaultresourcegroup-
cus/providers/microsoft.operationalinsights/workspaces/aks01day2lawcus"
                ],
                "timeContextFromParameter": "TimeRange",
                "visualization": "table",
                "size": 0
            },
            "styleSettings": {
                "showBorder": true,
                "borderStyle": "light thick",
                "w": 24,
                "h": 8,
                "x": 0,
                "y": 10
            },
            "id": "failure-timeline"
        },
        {
            "type": 3,
            "name": "Controller Impact - Workloads with NotReady Pods",
            "content": {
                "version": "KqlItem/1.0",
                "title": "Controller Impact - Workloads with NotReady Pods",
                "query": "let ns = 'pets';\nKubePodInventory\n| where Namespace == ns\n| summarize
lastSeen = arg_max(TimeGenerated, *) by Name, ControllerKind, ControllerName\n| where PodStatus !=
'Running'\n| summarize NotReady = count() by ControllerKind, ControllerName\n| order by NotReady
desc",
                "queryType": 0,
                "resourceType": "microsoft.operationalinsights/workspaces",
                "crossComponentResources": [
                    "/subscriptions/463a82d4-1896-4332-aeeb-
618ee5a5aa93/resourcegroups/defaultresourcegroup-
cus/providers/microsoft.operationalinsights/workspaces/aks01day2lawcus"
                ],
                "timeContextFromParameter": "TimeRange",
                "visualization": "table",
                "size": 0
            },
            "styleSettings": {
                "showBorder": true,
                "borderStyle": "light thick",
                "w": 12,
```

```
                "h": 8,
                "x": 0,
                "y": 18
            },
            "id": "controller-impact"
        },
        {
            "type": 3,
            "name": "App Symptoms - Connection/Timeout Diagnostics (Logs)",
            "content": {
                "version": "KqlItem/1.0",
                "title": "App Symptoms - Connection/Timeout Diagnostics (Logs)",
                "query": "let ns = 'pets';\nContainerLogV2\n| where PodNamespace == ns\n| where
isnotempty(PodName)\n| where LogMessage has_any ('connection reset', 'deadline exceeded',
'timeout', 'ECONNRESET', 'context deadline exceeded')\n| extend ErrorType = case(LogMessage has
'ECONNRESET', 'ECONNRESET', LogMessage has 'context deadline exceeded', 'context deadline
exceeded', LogMessage has 'deadline exceeded', 'deadline exceeded', LogMessage has 'connection
reset', 'connection reset', LogMessage has 'timeout', 'timeout', 'other')\n| summarize Count =
count(), LastSeen = max(TimeGenerated), Samples = make_set(LogMessage, 3) by PodName,
ContainerName, ErrorType\n| order by Count desc, LastSeen desc",
                "queryType": 0,
                "resourceType": "microsoft.operationalinsights/workspaces",
                "crossComponentResources": [
                    "/subscriptions/463a82d4-1896-4332-aeeb-
618ee5a5aa93/resourcegroups/defaultresourcegroup-
cus/providers/microsoft.operationalinsights/workspaces/aks01day2lawcus"
                ],
                "timeContextFromParameter": "TimeRange",
                "visualization": "table",
                "size": 0
            },
            "styleSettings": {
                "showBorder": true,
                "borderStyle": "light thick",
                "w": 12,
                "h": 8,
                "x": 12,
                "y": 18
            },
            "id": "app-symptoms"
        },
        {
            "type": 3,
            "name": "Per-Minute Event Impact by Reason",
            "content": {
                "version": "KqlItem/1.0",
                "title": "Per-Minute Event Impact by Reason",
                "query": "let ns = 'pets';\nKubeEvents\n| where Namespace == ns and
isnotempty(Name)\n| where Reason in ('Killing','Unhealthy','BackOff','FailedScheduling')\n|
summarize Pods = make_set(Name, 200), PodCount = dcount(Name) by Reason, Minute =
bin(TimeGenerated, 1m)\n| order by Minute asc, Reason asc",
                "queryType": 0,
                "resourceType": "microsoft.operationalinsights/workspaces",
                "crossComponentResources": [
                    "/subscriptions/463a82d4-1896-4332-aeeb-
618ee5a5aa93/resourcegroups/defaultresourcegroup-
cus/providers/microsoft.operationalinsights/workspaces/aks01day2lawcus"
                ],
                "timeContextFromParameter": "TimeRange",
                "visualization": "table",
                "size": 0
            },
            "styleSettings": {
                "showBorder": true,
                "borderStyle": "light thick",
                "w": 24,
                "h": 8,
                "x": 0,
                "y": 26
            },
```

```
            "id": "per-minute-impact"
        },
        {
            "type": 3,
            "name": "Control-Plane Diagnostics Hotspots (Summary)",
            "content": {
                "version": "KqlItem/1.0",
                "title": "Control-Plane Diagnostics Hotspots (Summary)",
                "query": "let ns = 'pets';\nlet N = 5;\nAzureDiagnostics\n| where Category in
('kube-apiserver','kubelet','cluster-autoscaler')\n| extend Message =
coalesce(column_ifexists('log_s',''), column_ifexists('rawlog_s',''),
column_ifexists('message',''))\n| where isnotempty(Message)\n| extend Symptom = case(Message
has_any ('429','Too Many Requests','throttle'), 'Throttle/429', Message has 'timeout', 'Timeout',
Message has 'connection refused', 'ConnectionRefused', Message has 'error', 'Error', 'Other')\n|
where Symptom in ('Throttle/429','Timeout','ConnectionRefused','Error')\n| summarize Count =
count(), LastSeen = max(TimeGenerated), Instances = make_set(pack('Time', TimeGenerated, 'Message',
Message), N) by Symptom, Category, Resource\n| order by Count desc, LastSeen desc",
                "queryType": 0,
                "resourceType": "microsoft.operationalinsights/workspaces",
                "crossComponentResources": [
                    "/subscriptions/463a82d4-1896-4332-aeeb-
618ee5a5aa93/resourcegroups/defaultresourcegroup-
cus/providers/microsoft.operationalinsights/workspaces/aks01day2lawcus"
                ],
                "timeContextFromParameter": "TimeRange",
                "visualization": "table",
                "size": 0
            },
            "styleSettings": {
                "showBorder": true,
                "borderStyle": "light thick",
                "w": 24,
                "h": 8,
                "x": 0,
                "y": 34
            },
            "id": "control-plane-summary"
        },
        {
            "type": 3,
            "name": "Control-Plane Diagnostics — Samples (Drilldown)",
            "content": {
                "version": "KqlItem/1.0",
                "title": "Control-Plane Diagnostics — Samples (Drilldown)",
                "query": "let ns = 'pets';\nlet N = 5;\nAzureDiagnostics\n| where Category in
('kube-apiserver','kubelet','cluster-autoscaler')\n| extend Message =
coalesce(column_ifexists('log_s',''), column_ifexists('rawlog_s',''),
column_ifexists('message',''))\n| where isnotempty(Message)\n| extend Symptom = case(Message
has_any ('429','Too Many Requests','throttle'), 'Throttle/429', Message has 'timeout', 'Timeout',
Message has 'connection refused', 'ConnectionRefused', Message has 'error', 'Error', 'Other')\n|
where Symptom in ('Throttle/429','Timeout','ConnectionRefused','Error')\n| summarize Count =
count(), LastSeen = max(TimeGenerated), Instances = make_set(pack('Time', TimeGenerated, 'Message',
Message), N) by Symptom, Category, Resource\n| mv-expand Instances\n| extend SampleTime =
todatetime(Instances.Time), SampleMessage = tostring(Instances.Message)\n| project Symptom,
Category, Resource, SampleTime, SampleMessage\n| order by SampleTime desc",
                "queryType": 0,
                "resourceType": "microsoft.operationalinsights/workspaces",
                "crossComponentResources": [
                    "/subscriptions/463a82d4-1896-4332-aeeb-
618ee5a5aa93/resourcegroups/defaultresourcegroup-
cus/providers/microsoft.operationalinsights/workspaces/aks01day2lawcus"
                ],
                "timeContextFromParameter": "TimeRange",
                "visualization": "table",
                "size": 0
            },
            "styleSettings": {
                "showBorder": true,
                "borderStyle": "light thick",
                "w": 24,
```

```
                "h": 10,
                "x": 0,
                "y": 42
            },
            "id": "control-plane-drilldown"
        }
    ],
    "layout": {
        "type": "grid"
    },
    "styleSettings": {
        "background": "lighter"
    },
    "fallbackResourceIds": [
        "/subscriptions/463a82d4-1896-4332-aeeb-618ee5a5aa93/resourcegroups/defaultresourcegroup-
cus/providers/microsoft.operationalinsights/workspaces/aks01day2lawcus"
    ],
    "$schema": "https://github.com/Microsoft/Application-Insights-
Workbooks/blob/master/schema/workbook.json"
}
```

Activity Logs: AKS scale-out/scale-in (Cluster Autoscaler or manual)

You should see node pool scale events during/after heavy load if autoscaling is enabled. This is since Cluster Autoscaler triggers Compute/agent pool operations that land in Activity Log.

```
AzureActivity
| where TimeGenerated > ago(2h)
| where ResourceProviderValue has "Microsoft.Compute" or ResourceProviderValue has
"Microsoft.ContainerService"
| where OperationNameValue has_any ("virtualMachineScaleSets/write",
"virtualMachineScaleSets/scale","/agentPools/write", "/agentPools/scale")
| project TimeGenerated, OperationNameValue, ActivityStatusValue, _ResourceId, Caller,
CorrelationId
| order by TimeGenerated desc

# To result in numeric values for Chart display
AzureActivity
| where TimeGenerated > ago(2h)
| where ResourceProviderValue has "Microsoft.Compute" or ResourceProviderValue has
"Microsoft.ContainerService"
| where OperationNameValue has_any
("virtualMachineScaleSets/write","virtualMachineScaleSets/scale","agentPools/write","agentPools/sca
le")
| summarize Count = count() by bin(TimeGenerated, 5m), OperationNameValue
| order by TimeGenerated desc
```

Activity Logs: Who deleted/changed what (useful if you clean up between runs)

See successful deletes (e.g., removing a Public IP or AKS child resource).

```
AzureActivity
| where TimeGenerated > ago(7d)
| where ActivityStatusValue == "Success"
| where OperationNameValue contains "delete"
| extend Parts = split(_ResourceId, "/")
| project TimeGenerated, Caller, CallerIpAddress,
          DeletedResource = Parts[-1], DeletedType = Parts[-2],
          ResourceGroup
| order by TimeGenerated desc

# To result in numeric values for Chart display
# deletes over time by Caller
AzureActivity
| where TimeGenerated > ago(7d)
| where ActivityStatusValue == "Success"
| where OperationNameValue has "delete"
| extend Parts = split(_ResourceId, "/")
| extend DeletedResource = tostring(Parts[-1]), DeletedType = tostring(Parts[-2])
```

```
| summarize Deletes = count() by bin(TimeGenerated, 1h), Caller
| order by TimeGenerated asc

