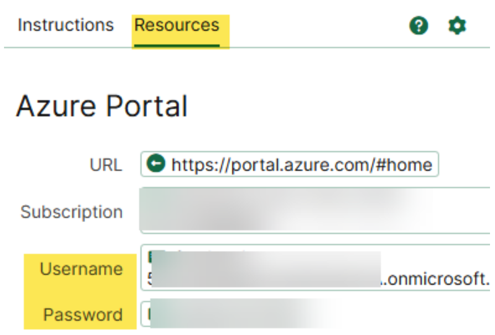
Module 1 – Initial Setup

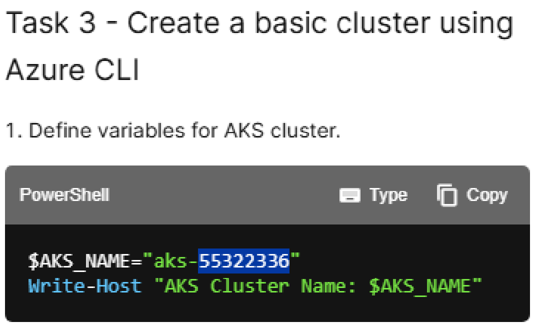
This section covers start-to-finish, AKS setup via az CLI (PowerShell) for East US 2, with a tainted system pool and a separate user pool, plus monitoring to a Log Analytics workspace.

Step 0 : Login & (optional) subscription



Azure Portal

URL	https://portal.azure.com/#home
Subscription	[redacted]
Username	[redacted].onmicrosoft.
Password	[redacted]



Task 3 - Create a basic cluster using Azure CLI

1. Define variables for AKS cluster.

```
$AKS_NAME="aks-55322336"
Write-Host "AKS Cluster Name: $AKS_NAME"
```

```
az login  # Choose "Work or School Account" -> Set the correct subscription -> Enter the
username/password found in the Resources tab. Auto sign > 'yes, all apps' if asked.
```

Step 1 : Set Variables

```
$id     = "55322336"  # Get this Id from the **Instructions** tab, as seen above
$loc    = "eastus2"
$rg     = "azure-rg"
$law    = "aks-$id-law"
$aks    = "aks-$id"
$aks
$rg

# set shortcut for Powershell.
set-alias k
> kubectl
k get all
```

Step 1 : 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 : 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
sleep 60
$ip = kubectl -n demo get svc hello -o jsonpath='{.status.loadBalancer.ingress[0].ip}' 2>$null
Invoke-WebRequest http://$ip/

# when done
kubectl delete ns demo
```

Module 1 – Horizontal Pod Autoscaler

- Horizontal Pod Autoscaler (HPA) is a native Kubernetes object that automatically scales the number of pods in a replication controller, deployment, replica set, or stateful set
- HPA is based on observed
    - **CPU utilization**
    - **custom metrics support like Memory**
- The **Metrics Server in Kubernetes** cluster is used by HPA to track resource demands of pods
- The number of pods is automatically increased whenever an application requires more resources. The number of pods is decreased when demand gets lower
- HPA helps to ensure that applications have sufficient resources to handle varying workloads
- HPA reduces the need for manual intervention in scaling applications
- HPA helps in achieving better resource utilization and cost-effectiveness

For more details see this link.

Prereqs & quick checks

Goal: confirm the cluster can surface resource metrics (required for HPA).

1. Make sure you can reach your cluster and kubectl is set.
2. Ensure **metrics-server** is present (HPA consumes the Metrics API). Check with:

```
kubectl top nodes
kubectl top pods -A
```

If these return values, you're good. Metrics Server exposes CPU/memory to HPA and kubectl top.

Lab 1: CPU-based Autoscaling

Autoscale a php-apache Deployment between 1-10 replicas to maintain ~50% average CPU utilization.

Step 1: Deploy sample app and Service

```
kubectl create ns hpatest
kubectl config set-context --current --namespace=hpatest
kubectl apply -f https://k8s.io/examples/application/php-apache.yaml -n hpatest
kubectl get all
```

This creates a php-apache Deployment (200m CPU request, 500m limit) and a Service on port 80. Validate using below and verify.

```
kubectl get pod -l run=php-apache -o jsonpath="{.items[*].spec.containers[*].resources}"
```

Step 2: Create the HPA (target 50% CPU)

This instructs HPA to keep average CPU at ~50% of the pod request (≈100m).

```
kubectl autoscale deployment php-apache --cpu-percent=50 --min=1 --max=10
```

Step 3: Confirm HPA exists

You should see php-apache with target 0%/50%, min 1, max 10. Keep a watch on hpa

```
kubectl get hpa -w
```

Step 4: Generate load

You will need to run the load generator and watch pods (in a separate terminal). This loop drives CPU on the php-apache pods.

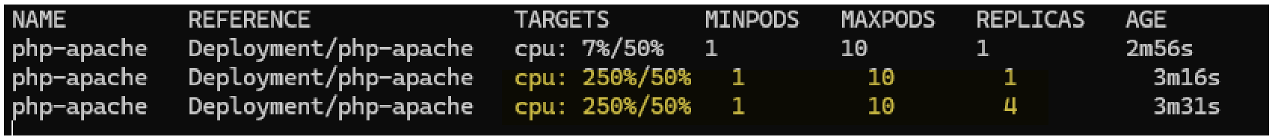
```
kubectl run -i --tty load-generator --rm --image=busybox:1.28 --restart=Never -- `
/bin/sh -c "while sleep 0.01; do wget -q -O- http://php-apache; done"
```

```
kubectl get pod -l run=php-apache -w
```

After ~5m do a CTRL+C to break the loop and terminate the load generator.

Step 5: Watch the autoscaling reaction

```
kubectl get hpa php-apache –watch
```



```
NAME         REFERENCE                 TARGETS       MINPODS   MAXPODS   REPLICAS   AGE
php-apache   Deployment/php-apache     cpu: 7%/50%   1         10        1          2m56s
php-apache   Deployment/php-apache     cpu: 250%/50%   1         10        1          3m16s
php-apache   Deployment/php-apache     cpu: 250%/50%   1         10        4          3m31s
```

# In another shell:
```
kubectl get deployment php-apache
```

```
NAME         READY   UP-TO-DATE   AVAILABLE   AGE
php-apache   5/5     5            5           10m
```

Within ~1-2 minutes you should see replicas increase (for example to 6/6) and later scale back down once load stops or break to stop generator.

Expected results

- `kubectl get hpa` shows rising current CPU %, desired replicas > 1.
- `kubectl get deployment php-apache` shows READY pods matching HPA's desired count, then scales back to 1 when idle.

Step 6: Terminate

```
k delete pod/load-generator
k delete hpa php-apache
k delete deployment.apps/php-apache
k delete service/php-apache
```

Lab 2: Memory-based Autoscaling

Objective

Extend the HPA to also scale on memory using the autoscaling/v2 API with a concrete **AverageValue** target. A **memory-based** Horizontal Pod Autoscaler (HPA) monitors how much memory each pod in a Deployment (or ReplicaSet/StatefulSet) is using.

The following will be done:

- Configure a **target**, e.g. averageValue: 200Mi.
- The HPA queries **metrics-server** for actual pod memory usage.
- If the average usage across pods **exceeds the target**, the HPA increases replicas (up to maxReplicas).
- If the average usage stays **below the target** for a while, it reduces replicas (down to minReplicas).
- Unlike CPU, memory is "sticky": once allocated, it may not drop quickly, so downscaling takes longer.

Step 1: Delete the HPA if it exists

```
kubectl delete hpa php-apache
```

Step 2: Apply the new spec

```
@"
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: mem-hpa
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: mem-hpa
  minReplicas: 1
  maxReplicas: 10
  metrics:
    - type: Resource
      resource:
        name: memory
        target:
          type: AverageValue
          averageValue: 150Mi    # target average memory per pod
"@ | Set-Content .\hpa-mem.yaml -Encoding ascii
```

```
kubectl apply -f .\hpa-mem.yaml -n hpatest
```

```
kubectl get hpa mem-hpa -w
```

```
NAME      REFERENCE            TARGETS                  MINPODS    MAXPODS    REPLICAS    AGE
mem-hpa   Deployment/mem-hpa   memory: 11968512/200Mi   1          10         1           32s
```

You should now see Memory entries. The HPA considers each metric and picks the **highest** desired replica count.

Step 3: Apply App + Service inline YAML

This will be the target for the memory load test that will cause HPA to trigger due to memory drive and on conclusion the GC will take it away instantly so a quick memory drop should bring down the HPA count also.

```
@"
apiVersion: v1
kind: ConfigMap
metadata:
  name: mem-app
data:
  app.js: |
    const http = require('http');
    const port = process.env.PORT || 8080;

    const CHUNK   = parseInt(process.env.CHUNK   || "20971520"); // 20 Mi
    const HOLD_MS = parseInt(process.env.HOLD_MS || "800");       // 0.8 s

    const server = http.createServer((req, res) => {
      if (req.url === '/gc') {
        if (global.gc) {
          global.gc();
          console.log('Manual GC triggered');
          res.end('GC done');
        } else {
          res.end('GC not available (start node with --expose-gc)');
        }
        return;
      }

      // Normal load path
      let buf = Buffer.alloc(CHUNK, 0x61);
      setTimeout(() => {
        buf = null;
        res.end('ok');
      }, HOLD_MS);
    });

    server.listen(port, () => console.log('Listening on ${port}'));
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: mem-hpa
spec:
  replicas: 1
  selector:
    matchLabels:
      app: mem-hpa
  template:
    metadata:
      labels:
        app: mem-hpa
    spec:
      containers:
        - name: app
          image: node:18-alpine
          command: ["node","--expose-gc","/app/app.js"]
          env:
            - name: CHUNK
              value: "20971520"   # 20 Mi per request
            - name: HOLD_MS
              value: "800"        # keep memory ~0.8s
          ports:
            - containerPort: 8080
          resources:
            requests:
              cpu: "50m"
              memory: "100Mi"
            limits:
              cpu: "500m"
              memory: "600Mi"
          volumeMounts:
            - name: app-code
              mountPath: /app
      volumes:
        - name: app-code
          configMap:
            name: mem-app
---
apiVersion: v1
```

```
kind: Service
metadata:
  name: mem-hpa
spec:
  selector:
    app: mem-hpa
  ports:
    - port: 80
      targetPort: 8080
"@ | Set-Content .\mem-app.yaml -Encoding ascii
```

Now apply file and check if running.

```
kubectl apply -f .\mem-app.yaml
kubectl get deploy
kubectl get svc mem-hpa
```

Step 4: Confirm app works

App should be returning a 200 OK

```
$pod = kubectl get pod -l app=mem-hpa -o jsonpath='{.items[0].metadata.name}'
kubectl exec -it $pod -- wget -qO- http://localhost:8080
```

Step 5: Start the memory load tester

Generate memory load that runs at 3 req/s for 200s (~600 requests):

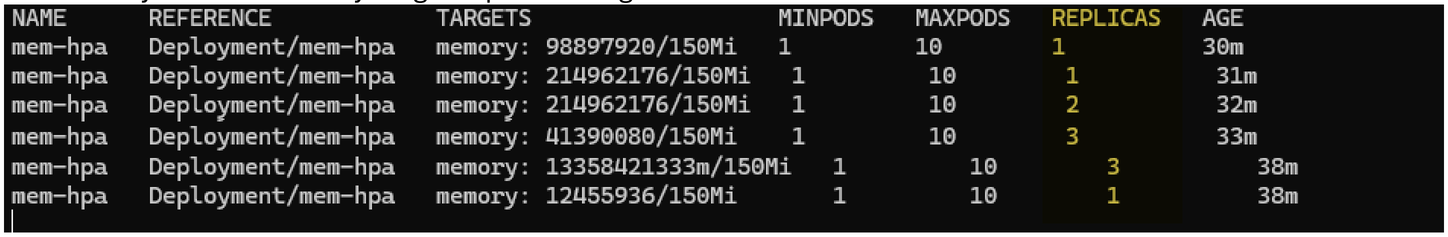
```
kubectl run -it --rm loader --image=busybox:1.36 --restart=Never -- /bin/sh -c '
for i in $(seq 1 6000); do (wget -q -O- http://mem-hpa >/dev/null 2>&1) & sleep 0.1; done; wait
'
```

Step 6: Trigger GC

On completion trigger GC by running below to force HPA to bring replica count to default.

```
kubectl exec -it $(kubectl get pod -l app=mem-hpa -o jsonpath='{.items[0].metadata.name}') -- wget
-qO- http://localhost:8080/gc
```

HPA initially detects memory surge so pod count goes to 3 and thereafter reduces from 3 to 1



```
NAME      REFERENCE              TARGETS                    MINPODS   MAXPODS   REPLICAS   AGE
mem-hpa   Deployment/mem-hpa     memory: 98897920/150Mi     1         10        1          30m
mem-hpa   Deployment/mem-hpa     memory: 214962176/150Mi     1         10        1          31m
mem-hpa   Deployment/mem-hpa     memory: 214962176/150Mi     1         10        2          32m
mem-hpa   Deployment/mem-hpa     memory: 41390080/150Mi      1         10        3          33m
mem-hpa   Deployment/mem-hpa     memory: 13358421333m/150Mi     1         10        3          38m
mem-hpa   Deployment/mem-hpa     memory: 12455936/150Mi         1         10        1          38m
```

Step 7: Clean-up (optional)