# deletes over time by DeletedType
AzureActivity
| where TimeGenerated > ago(7d)
| where ActivityStatusValue == "Success"
| where OperationNameValue has "delete"
| extend DeletedType = tostring(split(_ResourceId, "/")[-2])
| summarize Deletes = count() by bin(TimeGenerated, 1h), DeletedType
| order by TimeGenerated asc
```

Lab 2.13 – Creating Log based Alerts

1. Open the Query in Logs

- You're already in the **Logs blade** of your AKS workspace.
- Make sure your query is tested and returning the expected NotReady count (as in your screenshot).

2. New Alert Rule

1. At the top of the Logs query editor, click **New alert rule**.
2. Azure Monitor will carry your query into the **Create alert rule** workflow.

3. Define the Scope

- The scope is already set to your **Log Analytics Workspace** (the one linked to AKS).
- Confirm the subscription and resource group.

4. Configure the Condition

- In the **Condition** blade:
  - The signal type should be **Custom log search**.
  - Paste your KQL query (Azure will bring it in automatically).

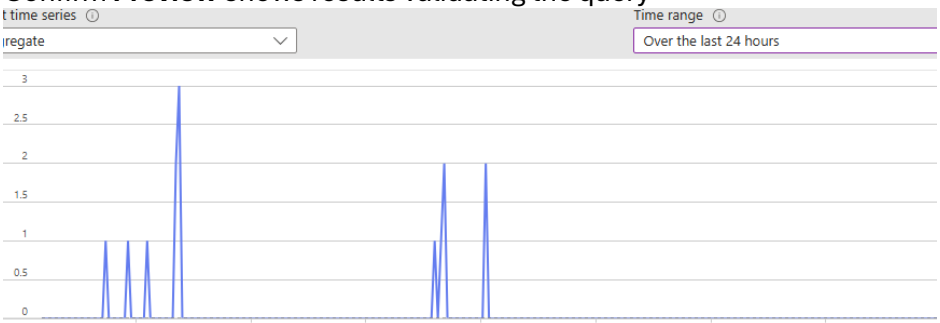
```
Activity Logs: AKS scale-out/scale-in
AzureActivity
| where TimeGenerated > ago(5m)
| where ResourceProviderValue has "Microsoft.Compute" or ResourceProviderValue has "Microsoft.ContainerService"
| where OperationNameValue has_any ("virtualMachineScaleSets/write",
"virtualMachineScaleSets/scale","/agentPools/write", "/agentPools/scale")
```

Under **Alert logic**

Example condition:

- Operator: **Greater than**
- Threshold value: **0**
- Aggregation granularity: **5 minutes**
- Frequency of evaluation: **5 minutes**

Confirm **Preview** shows results validating the query



Module 3 – Container Network Services

CNS is a suite of advanced networking features for AKS that deliver visibility, control, and security for containerized environments. CNS uses eBPF and Cilium to provide:

- **Observability (CNO):** real-time visibility.
- **Logs (CNLogs):** detailed traffic metadata.
- **Metrics (CNMetrics):** network health and performance.
- **Security (CNSecurity):** FQDN and L7-aware policies.

Lab 3.0 - Hubble CLI and UI Setup

Use the CLI as per link to install. Direct link. For Win11 use hubble-windows-amd64.tar.

Hubble CLI

1. Make sure that the Hubble pods are running:

```
kubectl get pods -o wide -n kube-system -l k8s-app=hubble-relay
# save output for later. E.g. hubble-relay-78bdfd99db-xv6qh
```

2. Ensure mTLS is being used for security of the Hubble Relay Server. Run below PS script to setup certs

```
$ErrorActionPreference = "Stop"

# Settings
$Ns         = "kube-system"
$Secret     = "hubble-relay-client-certs"
$CertDir    = Join-Path (Get-Location) ".certs"
$CertFiles = @{
  "tls.crt" = "tls-client-cert-file"
  "tls.key" = "tls-client-key-file"
  "ca.crt"  = "tls-ca-cert-files"
}

# Ensure output dir
if (-not (Test-Path $CertDir)) { New-Item -ItemType Directory -Path $CertDir | Out-Null }

# Pull the secret JSON once
$secretJson = kubectl get secret $Secret -n $Ns -o json
if (-not $secretJson) { throw "Secret '$Secret' not found in namespace '$Ns'." }
$secret = $secretJson | ConvertFrom-Json
if (-not $secret.data) { throw "Secret '$Secret' has no .data payload." }

# (Optional) show what keys are present
# Write-Host "Secret data keys: " ($secret.data.PSObject.Properties.Name -join ", ")

foreach ($fileName in $CertFiles.Keys) {
  $hubbleKey = $CertFiles[$fileName]

  # Grab the base64 string for this file (property names like 'tls.crt' include a dot)
  $prop = $secret.data.PSObject.Properties | Where-Object { $_.Name -eq $fileName }
  if (-not $prop) {
    Write-Warning "Secret '$Secret' is missing '$fileName'. Skipping."
    continue
  }
  $b64 = $prop.Value
  if ([string]::IsNullOrWhiteSpace($b64)) {
    Write-Warning "'$fileName' value is empty in secret '$Secret'. Skipping."
    continue
  }

  # Decode and write the exact bytes (no BOM, no newline transforms)
  $bytes   = [Convert]::FromBase64String($b64)
  $outFile = Join-Path $CertDir $fileName
  [IO.File]::WriteAllBytes($outFile, $bytes)

  # Configure hubble CLI
  & hubble config set $hubbleKey $outFile | Out-Host
  Write-Host "Wrote $outFile and set hubble key '$hubbleKey'."
}
```

```
hubble config set tls true
hubble config set tls-server-name instance.hubble-relay.cilium.io
```

3. Confirm that the secrets were generated as below should show 3 entries

```
kubectl get secrets -n kube-system | findstr hubble-
```

4. Port forward the Hubble Relay server:

```
kubectl port-forward -n kube-system svc/hubble-relay --address 127.0.0.1 4245:443
```

5. Verify that the Hubble Relay pod is running:

```
hubble observe --pod <hubble-relay-pod from step 1> --namespace kube-system --follow
e.g. hubble observe --pod store-front-5f7fb4f575-vrz5l--namespace pets --follow
```

```
                > hubble observe --pod hubble-relay-78bdfd99db-xv6qh --namespace kube-system --follow
Sep 18 02:36:27.026: kube-system/hubble-relay-78bdfd99db-xv6qh:45586 (ID:19810) -> 10.224.0.6:4244 (remote-node) to-stack FORWARDED (TCP Flags: ACK, PSH)
Sep 18 02:36:27.026: kube-system/hubble-relay-78bdfd99db-xv6qh:54354 (ID:19810) -> 10.224.0.5:4244 (remote-node) to-stack FORWARDED (TCP Flags: ACK, PSH)
Sep 18 02:36:27.026: kube-system/hubble-relay-78bdfd99db-xv6qh:34860 (ID:19810) -> 10.224.0.4:4244 (host) to-stack FORWARDED (TCP Flags: ACK, PSH)
Sep 18 02:36:27.026: kube-system/hubble-relay-78bdfd99db-xv6qh:45586 (ID:19810) <- 10.224.0.6:4244 (remote-node) to-endpoint FORWARDED (TCP Flags: ACK)
Sep 18 02:36:27.026: kube-system/hubble-relay-78bdfd99db-xv6qh:54354 (ID:19810) <- 10.224.0.5:4244 (remote-node) to-endpoint FORWARDED (TCP Flags: ACK)
```

Hubble UI

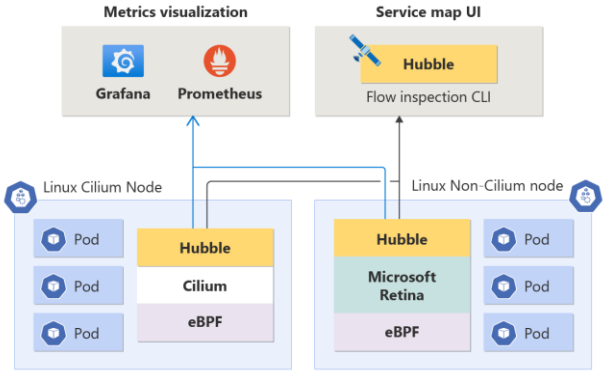
For Hubble UI, use this link to download YAML and apply. Use port forward and go to URL

```
kubectl -n kube-system port-forward svc/hubble-ui 12000:80
```

Go to http://localhost:12000/ and select Pets namespace

Lab 3.1 - Container Network Observability

- Provides insights into Network Traffic and Performance across containerized environments for both Cilium and non-Cilium data planes.
- Use eBPF to identify bottlenecks and congestion before applications are affected.
- Works with all CNI variants in Azure to provide node-level metrics and Hubble metrics for DNS.
- Integrates with Prometheus in Az Monitor for metrics storage and visualization.



Setup for Container Network Observability

```
az feature register --namespace "Microsoft.ContainerService" --name "AdvancedNetworkingFlowLogsPreview"
```

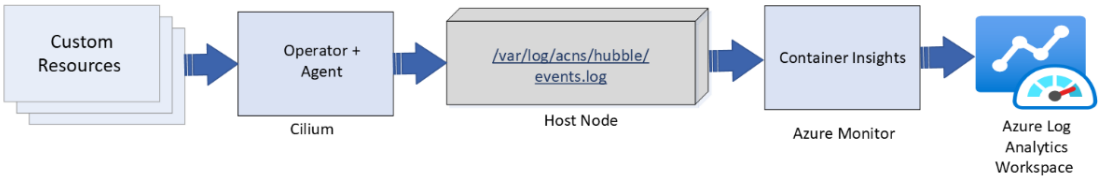
```
az aks update -g aks01day2-rg -n aks01day2 --enable-acns --acns-advanced-networkpolicies L7
```

```
az aks show -g aks01day2-rg -n aks01day2 --query "networkProfile.advancedNetworking"
```

```
  "networkProfile": {
    "advancedNetworking": {
      "enabled": true,
      "observability": {
        "enabled": true
      },
```

Lab 3.2 - Container Network Logs

- CNS uses eBPF technology with Cilium to fetch logs from nodes in the cluster.
- Cilium agent running on each node collects the network traffic that matches the criteria set in the custom resources. The logs are stored in JSON format on the host. If the Azure Monitoring add-on is enabled, agents for Container insights collect the logs from the host, apply the default throttling limits, and send them to a Log Analytics workspace.



- Logs capture essential metadata, including source and destination IP addresses, pod and service names, ports, protocols, and traffic direction for detailed insight into network behavior.
- Captures Layer 3 (IP), Layer 4 (TCP/UDP), and Layer 7 (HTTP/gRPC/Kafka) traffic to help you effectively monitor connectivity, troubleshoot, visualize network topology, and enforce security policy.
- **RetinaNetworkFlowLogs** is the table to run KQL to view logs. For Dashboards view **Networking-FlowLogs-ExternalTraffic**. Hubble CLI and UI allow to query, filter and analyze flow logs in the terminal.
- Needs Cilium data plane. L7 flow logs are only captured when **L7 policy** is enabled. DNS flow and metrics are only captured when a **FQDN policy** is enabled.

Steps for Container Network Logs enablement

Internal link

Configure an existing cluster to store logs in an existing Log Analytics workspace (LAW)

1. Disable monitoring addons (in prep for #2)

```
az feature register --namespace "Microsoft.ContainerService" --name "AdvancedNetworkingFlowLogsPreview"
az feature show --namespace "Microsoft.ContainerService" --name "AdvancedNetworkingL7PolicyPreview"
az aks disable-addons -a monitoring -g aks01day2-rg -n aks01day2
```

2. Enable for existing LAW

```
# Allows sending a much higher volume of logs/metrics to Log Analytics W without hitting throttling limits.

az aks enable-addons -a monitoring --enable-high-log-scale-mode -g aks01day2-rg -n aks01day2 --workspace-resource-id "/subscriptions/<sub-id>/resourceGroups/<resgrp-name>/providers/Microsoft.OperationalInsights/workspaces/<LAW-name>"
```

3. Update the AKS with the flag

```
az aks update -g aks01day2-rg -n aks01day2 --enable-acns --enable-retina-flow-logs
```

4. Validate that the Retina Network Flow Log capability is enabled

```
az aks show -g aks01day2-rg -n aks01day2 `
  --query "{observabilityEnabled:networkProfile.advancedNetworking.observability.enabled,
enableRetinaNetworkFlags:addonProfiles.omsagent.config.enableRetinaNetworkFlags,
logAnalyticsWorkspaceResourceID:addonProfiles.omsagent.config.logAnalyticsWorkspaceResourceID}" `
  -o json
```

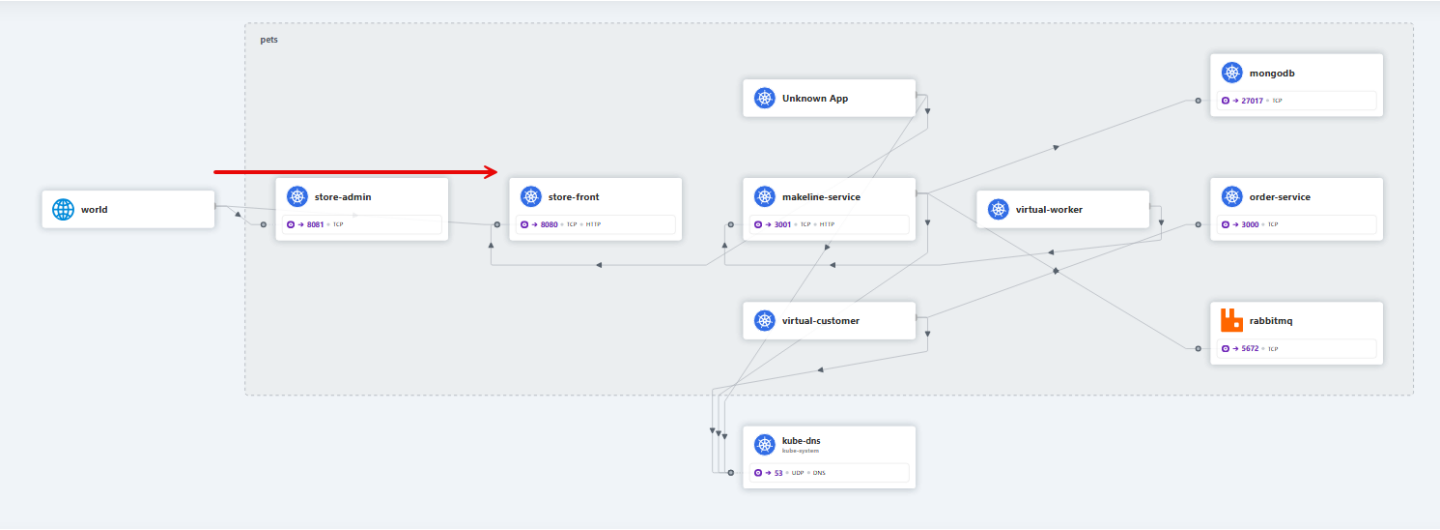
Result:

```
{
  "observabilityEnabled": true,
  "enableRetinaNetworkFlags": "True",
  "logAnalyticsWorkspaceResourceID": "/subscriptions/…/<lawName>"
}
```

6. Apply Retina policy file

To configure logging, you must define and apply the `RetinaNetworkFlowLog` type of Custom Resource (CR). You set filters like namespace, pod, service, port, protocol, and verdict. Multiple custom resources can exist in a cluster simultaneously.

Below shows the flow level logging in the Pets namespace and covers Pod to Pod communication. See section above on Hubble UI. It defines filters so you can capture traffic patterns between key services, (e.g., ingress from the internet to store-front, pod-to-pod flows like store-front ↔ makeline-service, virtual-worker → rabbitmq), core dependencies (all pets pods → kube-dns), and egress to external IPs.



- `Ingress from world → store-front`
- `store-front ↔ makeline-service`
- `virtual-worker → makeline-service`
- `makeline-service → mongodb`
- `virtual-worker → order-service`
- `virtual-worker → rabbitmq`
- `virtual-customer → store-front / makeline-service / virtual-worker`
- `All pets pods → kube-dns`
- `Pets pods → Unknown App (external)`

Run below commands to apply the Retina policy

```
kubectl apply -f .\retina-pets.yaml
```

Confirm policy deployed

```
kubectl get retinanetworkflowlogs
k describe retinanetworkflowlogs
```

retina-pets.yaml

```
apiVersion: acn.azure.com/v1alpha1
kind: RetinaNetworkFlowLog
metadata:
  name: pets-flowlog-compact
spec:
  includefilters:

    # World → store-front (ingress)
    - name: world-to-store-front
      from:
        ip: ["0.0.0.0/0"]
      to:
        labelSelector:
          matchLabels:
            app: store-front
            k8s.io/namespace: pets
      protocol: [tcp, udp]
      verdict: [forwarded, dropped]

    # store-front ↔ makeline-service
    - name: store-front-to-makeline-service
      from:
        labelSelector:
          matchLabels:
            app: store-front
            k8s.io/namespace: pets
      to:
        labelSelector:
          matchLabels:
            app: makeline-service
            k8s.io/namespace: pets
      protocol: [tcp, udp]
      verdict: [forwarded, dropped]

    - name: makeline-service-to-store-front
      from:
        labelSelector:
          matchLabels:
            app: makeline-service
            k8s.io/namespace: pets
      to:
        labelSelector:
          matchLabels:
            app: store-front
            k8s.io/namespace: pets
      protocol: [tcp, udp]
      verdict: [forwarded, dropped]

    # makeline-service → virtual-worker, mongodb
    - name: makeline-service-to-virtual-worker
      from:
        labelSelector:
          matchLabels:
            app: makeline-service
            k8s.io/namespace: pets
      to:
        labelSelector:
          matchLabels:
            app: virtual-worker
            k8s.io/namespace: pets
      protocol: [tcp, udp]
      verdict: [forwarded, dropped]

    - name: makeline-service-to-mongodb
      from:
        labelSelector:
          matchLabels:
            app: makeline-service
            k8s.io/namespace: pets
      to:
        labelSelector:
          matchLabels:
            app: mongodb
            k8s.io/namespace: pets
      protocol: [tcp]
      verdict: [forwarded, dropped]
```

```yaml
# virtual-worker → order-service, rabbitmq
- name: virtual-worker-to-order-service
  from:
    labelSelector:
      matchLabels:
        app: virtual-worker
        k8s.io/namespace: pets
  to:
    labelSelector:
      matchLabels:
        app: order-service
        k8s.io/namespace: pets
  protocol: [tcp]
  verdict: [forwarded, dropped]

- name: virtual-worker-to-rabbitmq
  from:
    labelSelector:
      matchLabels:
        app: virtual-worker
        k8s.io/namespace: pets
  to:
    labelSelector:
      matchLabels:
        app: rabbitmq
        k8s.io/namespace: pets
  protocol: [tcp]
  verdict: [forwarded, dropped]

# virtual-customer → any app in pets (covers its 3 arrows concisely)
- name: virtual-customer-to-pets
  from:
    labelSelector:
      matchLabels:
        app: virtual-customer
        k8s.io/namespace: pets
  to:
    labelSelector:
      matchLabels:
        k8s.io/namespace: pets
  protocol: [tcp, udp]
  verdict: [forwarded, dropped]

# DNS: all pets → kube-dns
- name: pets-to-kubedns
  from:
    labelSelector:
      matchLabels:
        k8s.io/namespace: pets
  to:
    labelSelector:
      matchLabels:
        k8s.io/namespace: kube-system
        k8s-app: kube-dns
  protocol: [udp, tcp, dns]
  verdict: [forwarded, dropped]

# External egress (Unknown App / world)
- name: pets-to-external
  from:
    labelSelector:
      matchLabels:
        k8s.io/namespace: pets
  to:
    ip: ["0.0.0.0/0"]
  protocol: [tcp, udp, dns]
  verdict: [forwarded, dropped]
```

Below provides a breakdown of each filter in the above RetinaNetworkFlowLog CR and what it logs.

1. **dns-to-coredns** logs egress DNS traffic from your DNS pod to CoreDNS to spot resolution failures, timeouts, or policy drops.
   - **From:** app=dns pod(s) in **pets**.
   - **To:** kube-dns (CoreDNS) in **kube-system**.
   - **Protocols/Verdicts:** UDP/TCP; forwarded & dropped.

2. **world-to-store-front** logs ingress traffic from the internet into the store-front service. Useful to monitor customer/user entry traffic.
   - **From:** any external IP (0.0.0.0/0).
   - **To:** pods labeled app=store-front in namespace pets.

3. **store-front-to-makeline-service** logs frontend → backend service calls, tracking how store-front communicates with makeline.
   - **From:** store-front pods in pets.
   - **To:** makeline-service pods in pets.

4. **makeline-service-to-store-front** captures reverse traffic from makeline back to store-front (likely status updates, callbacks).
   - **From:** makeline-service pods.
   - **To:** store-front pods.

5. **makeline-service-to-virtual-worker** logs flows between makeline and worker pods (work assignment, processing).
   - **From:** makeline-service.
   - **To:** virtual-worker.

6. **makeline-service-to-mongodb** tracks database access and logs makeline's queries to MongoDB.
   - **From:** makeline-service.
   - **To:** mongodb.

7. **virtual-worker-to-order-service** logs worker calls to order-service (e.g., order updates or fulfillment).
   - **From:** virtual-worker.
   - **To:** order-service.

8. **virtual-worker-to-rabbitmq** captures messaging interactions (workers publishing or consuming events).
   - **From:** virtual-worker.
   - **To:** rabbitmq.

9. **virtual-customer-to-pets** logs test or simulated customer traffic into all app components (store-front, makeline, worker).
   - **From:** virtual-customer pods.
   - **To:** any pod in pets.

10. **pets-to-kubedns** logs DNS queries from pets services to cluster DNS. Useful for debugging service discovery issues.
    - **From:** any pod in pets.
    - **To:** kube-dns in kube-system.

11. **pets-to-external** captures egress traffic leaving the namespace, e.g., calling external APIs or unknown destinations.
    - **From:** any pod in pets.
    - **To:** external IPs (0.0.0.0/0).

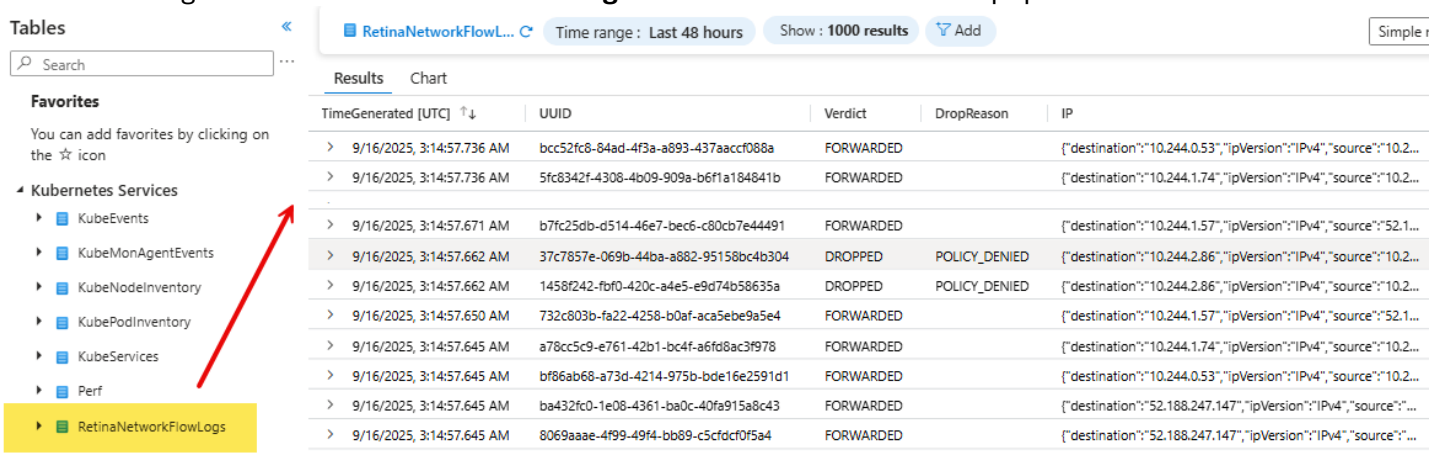
Each filter is a **traffic lens** - whether ingress from outside, inter-service flows, service-to-database/messaging, simulated customer load, DNS lookups, and egress. Together they give **full observability of the pets app's service mesh and external dependencies**.

7. Verify ConfigMap matches

`kubectl describe configmap acns-flowlog-config -n kube-system`

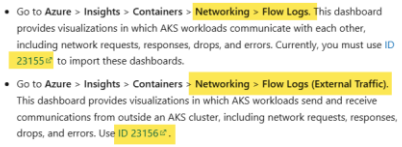
```
includeFilters:
- destination_label:
  - k8s.io/namespace=pets
  - app=store-front
  protocol:
  - tcp
  - udp
  source_ip:
  - 0.0.0.0/0
  verdict:
  - FORWARDED
  - DROPPED
```

8. Confirm if Logs > table **RetinaNetworkFlowLogs** exists and tables have been populated.



9. Confirm visualization

To visualize Flow Log traffic from AKS, use Dashboards w Grafana, specifically ID 23155 to import the Networking-FlowLogs dashboard. For more info, see this link.



Also confirm AMA pods are running, using `kubectl get pods -o wide -n kube-system | findstr ama-logs`


| Dashboards with Grafana (preview) | Networking-FlowLogs-ExtTraffic
Dashboards
Edit
Save As
Share
Export
Help
Feedback
Data Source
Azure Monitor
Subscription
ME-MngEnvMCAP993834-varghesejoji-1
Resource Group
aks01day2-rg
Cluster
aks01day2
Last 7 days
Refresh
Verdict
All
Protocol
All
External IP (optional)
Enter value
Direction
To
Namespace
All
Workload
All
Show IPs instead of 'world'
False
Dashboards: Network Observability
Documentation
Connection Graph
Total Flow Logs (requests and respo...
157 K
Requests (forwarded)
Total Flows
70.2 K
Unique Requests
58
Responses (forwarded)
Total Flows
75.4 K
Unique Responses
94
Dropped Requests
Total Request Flows
11.1 K
Unique Requests
53
DNS Response Errors
No data
HTTP Response Errors
No data
world
other-reserved
This graph displays each workload (e.g. Deployment) that sent traffic during the time period.
Each arrow draws a line from the requesting workload to the receiving workload. Arrows are color-coded:
Grey, solid line: forwarded requests (no drops, no DNS/HTTP response errors).
Red, dotted line: dropped requests (dropped by NetworkPolicy, etc.).
Orange, dashed line: requests which received DNS/HTTP response errors.
NOTE: in Grafana v10, both errors and drops are red/solid lines due to limitations.
Hover over a workload to highlight requests to/from that workload and to see the amount of flow logs for good requests, dropped requests, and response errors.
Flow Logs
All Flow Logs (first 1000)
Error Logs (first 1000)
Top Namespaces
Top Requests (forwarded)
Top Dropped Requests
Top DNS Response Errors
No data
Top HTTP Response Errors
No data
world -> default
world -> kube-system
world -> pets
world -> other-reserved
world -> pets
Top Workloads
Top Requests (forwarded)
Top Dropped Requests
Top DNS Response Errors
No data
Top HTTP Response Errors
No data
world -> default/nginx-ingress-ingress-nginx-controlle
world -> kube-system/coredns-6f776c8fb5
world -> pets/store-front-6ff78d4f79
world -> pets/vegeta-ramp-then-hold
world -> pets/virtual-worker-8650cdf78f
Top Workloads by Port/Query
Top Requests (TCP and/or UDP) (forwar...
Top Dropped Requests
Top DNS Response Errors
No data
Top HTTP Response Errors
No data
world -> default/nginx-ingress-ingress-nginx-controlle
world -> default/nginx-ingress-ingress-nginx-controlle
world -> pets/store-admin-5588c957-8d81
world -> pets/store-front-6ff78d4f79-b6b8d
Protocol Summary
Requests (forwarded)
Dropped Requests
DNS Responses by Error
An unexpected error happened
Details
HTTP Responses by Error
An unexpected error happened
Details
TCP
UDP
other
TCP
other

10. On completion delete all filters:

```
kubectl delete $(kubectl get retinanetworkflowlogs -o name)
```

Lab 3.3 - Container Network Metrics

- Metrics at node/pod level, traffic volume, dropped packets, connection states, DNS resolution.
- Deep visibility into network performance. Proactive Anomaly detection. Enhanced troubleshoot and optimization. Capacity planning and compliance.
- **Node-level metrics** are aggregated per node and include cluster or instance labels. These metrics provide insights into traffic volume, dropped packets, number of connections, and other data by node.
- **Pod-level metrics** include source/destination pod info for network-related issues at a granular level. Metrics cover information like traffic volume, dropped packets, TCP resets, and Layer 4/Layer 7 packet flows. DNS metrics like DNS errors and DNS requests missing responses
- Pod-level metrics are available only on Linux. DNS metrics are only available for pods with Cilium Network Policies applied or use Hubble CLI to view real-time logs.

Setup for CN Metrics

```
az aks update -g aks01day2-rg -n aks01day2 --enable-acns
```

Hubble metrics setup

The `hubble_flows_processed_total` metric isn't scraped by default due to high metric cardinality in large scale clusters. Because of this, the *Pods Flows* dashboards have panels with missing data. To enable this metric and populate the missing data, you need to do the following:

1. Download the ama-metrics-settings-configmap. See details in this link.
2. Locate the `networkobservabilityHubble = ""` and change it to `networkobservabilityHubble = "hubble.*"`
3. Confirm using this command to give resulting output.

   ```
   kubectl describe cm ama-metrics-settings-configmap -n kube-system | findstr networkobservabilityHubble
   ```

   ```
   networkobservabilityHubble = true
   networkobservabilityHubble = "hubble.*"
   networkobservabilityHubble = "30s"
   ```
4. The Pod flow metrics should now populate as this disables minimal ingestion.
5. Confirm if ama pods are running: `kubectl get pods -o wide -n kube-system | grep ama-`
6. From AKS > Dashboards w Grafana > confirm the below dashboards display data
   - **Networking | Clusters:** shows Node-level metrics for your clusters.
   - **Networking | DNS (Cluster):** shows DNS metrics on a cluster or selection of Nodes.
   - **Networking | DNS (Workload):** shows DNS metrics for the specified workload (for example, Pods of a DaemonSet or Deployment such as CoreDNS).
   - **Networking | Drops (Workload):** shows drops to/from the specified workload (for example, Pods of a Deployment or DaemonSet).
   - **Networking | Pod Flows (Namespace):** shows L4/L7 packet flows to/from the specified namespace (i.e. Pods in the Namespace).
   - **Networking | Pod Flows (Workload):** shows L4/L7 packet flows to/from the specified workload (for example, Pods of a Deployment or DaemonSet).

Lab 3.4 – Container Network Security

Securing container apps are based on **FQDN-based filtering** and **Layer 7 policy**. It uses Azure CNI Powered by Cilium DNS-based policies. **Layer 7 policy** allows granular control over application-level traffic by implementing policies based on protocols like HTTP, gRPC and Kafka.

Setup CNS for an existing cluster

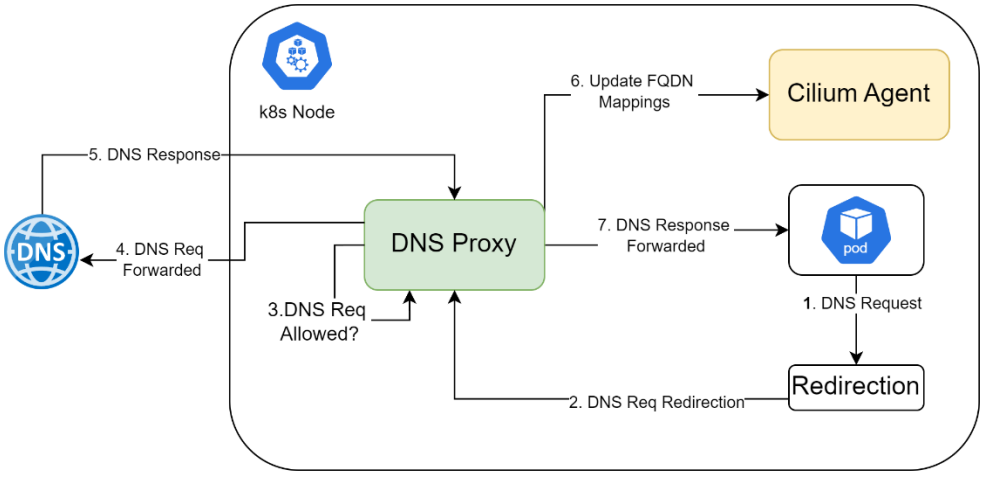
```
az extension update --name aks-preview
az feature register --namespace "Microsoft.ContainerService" --name "AdvancedNetworkingL7PolicyPreview"

# Use 'az feature show' to confirm
az aks update -g aks01day2-rg -n aks01day2 --enable-acns --acns-advanced-networkpolicies L7
```

FQDN-based filtering

Filtering is defined by policies based on domain names than IP addresses, which eliminates the need to update policies when an IP address changes. Components include:
Cilium Agent: runs as a DaemonSet within Azure CNI clusters powered by Cilium. It handles networking, load balancing, and network policies for pods in the cluster.
ACNS Security Agent: runs as DaemonSet in Azure CNI powered by Cilium cluster with Advanced Container Networking services enabled. It handles DNS resolution for pods and on successful DNS resolution, it updates Cilium Agent with FQDN to IP mappings.

When FQDN Filtering is enabled, DNS requests are first evaluated based on the network policy and if pods have access to specified domain names. The **Cilium Agent** marks DNS request packets originating from the pods enforcing FQDN policies, redirecting them to the **ACNS Security Agent**.

The ACNS Security Agent then decides whether to forward a DNS request to the DNS server based on the policy criteria. If permitted, the request is sent to the DNS server, and upon receiving the response, the ACNS Security Agent updates the Cilium Agent with FQDN mappings.



Setup for FQDN-based filtering

ACNS should be enabled

```
az aks update -g aks01day2-rg -n aks01day2 --enable-acns
```

Enforce a policy: In the case below DNS requests are made to an allowed FQDN (bing.com:80) and anything else (example.com:80) is blocked.

matchPattern is mandatory, without which Cilium only allows the DNS traffic and not inspect its contents to learn which IPs are associated with the FQDNs. Hence, connections to those IPs (non-DNS traffic) are blocked because Cilium can't associate them with the allowed domain.

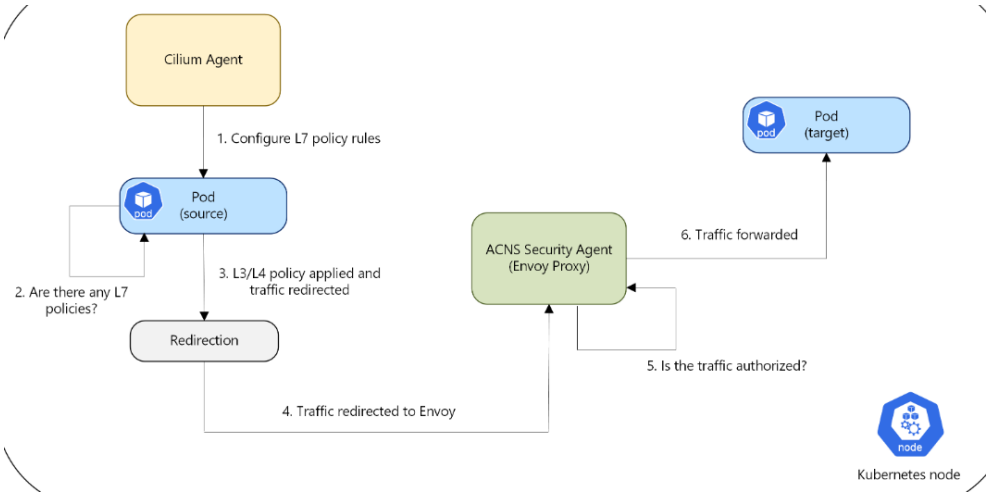
```
apiVersion: "cilium.io/v2"
kind: CiliumNetworkPolicy
metadata:
  name: "allow-bing-fqdn"
spec:
  endpointSelector:
    matchLabels:
      app: demo-container
  egress:
    - toEndpoints:
      - matchLabels:
          "k8s:io.kubernetes.pod.namespace": kube-system
          "k8s:k8s-app": kube-dns
      toPorts:
        - ports:
           - port: "53"
             protocol: ANY
          rules:
            dns:
              - matchPattern: "*.bing.com"
    - toFQDNs:
      - matchPattern: "*.bing.com"
```

L7 Policies

L7 policy support enables detailed inspection and management of application-level traffic. It uses Envoy proxy (separate DaemonSet) to inspect application traffic, comparing it against defined L7 policies.

Outgoing network traffic is first evaluated to determine compliance with the configured application-level rules. The eBPF probe attached to the source pod's network interface marks the packets, which are then redirected to a node-local Envoy Proxy. This redirection occurs only for **pods enforcing L7 policies.**

The Envoy proxy decides whether to forward the traffic to the destination pod based on policy criteria. If permitted, the traffic proceeds; if not, Envoy returns an appropriate error code to the originating pod. Upon successful authorization, the Envoy proxy facilitates the traffic flow, providing application-level visibility and control. This allows the Cilium agent to enforce detailed network policies within the policy engine. The following diagram illustrates the high-level flow of L7 policy enforcement.



Dashboards: "Kubernetes/**Networking/L7 (Namespace)**" and "Kubernetes/**Networking/L7 (Workload)**". Data only displayed if ACNS feature is enabled on your cluster and have relevant policies applied.

Setup for L7 Policies