```
kubectl delete hpa mem-hpa
kubectl delete -f .\mem-app.yaml
```

Module 2 – Vertical Pod Autoscaler (VPA)

Objective

A **CRD (Custom Resource Definition)** in Kubernetes is an extension that lets you define your own custom resource types, making Kubernetes natively understand and manage new objects beyond its built-in ones.

- **Vertical Pod Autoscaler (VPA)** is a Custom Resource Definition (CRD) that **adjusts resource allocations** for individual containers based on their usage patterns. This helps optimize resource utilization and reduce wastage.
- VPA scales the resources allocated to individual pods **vertically**, in contrast to the Horizontal Pod Autoscaler (HPA), which scales the number of replicas **horizontally**.
- VPA can be used to **optimize CPU, memory,** or other specific container resources.

VPA operates in four modes:
1. **Auto**: Allocates resource requests and modifies them for both existing and newly created pods. Updates are applied without requiring pod restarts when possible.
2. **Recreate**: Allocates resource requests and modifies them for existing and newly created pods. If the new recommendations differ significantly, pods are evicted and recreated.
3. **Initial**: Allocates resource requests only to newly created pods and never modifies them afterward.
4. **Off**: Does not automatically alter pod resource requests but provides recommendations that can be reviewed.

VPA has three main components:
1. **Recommender**: Generates resource recommendations for pods.
2. **Updater**: Applies VPA recommendations automatically without manual intervention.
3. **Admission Plugin**: Enforces the appropriate resource requests.

See link for more details

Step 1: Update existing AKS cluster

```
az aks update --name $aks --resource-group $rg --enable-vpa
az aks get-credentials -g $rg --name $aks
```

Step 2: Confirm VPA

List the 4 VPA pods and 2 CRDs

```
kubectl get pod -A | findstr vpa
kubectl get crd | findstr vertical

# Follow logs (each in a separate terminal if you want live views)
kubectl -n kube-system logs deploy/vpa-recommender -f
kubectl -n kube-system logs deploy/vpa-updater -f
kubectl -n kube-system logs deploy/vpa-admission-controller -f
```

Step 3: Application Deployment

You'll use the deployment script in this link. Open and right-click using Save As or use PS script below.
https://raw.githubusercontent.com/kubernetes/autoscaler/refs/heads/master/vertical-pod-autoscaler/examples/hamster.yaml

```
Invoke-WebRequest -Uri
"https://raw.githubusercontent.com/kubernetes/autoscaler/refs/heads/master/vertical-pod-
autoscaler/examples/hamster.yaml" -OutFile "hamster.yaml"
```

Script deploys two pods, each running a single container that initially requests 100 millicores but attempts to use slightly more than 500 millicores. A VPA configuration is then applied to the deployment. The VPA monitors the pods' behavior and, after approximately five minutes, adjusts the pod requests to 500 millicores. VPA allows a max of 1000 millicores vCPU.

1. Deploy the application

```
kubectl apply -f hamster.yaml
```

2. Check the VPA settings

```
k get vpa
NAME          MODE   CPU    MEM    PROVIDED   AGE
hamster-vpa   Auto   587m   50Mi   True       9m9s+
```

Step 4: Validate VPA Working

1. Note the cpu and memory as shown. This will change with load as VPA dynamically resizes.

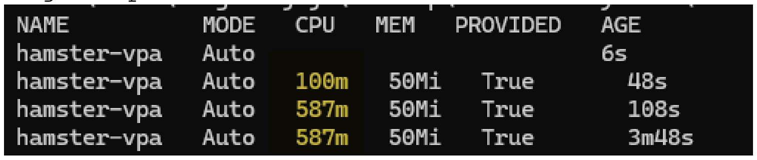
```
kubectl get pod -l app=hamster -o jsonpath="{.items[*].spec.containers[*].resources}"
=> {"requests":{"cpu":"100m","memory":"50Mi"}} {"requests":{"cpu":"100m","memory":"50Mi"}}
```

2. Watch pods for termination and recreation.

```
k get --watch pods -l app=hamster
```

3. Watch VPA changes

```
k get vpa -w
```



```
NAME          MODE   CPU    MEM    PROVIDED   AGE
hamster-vpa   Auto                            6s
hamster-vpa   Auto   100m   50Mi   True       48s
hamster-vpa   Auto   587m   50Mi   True       108s
hamster-vpa   Auto   587m   50Mi   True       3m48s
```

4. Running this again shows, which aligns with the VPA setting.

```
kubectl get pod -l app=hamster -o jsonpath="{.items[*].spec.containers[*].resources}"

{"requests":{"cpu":"587m","memory":"50Mi"}} {"requests":{"cpu":"587m","memory":"50Mi"}}
```

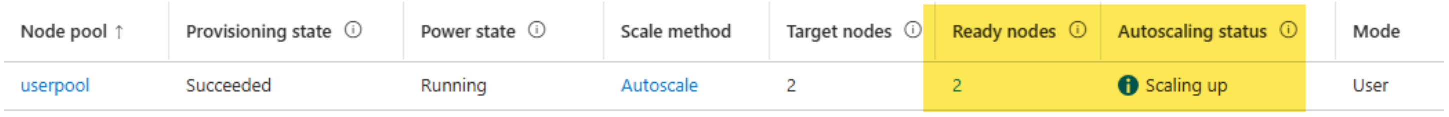
5. Test again (if needed)

```
# ensure pods are deleted
k scale deployment.apps/hamster --replicas=0

# watch pods in another cli
k get  --watch pods -l app=hamster

# ensure pods are created
k scale deployment.apps/hamster --replicas=10
```

# confirm resource size after ~5m. New pods get created and the old deleted

```
kubectl get pod -l app=hamster -o jsonpath="{.items[*].spec.containers[*].resources}"
{"requests":{"cpu":"587m","memory":"50Mi"}} {"requests":{"cpu":"587m","memory":"50Mi"}}
```

# new pods are in pending till the CAS scales up new nodes



Node pool ↑	Provisioning state ⓘ	Power state ⓘ	Scale method	Target nodes ⓘ	Ready nodes ⓘ	Autoscaling status ⓘ	Mode
userpool	Succeeded	Running	Autoscale	2	2	Scaling up	User

6. Terminate on completion

```
k delete -f hamster.yaml
```

Module 3 – Cluster Auto-Scaler

Objective

Cluster Autoscaler automatically adjusts the number of nodes to ensure workloads run efficiently and cost-effectively.

Scale-up

- **Monitors un-schedulable pods** caused by resource constraints and **increases the number of nodes** to meet application demand.
- The default interval for checking unschedulable pods is 10 seconds (this can be modified).
- Attempts to find a suitable node for unscheduled pods whenever the Kubernetes scheduler is unable to place them.

Scale-down

- Periodically evaluates the number of nodes and decreases them when appropriate, based on the following criteria:
  - A node is considered for removal if the total CPU and memory consumption of all running pods is less than 50% of the node's available resources.
  - All running pods (except static pods or those created by DaemonSets) are safely evicted and rescheduled to other nodes before the node is removed.

Other Details

- Node removal timing is configurable via the --scale-down-unneeded-time flag.
- A ConfigMap named cluster-autoscaler-status is created in the kube-system namespace to reflect the autoscalers status.
- The **HorizontalPodAutoscaler (HPA)** scales the number of pods, while the **Cluster Autoscaler** scales the number of nodes. These two components can operate individually or in coordination for efficient scaling.
- See link for more information.

Step 1: Update existing cluster with CAS

```
# Enable autoscaler on syspool
az aks nodepool update `
  --resource-group $rg `
  --cluster-name $aks `
  --name syspool `
  --enable-cluster-autoscaler `
  --min-count 1 `
  --max-count 3

# Enable autoscaler on userpool
az aks nodepool update `
  --resource-group $rg `
  --cluster-name $aks `
  --name userpool `
  --enable-cluster-autoscaler `
  --min-count 1 `
  --max-count 3
```

Step 2: Deploy the application

1. Deploy file

```
@"
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-cas
spec:
  selector:
    matchLabels:
      app: nginx-cas
  replicas: 1
  template:
    metadata:
      labels:
        app: nginx-cas
    spec:
      containers:
        - name: nginx-cas
          image: nginx
          resources:
            requests:
              memory: "2Gi"
"@ | Out-File -FilePath nginx-cas.yaml -Encoding utf8

kubectl apply -f nginx-cas.yaml
```

Step 3: Check memory requirement on node

1. Get node listing

```
k get pods -o wide
```

```
NAME          READY   STATUS    RESTARTS   AGE     IP            NODE
nginx-cas-xx  1/1     Running   0          7m10s   10.244.2.160  aks-nodepool1-20869648-vmss000001
```

2. Find allocatable util

```
k describe node  <node from above>
```

```
Allocatable:
  memory:              12051740Ki
Allocated resources:
   (Total limits may be over 100 percent, i.e., overcommitted.)
  Resource             Requests      Limits
  --------             --------      ------
  memory               3152Mi (26%)  11344Mi (96%)
```

Based on above 4 more pods of 2Gi will be allow till the 5th will go Pending and trigger Cluster Autoscaler.

Step 4: Scale deployment

1. Increase replica count to 5 to get more pods

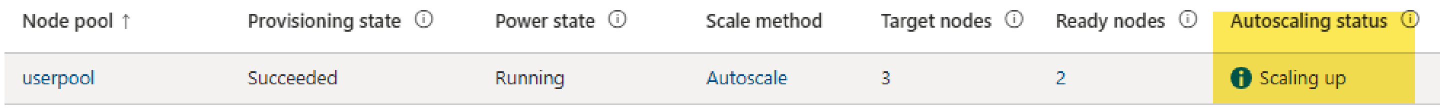
```
kubectl scale deployment nginx-cas --replicas 5
NAME                               READY   STATUS    RESTARTS   AGE
pod/nginx-cas-67c5677977-8jc4q     1/1     Running   0          4s
pod/nginx-cas-67c5677977-j8mrh     1/1     Running   0          10m
pod/nginx-cas-67c5677977-n8n7n     1/1     Running   0          4s
pod/nginx-cas-67c5677977-q22jd     0/1     Pending   0          4s
pod/nginx-cas-67c5677977-r27wf     1/1     Running   0          4s
```

2. If pod goes to Pending state, check the reasoning

```
kubectl describe pod/nginx-cas-67c5677977-q22jd
```

```
0/1 nodes are available: 1 Insufficient memory. preemption: 0/1 nodes are available: 1 No
preemption victims found for incoming pod.
```

3. CAS should create a new node. Confirm from UI Az Portal in Settings > Node Pool



Node pool ↑	Provisioning state ⓘ	Power state ⓘ	Scale method	Target nodes ⓘ	Ready nodes ⓘ	Autoscaling status ⓘ
userpool	Succeeded	Running	Autoscale	3	2	● Scaling up

4. Scale deployment down to 1. CAS should scale down nodes to 1. Confirm from UI.

```
kubectl scale deployment nginx-cas --replicas 1
```

5. Check CAS events and status to reveal scale-up and scale-down

```
kubectl describe cm cluster-autoscaler-status -n kube-system
```

Events:

```
  Type    Reason         Age   From                Message
  ----    ------         ----  ----                -------
  Normal  ScaleDown      15m   cluster-autoscaler  Scale-down: node aks-nodepool1-20869648-vmss000000 removed with drain
  Normal  ScaledUpGroup  11m   cluster-autoscaler  Scale-up: setting group aks-nodepool1-20869648-vmss size to 2 instead of 1
(max: 2)
  Normal  ScaledUpGroup  11m   cluster-autoscaler  Scale-up: group aks-nodepool1-20869648-vmss size set to 2 instead of 1 (max:
2)
```

or use below to watch

```
while ($true) {
    kubectl describe cm cluster-autoscaler-status -n kube-system | Select-String "Events" -Context
0,10
    Start-Sleep -Seconds 30
}
```

Module 4 – KEDA (K8S Event Driven Autoscaler)

Objective

KEDA (Kubernetes-based Event Driven Autoscaling) is an open-source component for event-driven autoscaling of workloads in Kubernetes.

How it works

- KEDA dynamically scales workloads based on the number of events they receive.
- It extends Kubernetes with a custom resource definition (CRD) called **ScaledObject**, which defines how applications should scale in response to specific events.
- Event-driven scaling is particularly useful in scenarios where workloads experience sudden traffic spikes or handle large volumes of data.

Difference between HPA and KEDA

- **HPA** (Horizontal Pod Autoscaler) is *metrics-driven* and scales workloads based on resource utilization, such as CPU and memory.
- **KEDA** is *event-driven* and scales workloads based on the number of incoming events.
- KEDA can also integrate with HPA for replica scaling.

KEDA in AKS

- AKS offers an add-on to simplify KEDA installation using the `--enable-keda` flag.
- The AKS KEDA add-on currently provides only basic common configurations; manual editing of KEDA YAML files is recommended for advanced customization.

More details can be found in this link

Step 1: Install KEDA and HTTPAddon to KEDA namespace

```
# Add the KEDA charts repo and refresh indices
helm repo add kedacore https://kedacore.github.io/charts
helm repo update

# (Optional) Verify it's there
helm repo list
helm search repo kedacore

kubectl create ns keda
helm install keda kedacore/keda -n keda
helm install http-add-on kedacore/keda-add-ons-http -n keda
```

Step 2: Increase Resource Limits

Upgrade to higher resource limit to stand stress of load generator

```
helm upgrade http-add-on kedacore/keda-add-ons-http -n keda `
  --set interceptor.resources.requests.cpu=100m `
  --set interceptor.resources.requests.memory=128Mi `
  --set interceptor.resources.limits.cpu=500m `
  --set interceptor.resources.limits.memory=512Mi
```

Step 3: Enable CAS on the userpool

```
az aks nodepool update --resource-group $rg --cluster-name $aks --name userpool --enable-cluster-autoscaler --min-count 3 --max-count 3