```
az aks update -g aks01day2-rg -n aks01day2 --enable-acns --acns-advanced-networkpolicies L7
```

Setup validation for L7 Policies

```
az aks show -g aks01day2-rg -n aks01day2 `
  --query "{acns:networkProfile.advancedNetworking.enabled,
acnsPolicy:networkProfile.advancedNetworking.security.advancedNetworkPolicies}" `
  -o jsonc

{
  "acns": true,
  "acnsPolicy": "L7"
}
```

Policy test

1. Apply below L7 policy that block requests to /health

pets-l7-policy.yaml

```
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: store-front-ingress-http
  namespace: pets
spec:
  endpointSelector:
    matchLabels: { app: store-front }
  ingress:
    - fromEntities: [cluster, world, remote-node, host]
      toPorts:
        - ports: [{ port: "8080", protocol: TCP }]
          rules:
            http:
              - { method: "GET", path: "/" }
              # - { method: "GET", path: "/health" }
```

2. Run the below to hubble trace on the store-front pod

```
kubectl port-forward -n kube-system svc/hubble-relay --address 127.0.0.1 4245:443
kubectl get pod -l app=store-front
hubble observe --pod <store-front pod name> --namespace pets --follow
```

3. Run the curl test that makes requests to /health

```
kubectl run curl-once -n pets --rm -i --restart=Never --image=curlimages/curl:8.10.1 `
  -- curl -s -o /dev/null -w "%{http_code}\n" http://store-front:8080/health
```

4. The flow trace using 'hubble observe' reveals request drops to /health

5. Uncommenting last line (`{ method: "GET", path: "/health" }`) allows http requests to /health, since the Envoy proxy does not explicitly allow /health traffic through without explicitly stating it.

7. Delete all Cilium Network Policies when done

```
kubectl delete $(kubectl get cnp -o name)
```

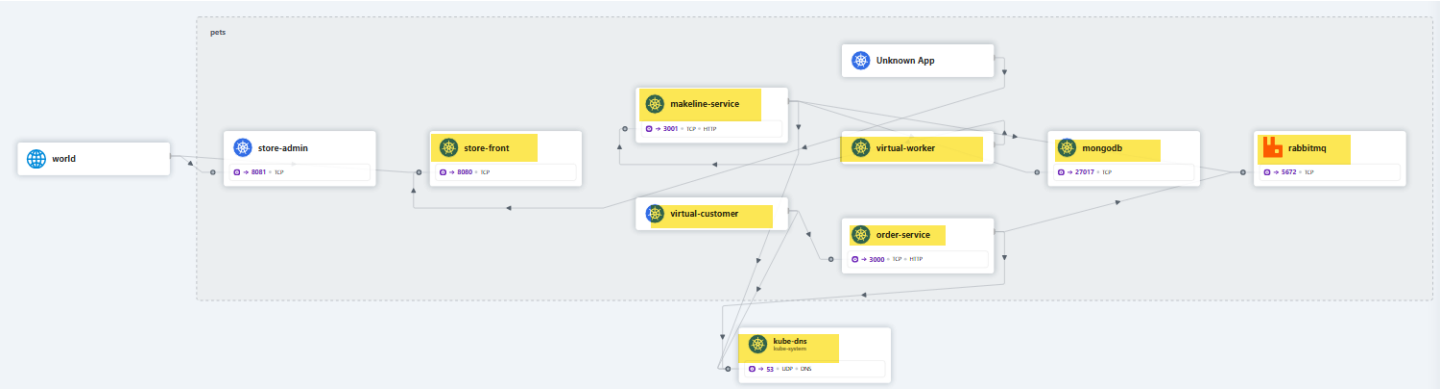
8. If store-front doesn't recover scale down and up

```
k scale deploy store-front --replicas=0
k scale deploy store-front --replicas=1
```

Module 4 – CNS Policies applied to Pets Application

Lab 4.0 – Application Layout

Below is the Hubble UI graph showing **Pets** namespace



1. Ingress from the Outside World
These are your external entry points. Traffic from outside the cluster (current traffic is using Vegeta load generator) reaches your **store-front** and **store-admin** services directly.

- **World → store-front (8080/TCP)**
- **World → store-admin (8081/TCP)**

2. Frontend to Mid-tier
The store-front acts as the customer-facing frontend. It sends API calls downstream to makeline-service and order-service.

- **store-front → makeline-service (3001/TCP)**
- **store-front → order-service (3000/TCP)**

3. Virtual Clients (Job Pods)
Both these jobs behave like workload simulators and drive the order pipeline.

- **virtual-customer → order-service (3000/TCP)**
  - Simulates real customers placing orders.
- **virtual-worker → makeline-service (3001/TCP)**
  - Simulates backend workers fulfilling/making orders.

4. Makeline-Service Orchestration
Makeline orchestrates orders: Talks to **order-service** for order metadata and persists data in **mongodb**. It also publishes events/messages to **rabbitmq**.

- **makeline-service → order-service (3000/TCP)**
- **makeline-service → mongodb (27017/TCP)**
- **makeline-service → rabbitmq (5672/TCP)**

5. Order-Service Dependencies
Order-service stores order state in **mongodb** and also publishes/consumes events via **rabbitmq**.

- **order-service → mongodb (27017/TCP)**
- **order-service → rabbitmq (5672/TCP)**

6. Shared Backends
These don't initiate connections, as they only receive traffic from order-service and makeline-service.

- **mongodb** and **rabbitmq** sit at the back as shared infrastructure for persistence and messaging.

7. DNS Resolution
Every pod in pets uses **CoreDNS** for resolving service FQDNs (e.g., **order-service.pets.svc.cluster.local**). That's why you see those multiple lines going to kube-dns.

- Many arrows from workloads (virtual-customer, makeline-service, order-service, etc.) → **kube-dns (UDP/53)**

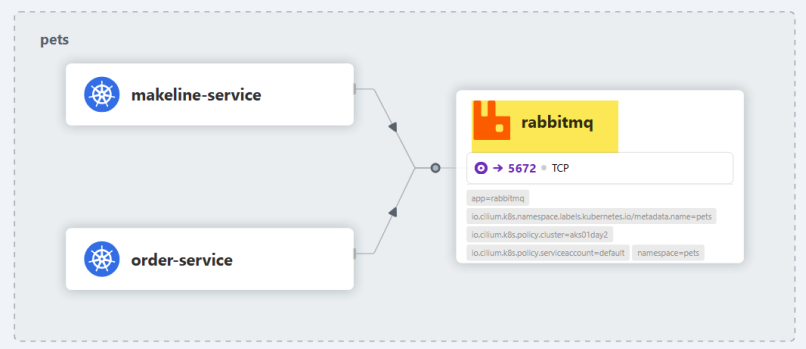
8. Unknown App

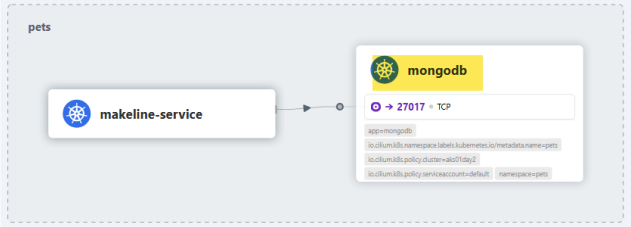
- Hubble detected an "Unknown App" pod in the namespace sending traffic.
- Because it lacks recognizable labels (app=...), Hubble can't map it to a known workload.

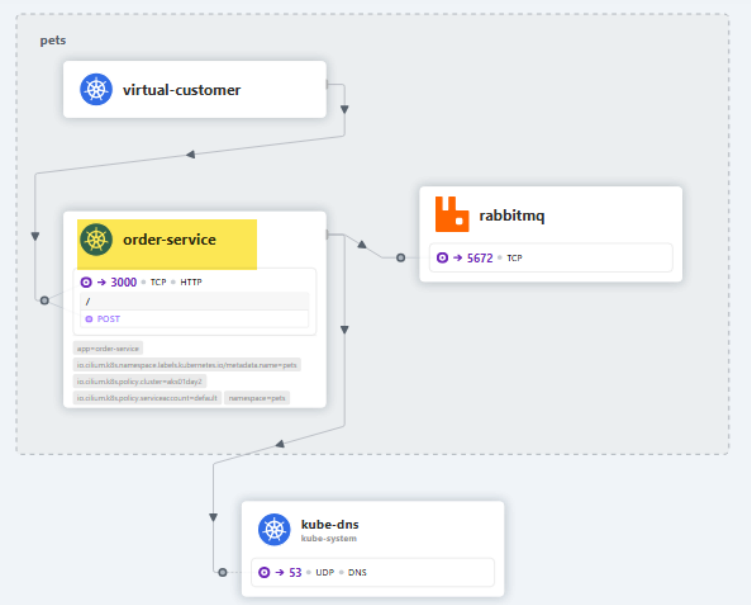
Summary of flow

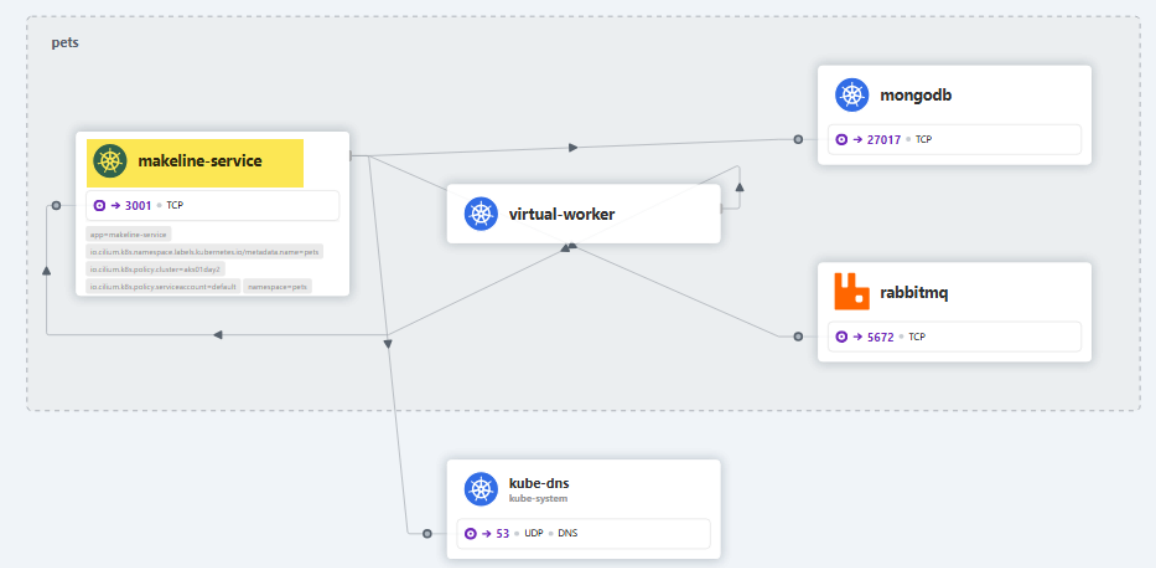
1. **World → store-front / store-admin** (entry).
2. **Store-front → makeline-service + order-service** (frontend → API).
3. **Virtual jobs → makeline-service / order-service** (simulated clients).
4. **Makeline + order-service → mongodb + RabbitMQ** (persistence/messaging).
5. **All pods → kube-dns** for DNS resolution.
6. **Unknown App** also doing lookups or calling into services.

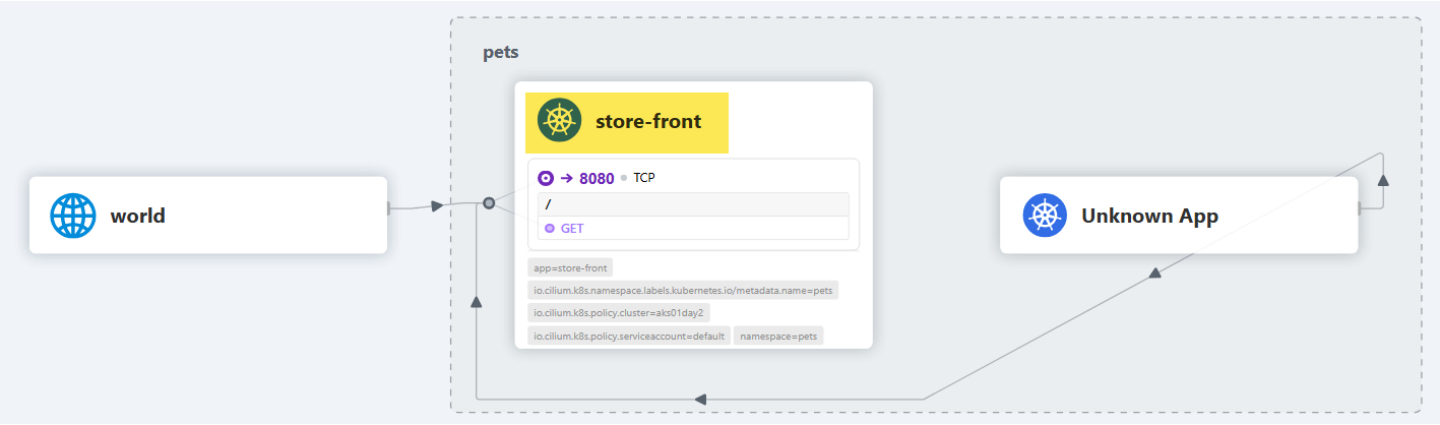
Detailed Flows for each service

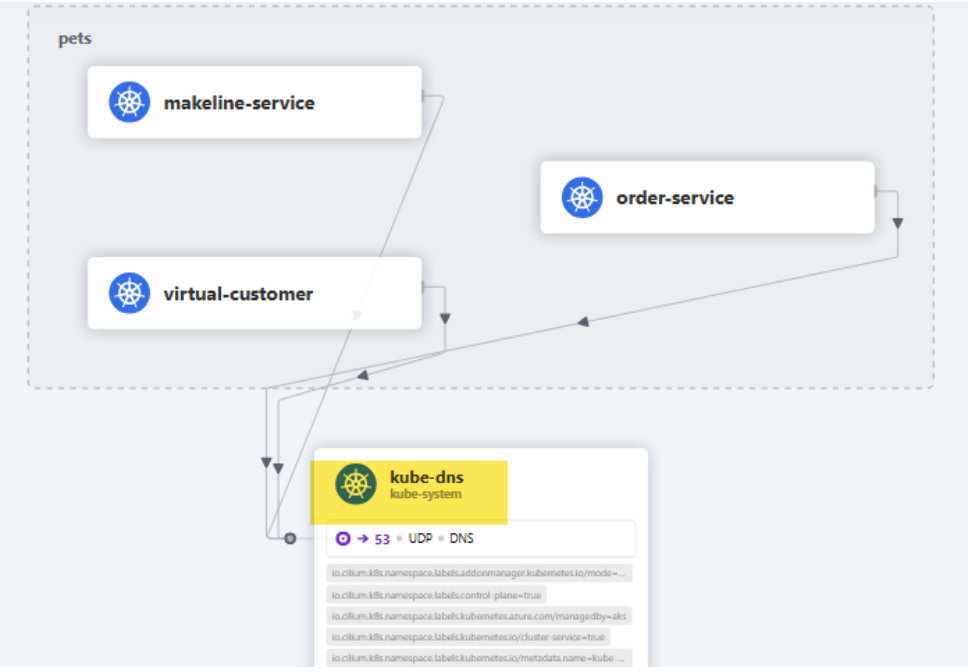







pets
mongodb
27017 TCP
makeline-service
3001 TCP
virtual-worker
rabbitmq
5672 TCP
kube-dns
kube-system
53 UDP DNS


pets
store-front
8080 TCP
/
GET
app=store-front
io.cilium.k8s.namespace.labels.kubernetes.io/metadata.name=pets
io.cilium.k8s.policy.cluster=aks01day2
io.cilium.k8s.policy.serviceaccount=default
namespace=pets
world
Unknown App


pets
makeline-service
order-service
virtual-customer
kube-dns
kube-system
53 UDP DNS

Lab 4.1 – Combined L7 and FQDN Cilium Policies

Combined.yaml