# if already done then just update
az aks nodepool update --resource-group $rg --cluster-name $aks --name userpool --update-cluster-autoscaler --min-count 3 --max-count 3
```

Step 4: Confirm KEDA deployment

1. Confirm overall deployment

Confirm KEDA CRDs in place, with 6 entries in total

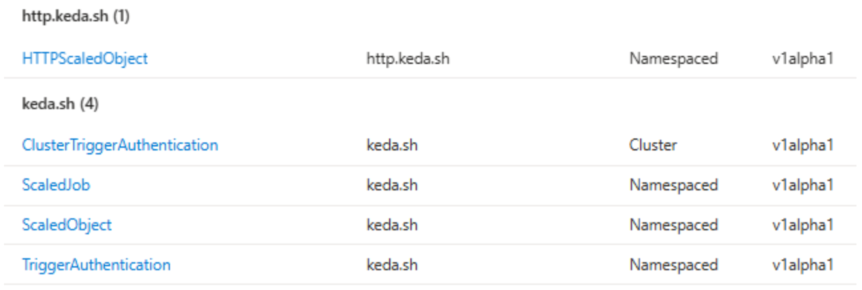
```
kubectl get crd | findstr keda
```

```
cloudeventsources.eventing.keda.sh
clustercloudeventsources.eventing.keda.sh
clustertriggerauthentications.keda.sh
httpscaledobjects.http.keda.sh
scaledjobs.keda.sh
scaledobjects.keda.sh
triggerauthentications.keda.sh
```

Also confirm metrics server and KEDA metrics is running, with 3 entries in total

```
k get pods -A  | findstr /I metrics
```

From AKS Portal > Kubernetes resources > Custom > find KEDA to confirm



http.keda.sh (1)			
HTTPScaledObject	http.keda.sh	Namespaced	v1alpha1
keda.sh (4)			
ClusterTriggerAuthentication	keda.sh	Cluster	v1alpha1
ScaledJob	keda.sh	Namespaced	v1alpha1
ScaledObject	keda.sh	Namespaced	v1alpha1
TriggerAuthentication	keda.sh	Namespaced	v1alpha1

2. Confirm KEDA deployment and pods are running

```
kubectl get deploy -n keda
```

```
NAME                                    READY   UP-TO-DATE   AVAILABLE   AGE
# http-addon
keda-add-ons-http-controller-manager    1/1     1            1           104s
keda-add-ons-http-external-scaler       3/3     3            3           104s
keda-add-ons-http-interceptor           3/3     3            3           104s
# keda-addon
keda-admission-webhooks                 1/1     1            1           6m30s
keda-operator                           1/1     1            1           6m30s
keda-operator-metrics-apiserver         1/1     1            1           6m30s
```

```
k get pods -n keda
```

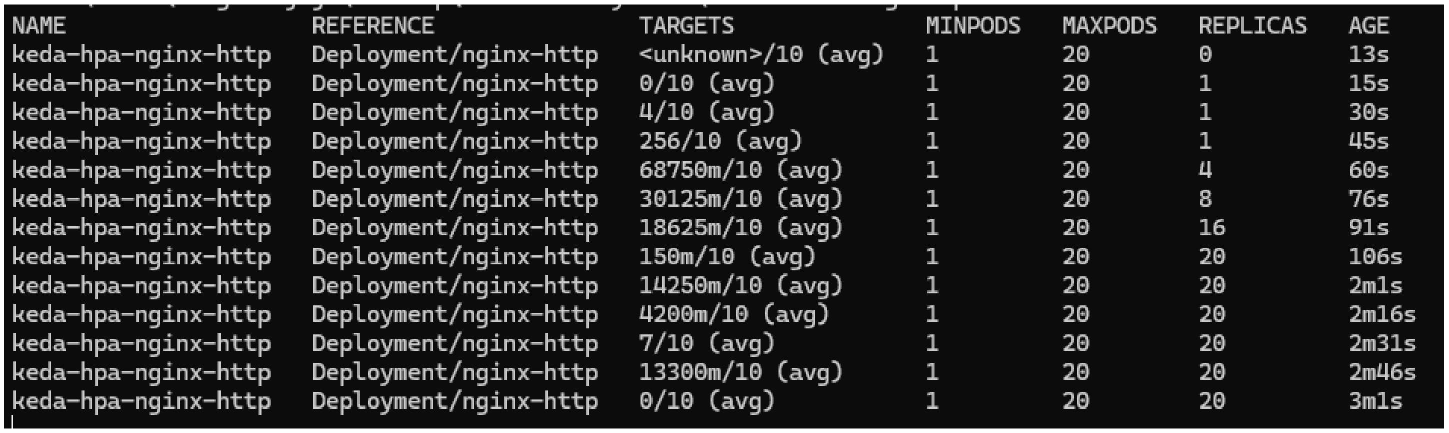
Step 5: Deploy application

```
@"
# Deployment of nginx, to be scaled by KEDA HTTP add-on
apiVersion: apps/v1
kind: Deployment
metadata:
  name: nginx-http
spec:
  replicas: 1
  selector:
    matchLabels:
      app: nginx-http
  template:
    metadata:
      labels:
        app: nginx-http
    spec:
      containers:
        - name: nginx-http
          image: nginx
          resources:
            limits:
              memory: 256Mi
              cpu: 200m
          ports:
            - containerPort: 80
---
# Expose nginx deployment as a service to connect and generate load
apiVersion: v1
kind: Service
metadata:
  name: nginx-http-service
  labels:
    app: nginx-http
spec:
  selector:
    app: nginx-http
  ports:
    - protocol: TCP
      port: 8082
      targetPort: 80
---
# HTTPScaledObject per CRD (http.keda.sh/v1alpha1) will scale the nginx-http deployment
# with 10 requests per second per pod and min replicas 1 and max replicas 20
apiVersion: http.keda.sh/v1alpha1
kind: HTTPScaledObject
metadata:
  name: nginx-http
spec:
  hosts:
    - nginx-http.default.svc.cluster.local
  targetPendingRequests: 10
  scaleTargetRef:
    service: nginx-http-service
    port: 8082
    name: nginx-http            # <-- explicit workload name
    kind: Deployment            # <-- explicit kind
    apiVersion: apps/v1         # <-- explicit apiVersion
  replicas:
    min: 1
    max: 20
---
# Load generator to create HTTP traffic via the KEDA HTTP interceptor proxy
apiVersion: apps/v1
kind: Deployment
metadata:
  name: load-http
  labels:
    app: load-http
spec:
  replicas: 5
  selector:
    matchLabels:
      app: load-http
  template:
    metadata:
      labels:
        app: load-http
    spec:
      containers:
        - name: load-http
          image: ubuntu:22.04
          imagePullPolicy: Always
          command: ["/bin/sh"]
          args:
            - "-c"
            - >
              apt-get update &&
              apt-get install -y siege &&
              siege -d 1 -c 60 -t 360s
              -H "Host: nginx-http.default.svc.cluster.local"
              http://keda-add-ons-http-interceptor-proxy.keda.svc.cluster.local:8080
          resources:
            limits:
              memory: 128Mi
              cpu: 500m
"@ | Out-File -FilePath keda-test.yaml -Encoding utf8

kubectl apply -f keda-test.yaml
```

Step 6: Confirm HPA is in place

HPA gets created by the HTTP scale object and dynamically scales up replicas, up to the max, before scaling them down.

```
k get hpa -w
```



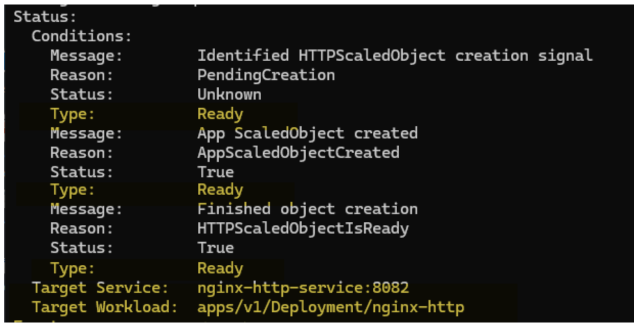
```
NAME                  REFERENCE                 TARGETS              MINPODS   MAXPODS   REPLICAS   AGE
keda-hpa-nginx-http   Deployment/nginx-http     <unknown>/10 (avg)   1         20        0          13s
keda-hpa-nginx-http   Deployment/nginx-http     0/10 (avg)           1         20        1          15s
keda-hpa-nginx-http   Deployment/nginx-http     4/10 (avg)           1         20        1          30s
keda-hpa-nginx-http   Deployment/nginx-http     256/10 (avg)         1         20        1          45s
keda-hpa-nginx-http   Deployment/nginx-http     68750m/10 (avg)      1         20        4          60s
keda-hpa-nginx-http   Deployment/nginx-http     30125m/10 (avg)      1         20        8          76s
keda-hpa-nginx-http   Deployment/nginx-http     18625m/10 (avg)      1         20        16         91s
keda-hpa-nginx-http   Deployment/nginx-http     150m/10 (avg)        1         20        20         106s
keda-hpa-nginx-http   Deployment/nginx-http     14250m/10 (avg)      1         20        20         2m1s
keda-hpa-nginx-http   Deployment/nginx-http     4200m/10 (avg)       1         20        20         2m16s
keda-hpa-nginx-http   Deployment/nginx-http     7/10 (avg)           1         20        20         2m31s
keda-hpa-nginx-http   Deployment/nginx-http     13300m/10 (avg)      1         20        20         2m46s
keda-hpa-nginx-http   Deployment/nginx-http     0/10 (avg)           1         20        20         3m1s
```

```
k get httpscaledobjects.http.keda.sh
```

```
NAME         TARGETWORKLOAD   TARGETSERVICE             MINREPLICAS   MAXREPLICAS   AGE   ACTIVE
nginx-http   //               nginx-http-service:8082   1             20            72s   True
```

Check ScaleTargetRef if its pointing to right app and Status shows Ready

```
k describe httpscaledobjects.http.keda.sh nginx-http
```



```
Status:
  Conditions:
    Message:        Identified HTTPScaledObject creation signal
    Reason:         PendingCreation
    Status:         Unknown
    Type:           Ready
    Message:        App ScaledObject created
    Reason:         AppScaledObjectCreated
    Status:         True
    Type:           Ready
    Message:        Finished object creation
    Reason:         HTTPScaledObjectIsReady
    Status:         True
    Type:           Ready
  Target Service:   nginx-http-service:8082
  Target Workload:  apps/v1/Deployment/nginx-http
```

Scale down load generator when replicas hit peak

```
k scale deploy load-http --replicas 0
```

What is an HTTPScaledObject?

An **HTTPScaledObject** is a KEDA **Custom Resource Definition (CRD)** specifically designed to handle HTTP-based autoscaling scenarios. It works alongside the KEDA **HTTP Add-on**, which provides an HTTP proxy and queue mechanism for routing traffic.

In short, while a regular ScaledObject reacts to *events* (like messages in a queue, Kafka topics, or database triggers), an HTTPScaledObject reacts to **incoming HTTP requests**, allowing autoscaling based directly on web traffic.

How it Helps

1. **Event-driven scaling for HTTP traffic**
   - Traditional HPA can only scale pods based on CPU/memory metrics.
   - With HTTPScaledObject, KEDA can scale services based on the actual volume of HTTP requests, which is often a more direct measure of demand.
2. **Handles sudden traffic spikes**
   - If your app receives a sudden burst of HTTP requests, the KEDA HTTP Add-on queues the requests temporarily and then scales out the application by creating more pods (based on the HTTPScaledObject config).
   - This prevents overload while ensuring requests are not dropped.
3. **Smooth integration with existing services**
   - You don't need to modify your application code to work with queues.
   - The add-on acts as a proxy that routes HTTP traffic through KEDA's scaling logic.
4. **Fine-grained scaling rules**
   - With HTTPScaledObject, you can define scaling thresholds such as *concurrent requests per pod*.
   - Example: Scale up when more than 50 concurrent HTTP requests are being handled per pod.

Example Use Case

Let's say you have a microservice API that usually handles 100 requests per second but occasionally spikes to 1,000 requests per second.

- **Without KEDA/HTTPScaledObject**: You'd either over-provision (wasting resources) or risk outages during spikes.
- **With HTTPScaledObject**: KEDA scales pods dynamically when the incoming HTTP traffic crosses thresholds you've defined. Once traffic drops, it scales back down—optimizing both performance and cost.

HTTPScaledObject extends KEDA's event-driven model to HTTP workloads, enabling efficient, responsive autoscaling of web applications without requiring extra middleware or manual scaling.

Module 5 – NAP (Node Auto Provisioning)

Link

Prep Work

Since NAP requires a newer version of the Azure CLI. Run below steps to get the required version.

```
winget uninstall Microsoft.AzureCLI
winget install Microsoft.AzureCLI --version 2.76.0

# Test
az --version
```

Step 0 : Cluster creation

```
$id    = "55244005"  # Set a different Id than the one created in Module 1. This will allow a
second AKS creation.
$loc   = "eastus2"
$RESOURCE_GROUP_NAME = "azure-rg"
$CLUSTER_NAME = "aks-$id"
$aks
$rg

$VM_SKU  = "Standard_A2m_v2"
```

Create cluster if none exists with NAP enabled

```
az aks create --nodepool-name nodepool --node-count 2 --generate-ssh-keys --node-vm-size $VM_SKU --
name $CLUSTER_NAME --resource-group $RESOURCE_GROUP_NAME --node-provisioning-mode Auto --network-
plugin azure  --network-plugin-mode overlay --network-dataplane cilium

az aks get-credentials --resource-group $RESOURCE_GROUP_NAME --name $CLUSTER_NAME
```

Step 1: Cluster stop (at the end)

Set CPU limits to 0 and issue aks stop