```
# 0) Allows dns pods in pets namespace to reach CoreDNS in kube-system on port 53 (UDP/TCP).
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: dns-egress-coredns
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: dns } }
  egress:
    - toEndpoints: [ { matchLabels: { k8s:io.kubernetes.pod.namespace: kube-system, k8s-app: kube-
dns } } ]
      toPorts:     [ { ports: [ { port: "53", protocol: UDP }, { port: "53", protocol: TCP } ] } ]

---
# 1) Namespace-wide egress baseline: All Pet pods can DNS queries and reach ext. FQDNs on HTTP/S.
# other than what's defined, all egress blocked (default deny principle).
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: pets-egress-foundation
  namespace: pets
spec:
  endpointSelector: {}
  egress:
    - toEndpoints:
        - matchExpressions:
            - { key: "k8s:io.kubernetes.pod.namespace", operator: In, values: ["kube-system"] }
            - { key: "k8s:k8s-app", operator: In, values: ["kube-dns", "coredns"] }
      toPorts:
        - ports:
            - { port: "53", protocol: UDP }
            - { port: "53", protocol: TCP }
          rules:
            dns:
              - { matchPattern: "*" }
    - toFQDNs:
        - { matchPattern: "*" }          # tighten later if needed
      toPorts:
        - ports:
            - { port: "80",  protocol: TCP }
            - { port: "443", protocol: TCP }

---
# 2) Ensures Load Test pods (Vegeta, or any in Pets) can hit store-front service on port 8080.
# DON'T COMMENT THIS! or you get 'policy-verdict:none EGRESS DENIED | Policy denied DROPPED'
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: pets-egress-to-store-front
  namespace: pets
spec:
  endpointSelector: {}
  egress:
    - toEndpoints:
        - matchLabels: { app: store-front }
      toPorts:
        - ports: [ { port: "8080", protocol: TCP } ]

---
# 2) Store-front can call product-service at TCP 3002.
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: store-front-egress-to-product
  namespace: pets
spec:
  endpointSelector:
```

```
    matchLabels:
      app: store-front
  egress:
    - toEndpoints:
        - matchLabels:
            app: product-service
      toPorts:
        - ports:
            - port: "3002"
              protocol: TCP

---
# 3) L7 Ingress policies for HTTP services.
# Each service gets ingress allowed on its exposed port, with HTTP rules
# store-front ingress on 8080
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: store-front-ingress-any
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: store-front } }
  ingress:
    - toPorts:
        - ports: [ { port: "8080", protocol: TCP } ]
          rules: { http: [ {} ] }

---
# order-service ingress on 3000
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: order-service-ingress-any
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: order-service } }
  ingress:
    - toPorts:
        - ports: [ { port: "3000", protocol: TCP } ]
          rules: { http: [ {} ] }

---
# makeline-service ingress on 3001
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: makeline-service-ingress-any
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: makeline-service } }
  ingress:
    - toPorts:
        - ports: [ { port: "3001", protocol: TCP } ]
          rules: { http: [ {} ] }

---
# store-admin ingress on 8081
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: store-admin-ingress-any
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: store-admin } }
  ingress:
    - toPorts:
        - ports: [ { port: "8081", protocol: TCP } ]
          rules: { http: [ {} ] }

---
```

```
# 4) Ingress L4 for non-HTTP backends: mongo and rabbitmq
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: mongodb-ingress-any
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: mongodb } }
  ingress:
    - toPorts:
        - ports: [ { port: "27017", protocol: TCP } ]

---
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: rabbitmq-ingress-any
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: rabbitmq } }
  ingress:
    - toPorts:
        - ports: [ { port: "5672", protocol: TCP } ]

---
# 5) Internal service-to-service egress.
# store-front-egress-internal: store-front -> makeline-service:3001, order-service:3000, product-
service:3002
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: store-front-egress-internal
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: store-front } }
  egress:
    - toEndpoints: [ { matchLabels: { app: makeline-service } } ]
      toPorts:   [ { ports: [ { port: "3001", protocol: TCP } ] } ]
    - toEndpoints: [ { matchLabels: { app: order-service } } ]
      toPorts:   [ { ports: [ { port: "3000", protocol: TCP } ] } ]

---
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: makeline-egress-backends
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: makeline-service } }
  egress:
    - toEndpoints: [ { matchLabels: { app: order-service } } ]
      toPorts:   [ { ports: [ { port: "3000", protocol: TCP } ] } ]
    - toEndpoints: [ { matchLabels: { app: mongodb } } ]
      toPorts:   [ { ports: [ { port: "27017", protocol: TCP } ] } ]
    - toEndpoints: [ { matchLabels: { app: rabbitmq } } ]
      toPorts:   [ { ports: [ { port: "5672", protocol: TCP } ] } ]

---
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: order-egress-backends
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: order-service } }
  egress:
    - toEndpoints: [ { matchLabels: { app: mongodb } } ]
      toPorts:   [ { ports: [ { port: "27017", protocol: TCP } ] } ]
    - toEndpoints: [ { matchLabels: { app: rabbitmq } } ]
      toPorts:   [ { ports: [ { port: "5672", protocol: TCP } ] } ]
```

```
---
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: product-service-ingress
  namespace: pets
spec:
  endpointSelector:
    matchLabels:
      app: product-service
  # ingressDeny:
  #   - fromEntities:
  #       - all
  ingress:
    - toPorts:
        - ports:
            - port: "3002"
              protocol: TCP

---
# 6) Client jobs: Virtual clients restricted to hitting only their target services.
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: virtual-worker-egress
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: virtual-worker } }
  egress:
    - toEndpoints: [ { matchLabels: { app: makeline-service } } ]
      toPorts:   [ { ports: [ { port: "3001", protocol: TCP } ] } ]

---
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: virtual-customer-egress
  namespace: pets
spec:
  endpointSelector: { matchLabels: { app: virtual-customer } }
  egress:
    - toEndpoints: [ { matchLabels: { app: order-service } } ]
      toPorts:   [ { ports: [ { port: "3000", protocol: TCP } ] } ]
```

Lab 4.2 – Commands for Hubble traces

```
$storefrontpod = kubectl -n pets get pod -l app=store-front -o jsonpath='{.items[0].metadata.name}'
$makelinepod = kubectl -n pets get pod -l app=makeline-service -o jsonpath='{.items[0].metadata.name}'
$orderpod = kubectl -n pets get pod -l app=order-service -o jsonpath='{.items[0].metadata.name}'
$mongopod = kubectl -n pets get pod -l app=mongodb -o jsonpath='{.items[0].metadata.name}'
$rabbitmqpod = kubectl -n pets get pod -l app=rabbitmq -o jsonpath='{.items[0].metadata.name}'
$virtcustomerpod = kubectl -n pets get pod -l app=virtual-customer -o jsonpath='{.items[0].metadata.name}'
$virtworkerpod = kubectl -n pets get pod -l app=virtual-worker -o jsonpath='{.items[0].metadata.name}'
$productsvcpod = kubectl -n pets get pod -l app=product-service -o jsonpath='{.items[0].metadata.name}'

$storefrontpod
$makelinepod
$orderpod
$mongopod
$rabbitmqpod
$virtcustomerpod
$virtworkerpod
$productsvcpod

hubble observe -n pets --pod $storefrontpod --follow
hubble observe -n pets --pod $makelinepod --follow
hubble observe -n pets --pod $orderpod --follow
hubble observe -n pets --pod $virtcustomerpod --follow
hubble observe -n pets --pod $virtworkerpod --follow
hubble observe -n pets --pod $productsvcpod --follow
#
hubble observe -n pets --pod $mongopod --follow
hubble observe -n pets --pod $rabbitmqpod --follow

# Apply Cilium Network Policies (CNP)
kubectl apply -f .\combined.yaml
kubectl get cnp

# Delete Cilium Network Policies (CNP)
kubectl delete $(kubectl get cnp -o name)
kubectl get cnp

# Described applied Policies
kubectl describe cnp
```

Lab 4.3 – Load test

Run this to delete, in case of previous stale jobs:

```
kubectl delete job vegeta-ramp-then-hold  --ignore-not-found
```

Run this to start off the load:

```
@'
apiVersion: batch/v1
kind: Job
metadata:
  name: vegeta-ramp-then-hold
  namespace: pets
spec:
  backoffLimit: 0
  template:
    spec:
      restartPolicy: Never
      containers:
      - name: vegeta
        image: peterevans/vegeta:6.9.1
        command: ["/bin/sh","-lc"]
        args:
        - |
          printf "GET http://store-front.pets.svc.cluster.local/\n" > /tmp/t.txt
          rm -f /tmp/results.bin

          echo "--- Phase 1: ramp to 1000 rps (steps in 20s each) ---"
          for r in 600 800 1000; do
            echo "[Phase1] $r rps for 20s"
            vegeta attack -targets=/tmp/t.txt -timeout=10s -rate=$r -duration=20s \
              | tee -a /tmp/results.bin \
              | vegeta report -type=text -every=5s
          done

          echo "--- Phase 2: steady 1100 rps for 10m ---"
          vegeta attack -targets=/tmp/t.txt -timeout=10s -rate=1100 -duration=10m \
            | tee -a /tmp/results.bin \
            | vegeta report -type=text -every=5s

          echo "=== Final summary (all phases) ==="
          vegeta report < /tmp/results.bin
'@ | Out-File vegeta-ramp-then-hold.yaml -Encoding utf8

kubectl apply -f vegeta-ramp-then-hold.yaml
sleep 5
kubectl -n pets logs -f job/vegeta-ramp-then-hold

# USE ONLY IF PODS NOT DELETED
kubectl delete job vegeta-ramp-then-hold -n pets
```

Lab 4.3 – Policy tests

Test 1: Connectivity to store-front fails

1. Drop the policy that explicitly allows ingress pods as Vegeta to reach the store-front pod.