```
kubectl patch nodepool.karpenter.sh default --type merge --patch '{\"spec\":{\"limits\":{\"cpu\":\"0\"}}}'
kubectl patch nodepool.karpenter.sh system-surge --type merge --patch '{\"spec\":{\"limits\":{\"cpu\":\"0\"}}}'
az aks update -g $RESOURCE_GROUP_NAME -n $CLUSTER_NAME --node-provisioning-mode Manual
az aks stop  -g $RESOURCE_GROUP_NAME -n $CLUSTER_NAME
```

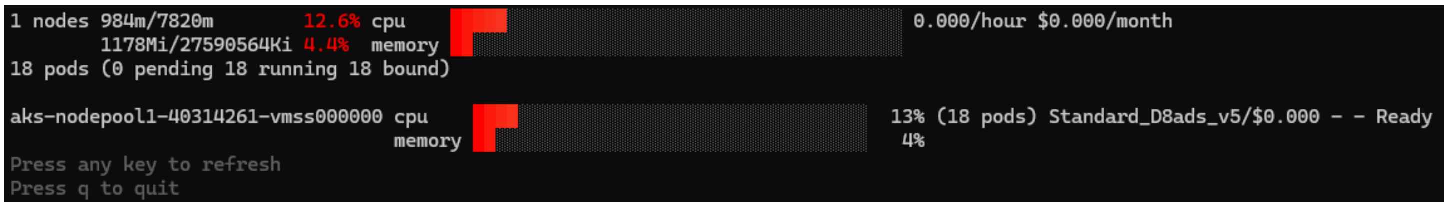
Restart aks

```
az aks start  -g $RESOURCE_GROUP_NAME -n $CLUSTER_NAME
az aks update -g $RESOURCE_GROUP_NAME -n $CLUSTER_NAME --node-provisioning-mode Auto
```

Step 2: Install AKS Node viewer

Use this link for steps involved to install AKS-Node-Viewer. Once done run using below command. Currently 0 pods are pending with no application workloads.

```
aks-node-viewer --resources cpu,memory
```



```
1 nodes 984m/7820m          12.6% cpu                                              0.000/hour $0.000/month
        1178Mi/27590564Ki  4.4%  memory
18 pods (0 pending 18 running 18 bound)

aks-nodepool1-40314261-vmss000000 cpu                                              13% (18 pods) Standard_D8ads_v5/$0.000 - - Ready
                                  memory                                            4%
Press any key to refresh
Press q to quit
```

Validate

```
k get crd | findstr /i carpenter
aksnodeclasses.karpenter.azure.com
```

```
nodeclaims.karpenter.sh
nodepools.karpenter.sh

# 2 pools: general purpose for workloads and surge used to provide critical components with fast
scaling
k get nodepools
NAME           NODECLASS
default        default
system-surge   default

# default spec shows support for arch=amd64, os=linux, sku=D family, capacity=on-demand
k get nodepool default -o yaml

# surge spec shows taint for CriticalAddonsOnly=true, arch=amd64, os=linux
k get nodepools system-surge -o yaml

# AKS custom CRD for Karpenter that defines scale settings
k get aksnodeclass
NAME
default
system-surge

# Spec defines image=ubuntu, osdisksize=128gb, arch=amd64
k get aksnodeclass default -o yaml
k get aksnodeclass system-surge  -o yaml
```

Events

```
k get events -A --field-selector source=karpenter -w
```

Lab1 – Trigger NAP based on memory requests

Step 1: Deploy 10 replicas

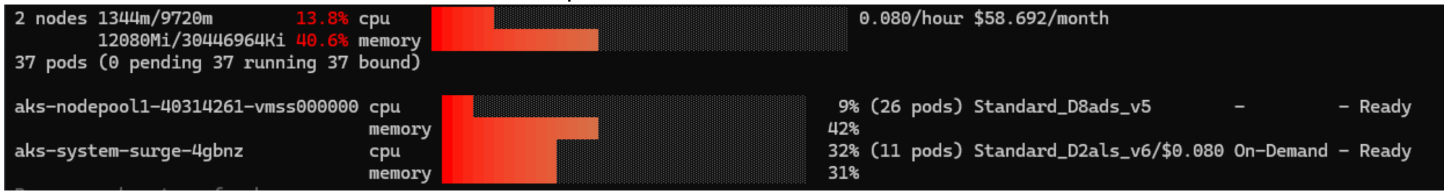
Use the deployment file below to create 10 replicas each of 1Gi.

```
$yamlContent = @"
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-deploy
spec:
  replicas: 11
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      containers:
        - name: test
          image: nginx
          resources:
            requests:
              memory: "1Gi"
"@
$yamlContent | Out-File -FilePath "nap-lab1-deployment.yaml" -Encoding utf8

k apply -f .\nap-lab1-deployment.yaml
```

If no pending pods > `k scale deployment test-deploy –replicas=xx`

Step 2: Validate the surge pool

This increase should accommodate the new pods



```
2 nodes 1344m/9720m          13.8% cpu                                   0.080/hour $58.692/month
        12080Mi/30446964Ki  40.6% memory
37 pods (0 pending 37 running 37 bound)

aks-nodepool1-40314261-vmss000000 cpu                                      9% (26 pods) Standard_D8ads_v5          -          - Ready
                                  memory                                  42%
aks-system-surge-4gbnz            cpu                                     32% (11 pods) Standard_D2als_v6/$0.080 On-Demand - Ready
                                  memory                                  31%
```

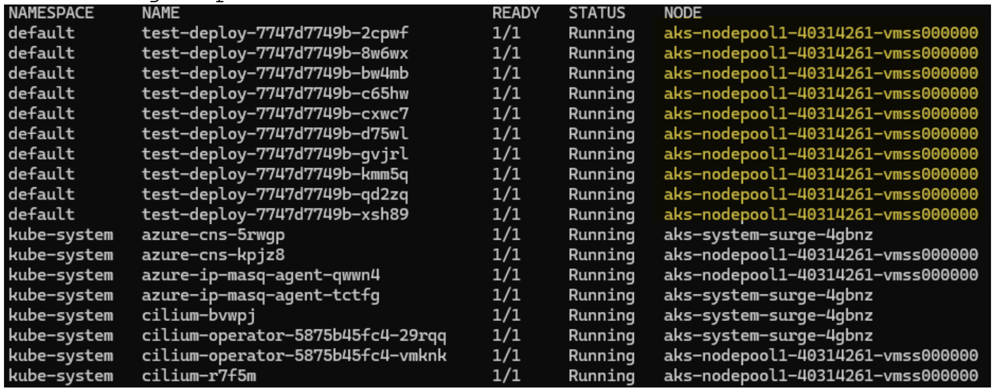
Step 3: Confirm the Node claim

Validate using below nodeclaim and pods listing that NAP is functional. Nodepool holds all app pods and system pods are spread across.