```yaml
# 2) Fix vegeta egress: allow ANY pod in pets to reach store-front:8080. Patching egress above
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: pets-egress-to-store-front
  namespace: pets
spec:
  endpointSelector: {}
  egress:
    - toEndpoints:
        - matchLabels: { app: store-front }
      toPorts:
        - ports: [ { port: "8080", protocol: TCP } ]
```

```
kubectl delete cnp pets-egress-to-store-front
```

2. Observe the drop in store-front

```
$storefrontpod = kubectl -n pets get pod -l app=store-front -o jsonpath='{.items[0].metadata.name}'
hubble observe -n pets --pod $storefrontpod --follow
pets/vegeta-ramp-then-hold-j5jht:55557 (ID:5060) <> pets/store-front-5b8cf5944b-ffdr9:8080 (ID:49701) Policy denied DROPPED (TCP Flags: SYN)
pets/vegeta-ramp-then-hold-j5jht:40351 (ID:5060) <> pets/store-front-5b8cf5944b-ffdr9:8080 (ID:49701) policy-verdict:none EGRESS DENIED (TCP Flags: SYN)
```

3. Keep this going for a while and then restore the policy.

```
# Apply Cilium Network Policies (CNP)
kubectl apply -f .\combined.yaml
kubectl get cnp
```

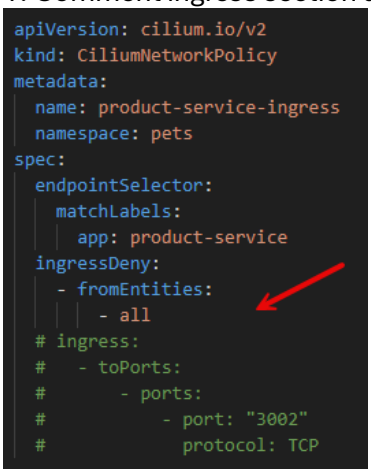
4. Verify from the hubble logs that traffic isn't blocked

```
pets/vegeta-ramp-then-hold-j5jht:42133 (ID:5060) -> pets/store-front-5b8cf5944b-ffdr9:8080 (ID:49701) http-request FORWARDED (HTTP/1.1 GET http://store-front.pets.svc.cluster.local/)
pets/vegeta-ramp-then-hold-j5jht:47705 (ID:5060) -> pets/store-front-5b8cf5944b-ffdr9:8080 (ID:49701) http-request FORWARDED (HTTP/1.1 GET http://store-front.pets.svc.cluster.local/)
pets/vegeta-ramp-then-hold-j5jht:47705 (ID:5060) <- pets/store-front-5b8cf5944b-ffdr9:8080 (ID:49701) http-response FORWARDED (HTTP/1.1 200 0ms (GET http://store-front.pets.svc.cluster.local/)
```

Test 2: Connectivity from store-front to product-service fails

Product-service provides product content to the store-front service. Block ingress and see outcome

1. Comment ingress section and uncomment ingressDeny section.



```yaml
apiVersion: cilium.io/v2
kind: CiliumNetworkPolicy
metadata:
  name: product-service-ingress
  namespace: pets
spec:
  endpointSelector:
    matchLabels:
      app: product-service
  ingressDeny:
    - fromEntities:
        - all
  # ingress:
  #   - toPorts:
  #       - ports:
  #           - port: "3002"
  #             protocol: TCP
```

```
# Apply Cilium Network Policies (CNP)
kubectl apply -f .\combined.yaml
kubectl get cnp
```

2. Get LB IP of store-front and validate images don't show and check for drops from hubble trace

```
kubectl get svc store-front -o jsonpath="{.status.loadBalancer.ingress[0].ip}"
# open browser to this IP
```

```
$productsvcpod = kubectl -n pets get pod -l app=product-service -o
jsonpath='{.items[0].metadata.name}'
$productsvcpod
hubble observe -n pets --pod $productsvcpod –follow
```

3. Quick in-cluster confirm (fail-fast) using name resolution

```
kubectl -n pets run -it --rm t --image=busybox --restart=Never -- `
  sh -lc 'timeout 5 nc -vz product-service 3002 || exit 1'

# From store-front specifically (if shelling into it is easier)
kubectl -n pets exec -it deploy/store-front -- sh -lc 'timeout 5 nc -vz product-service 3002 ||
exit 1'
```

3. Connectivity from store-front

```
kubectl -n pets exec -it deploy/store-front -- `
  sh -lc 'timeout 5 nc -vz product-service 3002 && echo OK'

-- results in product-service (10.0.84.68:3002) open OK

# (or) check for content returned
kubectl -n pets exec -it deploy/store-front -- `
  sh -lc 'wget -qO- http://product-service:3002/ -T 5 || echo FAIL'

-- results in web page
```

4. Confirm drops. Once complete, comment ingressDeny and uncomment ingress sections. Apply combined.yaml and service should be restored, as seen below.

```
10.224.0.5:46902 (host) <> pets/product-service-856c5db485-lk6gm:3002 (ID:2711) policy-verdict:all INGRESS DENIED (TCP Flags: SYN)
10.224.0.5:46902 (host) <> pets/product-service-856c5db485-lk6gm:3002 (ID:2711) Policy denied by denylist DROPPED (TCP Flags: SYN)
10.224.0.5:46892 (host) -> pets/product-service-856c5db485-lk6gm:3002 (ID:2711) policy-verdict:L4-Only INGRESS ALLOWED (TCP Flags: SYN)
10.224.0.5:46892 (host) -> pets/product-service-856c5db485-lk6gm:3002 (ID:2711) to-endpoint FORWARDED (TCP Flags: SYN)
10.224.0.5:46892 (host) <- pets/product-service-856c5db485-lk6gm:3002 (ID:2711) to-stack FORWARDED (TCP Flags: SYN, ACK)
10.224.0.5:46892 (host) -> pets/product-service-856c5db485-lk6gm:3002 (ID:2711) to-endpoint FORWARDED (TCP Flags: ACK)
10.224.0.5:46892 (host) -> pets/product-service-856c5db485-lk6gm:3002 (ID:2711) to-endpoint FORWARDED (TCP Flags: ACK, FIN)
```

Test 3: Traffic Surge

```
# INCREASE LOAD
kubectl -n pets set env deploy/virtual-customer ORDERS_PER_HOUR=90000
kubectl -n pets set env deploy/virtual-worker ORDERS_PER_HOUR=90000

# CHECK SERVICE LOGS
kubectl -n pets logs deploy/virtual-customer --tail=200 -f
kubectl -n pets logs deploy/virtual-worker --tail=200 -f

# CHECK HUBBLE TRACE
$virtworkerpod = kubectl -n pets get pod -l app=virtual-worker -o
jsonpath='{.items[0].metadata.name}'
$virtcustomerpod = kubectl -n pets get pod -l app=virtual-customer -o
jsonpath='{.items[0].metadata.name}'

hubble observe -n pets --pod $virtworkerpod --follow
hubble observe -n pets --pod $virtcustomerpod --follow

# RESET on completion
kubectl -n pets set env deploy/virtual-customer ORDERS_PER_HOUR=100
kubectl -n pets set env deploy/virtual-worker ORDERS_PER_HOUR=100
```

Test 4: Node MemoryPressure from Multiple Memory Hogs

Observations

When multiple memory-hog pods are scheduled onto the same node, they collectively push usage close to or beyond the node's allocatable memory, causing the kubelet to flag `MemoryPressure=True` and triggering pod evictions or `OOMKilling` events. This would mirror a failure scenario like a runaway workload with unbounded memory growth or several services misconfigured without memory limits.

Steps

1. Set the node to the one running store-front and label the node so as to target it with the stress pods

```
$storefrontpod = kubectl -n pets get pod -l app=store-front -o jsonpath='{.items[0].metadata.name}'
$node = kubectl -n pets get pod $storefrontpod -o jsonpath='{.spec.nodeName}'
kubectl label node $node  burn=mem –overwrite
```

2. Deploy a fleet of memory hogs, each allocating ~80% → 90% → ~100% of their cgroup limit in phases. It runs on nodes with label burn=mem

```
@'
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mem-hog-ramped-fleet
  namespace: pets
spec:
  replicas: 5      # adjust based on node size
  selector:
    matchLabels:
      app: mem-hog-ramped-fleet
  template:
    metadata:
      labels:
        app: mem-hog-ramped-fleet
    spec:
      nodeSelector:
        burn: mem
      containers:
      - name: hog
        image: python:3.11-alpine
        env:
          - { name: PCT_PHASES, value: "80,90,100" }
          - { name: HOLD_SEC, value: "60" }
          - { name: STEP_MIB, value: "8" }
          - { name: HEADROOM_MIB, value: "64" }
        resources:
          requests: { memory: "2200Mi" }
          limits:   { memory: "3Gi" }
        command: ["python","-c"]
        args:
          - |
            import os,time
            STEP=int(os.getenv("STEP_MIB","8"))
            HOLD=int(os.getenv("HOLD_SEC","60"))
            HEAD=int(os.getenv("HEADROOM_MIB","64"))
            phases=[int(x) for x in os.getenv("PCT_PHASES","80,90,100").split(",")]
            LIMIT=int(open("/sys/fs/cgroup/memory.max").read().strip())
            blocks=[]
            def rss_mib():
                pages=int(open("/proc/self/statm").read().split()[1])
                return (pages*os.sysconf("SC_PAGE_SIZE"))//(1024*1024)
            for pct in phases:
                target=int(LIMIT*pct/100)//(1024*1024)-HEAD
                while rss_mib()<target:
                    blocks.append(bytearray(1024*1024*STEP))
                    time.sleep(0.05)
                print(f"Reached ~{rss_mib()} MiB ({pct}%)", flush=True)
                time.sleep(HOLD)
            time.sleep(300)
```

```
'@ | kubectl apply -f -
```

3. Verify placement: all pods should run on the same node.

```
kubectl -n pets get pod -l app=mem-hog-ramped-fleet -o wide
```

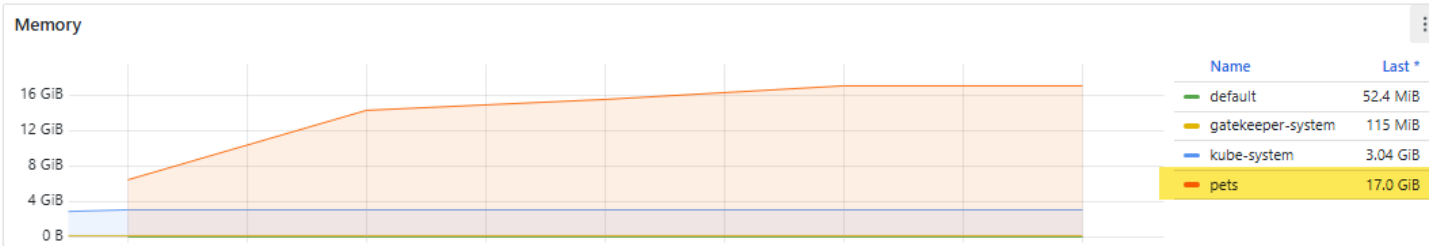
4. Get events and check node condition:

```
kubectl get events --all-namespaces --sort-by=.lastTimestamp | Select-String
"MemoryPressure|Evicted|OOMKilling"
kubectl describe node $node | Select-String MemoryPressure
```

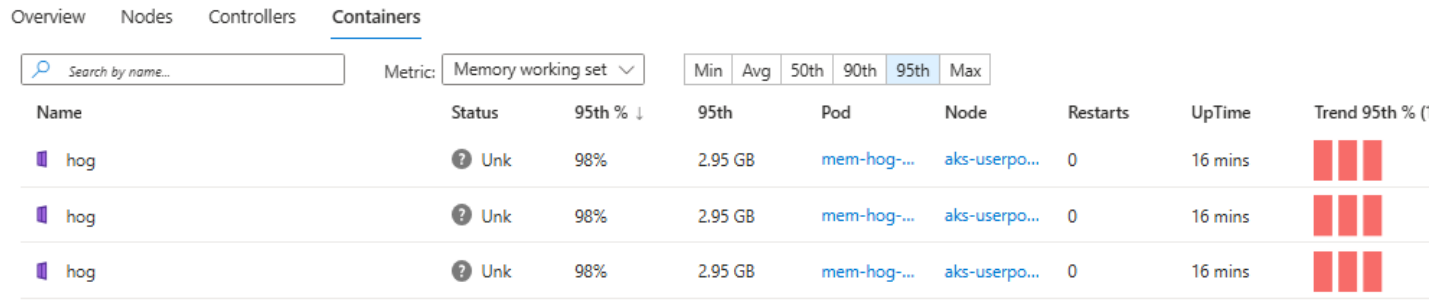
5. Locate offenders to see top hoggers

```
kubectl -n pets top pod --sort-by=memory
```

6. Check Dashboards with '**Kubernetes | Compute Resources | Cluster'**. Also 'Kubernetes | Compute Resources | Namespace (Workloads)'



Also check Insights > Containers



Resolution

1. Delete the hog fleet:

```
kubectl -n pets delete deploy mem-hog-ramped-fleet
kubectl label node $node burn-
```

2. Confirm node recovers and Status should return to False.

```
kubectl describe node aks-userpool-41046857-vmss00002h | Select-String MemoryPressure
```

On completion

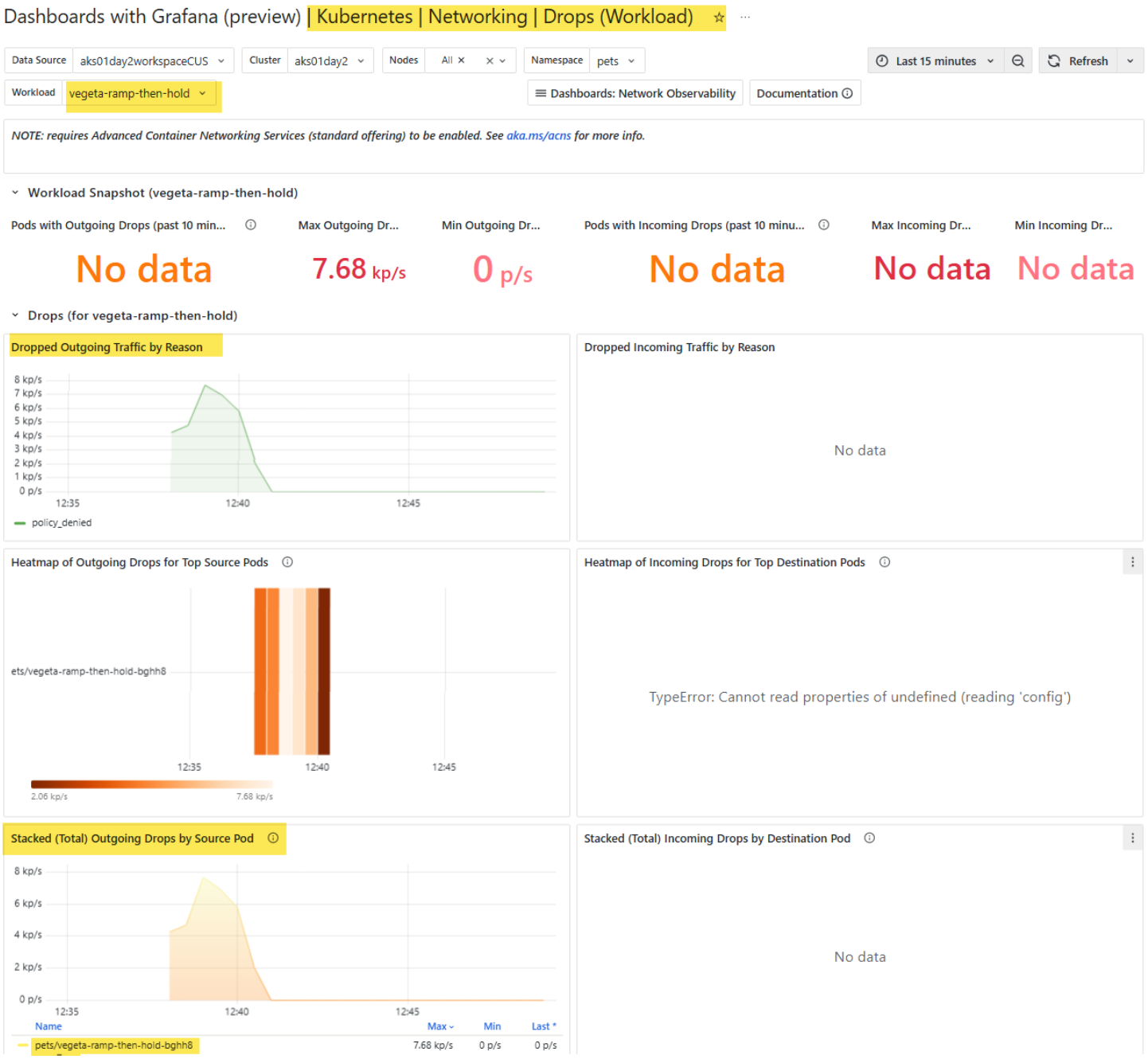
Delete existing Cilium Network Policies

```
kubectl delete $(kubectl get cnp -o name); kubectl get cnp
```

Lab 4.4 – Grafana Dashboards for CNS

Networking-Drops-Workload

The **vegeta-ramp-then-hold** pod is being denied egress (outbound) traffic by Cilium policies. Drops are exclusively **policy_denied** and concentrated in this workload. Likely cause: either missing egress policies to store-front or incomplete service-to-service egress rules.

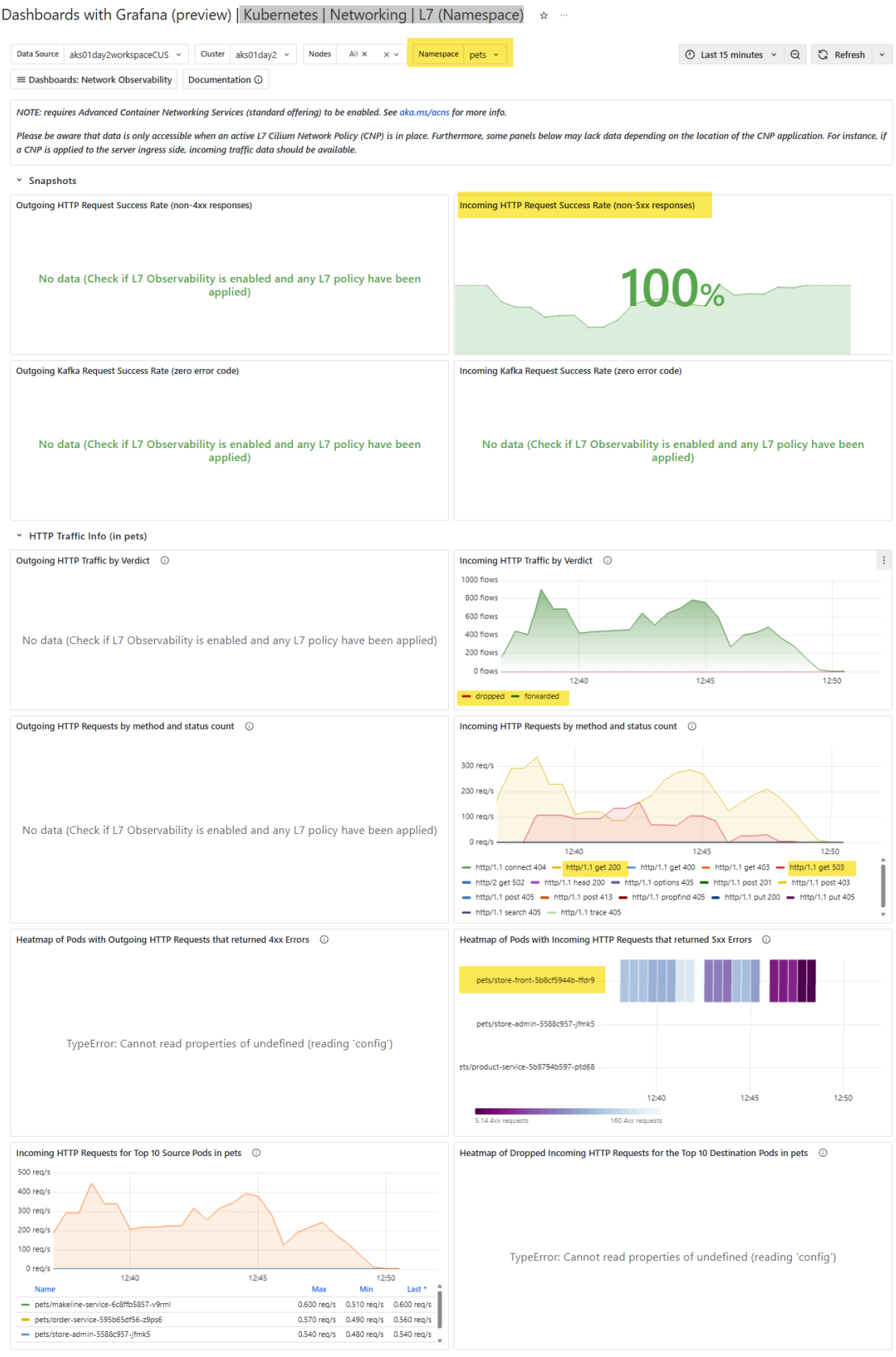


Networking-DNS-Cluster

The DNS (Cluster) dashboard shows that CoreDNS is handling most queries successfully, with responses matching requests, but there are spikes of **NXDOMAIN errors for AAAA lookups** (expected in an IPv4-only cluster). These errors aren't service failures but noise from apps requesting IPv6 records. Heavy DNS activity comes from virtual-customer, virtual-worker, makeline-service, and order-service pods. Overall, DNS performance is healthy, with brief missing responses; policies allowing both **A/AAAA** queries ensure traffic is not being blocked.

Networking-L7-Namespace

The L7 (Namespace) dashboard shows healthy overall HTTP success rates for the pets namespace, but a significant portion of incoming requests are **dropped or erroring (4xx/5xx)**, especially on store-front, store-admin, and product-service. This indicates **L7 network policies are enforcing restrictions**, with some client traffic failing due to invalid paths/methods or application-level issues, while services still return mostly successful 200 responses.



Networking-Pod flows-Namespace

The Pod Flows dashboard shows a healthy mesh of traffic in the pets namespace, with client-like pods **(virtual-customer, virtual-worker, vegeta-ramp-then-hold)** driving most of the egress and services like store-front and makeline-service receiving the bulk of incoming flows. While most traffic is forwarded successfully, some **egress drops and high RST spikes** indicate connections being rejected or reset, often tied to missing or restrictive policies, especially under load.