```
kubectl get nodeclaim -o wide
```

```
NAME                     TYPE                CAPACITY     ZONE         NODE                        READY
AGE     NODEPOOL          NODECLASS    DRIFTED
system-surge-4gbnz       Standard_D2als_v6   on-demand    eastus2-3    aks-system-surge-4gbnz      True
7m1s    system-surge      default
```

```
kubectl get pods -A -o wide
```



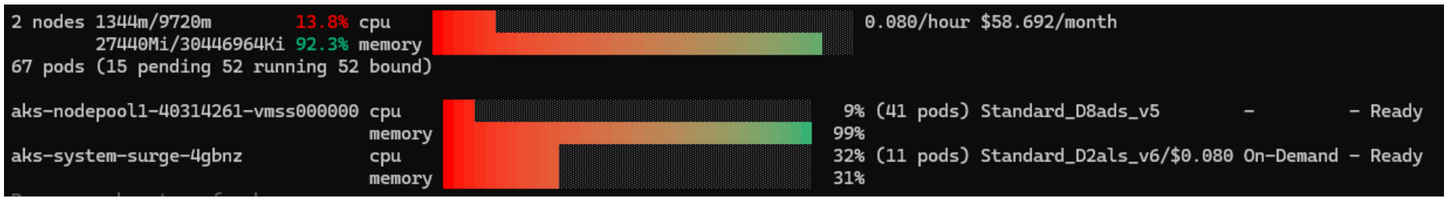
```
NAMESPACE    NAME                             READY   STATUS    NODE
default      test-deploy-7747d7749b-2cpwf     1/1     Running   aks-nodepool1-40314261-vmss000000
default      test-deploy-7747d7749b-8w6wx     1/1     Running   aks-nodepool1-40314261-vmss000000
default      test-deploy-7747d7749b-bw4mb     1/1     Running   aks-nodepool1-40314261-vmss000000
default      test-deploy-7747d7749b-c65hw     1/1     Running   aks-nodepool1-40314261-vmss000000
default      test-deploy-7747d7749b-cxwc7     1/1     Running   aks-nodepool1-40314261-vmss000000
default      test-deploy-7747d7749b-d75wl     1/1     Running   aks-nodepool1-40314261-vmss000000
default      test-deploy-7747d7749b-gvjrl     1/1     Running   aks-nodepool1-40314261-vmss000000
default      test-deploy-7747d7749b-kmm5q     1/1     Running   aks-nodepool1-40314261-vmss000000
default      test-deploy-7747d7749b-qd2zq     1/1     Running   aks-nodepool1-40314261-vmss000000
default      test-deploy-7747d7749b-xsh89     1/1     Running   aks-nodepool1-40314261-vmss000000
kube-system  azure-cns-5rwgp                  1/1     Running   aks-system-surge-4gbnz
kube-system  azure-cns-kpjz8                  1/1     Running   aks-nodepool1-40314261-vmss000000
kube-system  azure-ip-masq-agent-qwwn4        1/1     Running   aks-nodepool1-40314261-vmss000000
kube-system  azure-ip-masq-agent-tctfg        1/1     Running   aks-system-surge-4gbnz
kube-system  cilium-bvwpj                     1/1     Running   aks-system-surge-4gbnz
kube-system  cilium-operator-5875b45fc4-29rqq 1/1     Running   aks-system-surge-4gbnz
kube-system  cilium-operator-5875b45fc4-vmknk 1/1     Running   aks-nodepool1-40314261-vmss000000
kube-system  cilium-r7f5m                     1/1     Running   aks-nodepool1-40314261-vmss000000
```

Confirm from AKS Portal > Node Pools > Nodes. In this case, NAP decides the next fit as **Standary_D2als_v6** created on-demand.

```
k get nodeclaim system-surge-4gbnz -o yaml # shows the instance-types available
```

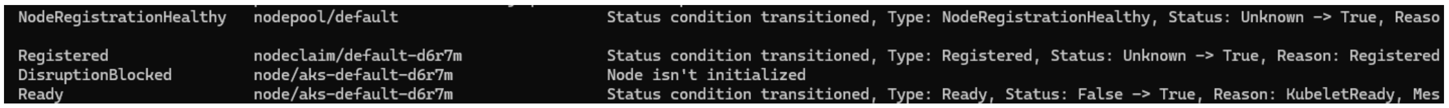
Step 4: Scale to 40 replicas which leads to 15 pending

```
k scale deploy test-deploy --replicas=40
```



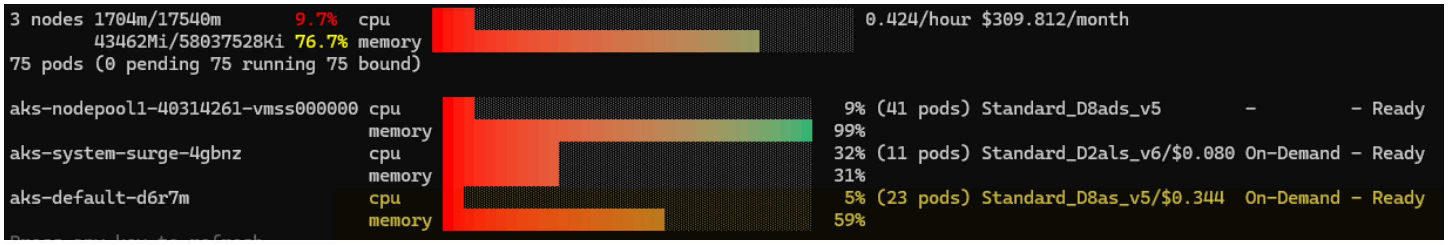
```
2 nodes 1344m/9720m          13.8% cpu                                   0.080/hour $58.692/month
        27440Mi/30446964Ki  92.3% memory
67 pods (15 pending 52 running 52 bound)

aks-nodepool1-40314261-vmss000000 cpu                                      9% (41 pods) Standard_D8ads_v5          -          - Ready
                                  memory                                  99%
aks-system-surge-4gbnz            cpu                                     32% (11 pods) Standard_D2als_v6/$0.080 On-Demand - Ready
                                  memory                                  31%
```

```
k get events -A --field-selector source=karpenter -w  # new node created
```



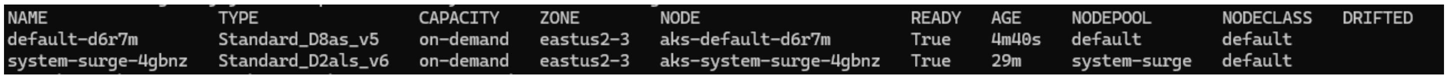
```
 NodeRegistrationHealthy   nodepool/default                  Status condition transitioned, Type: NodeRegistrationHealthy, Status: Unknown -> True, Reaso

 Registered                nodeclaim/default-d6r7m           Status condition transitioned, Type: Registered, Status: Unknown -> True, Reason: Registered
 DisruptionBlocked         node/aks-default-d6r7m            Node isn't initialized
 Ready                     node/aks-default-d6r7m            Status condition transitioned, Type: Ready, Status: False -> True, Reason: KubeletReady, Mes
```

NAP decides another **Standard_D8as_v5 is needed**!



```
3 nodes 1704m/17540m        9.7%  cpu                                          0.424/hour $309.812/month
        43462Mi/58037528Ki 76.7% memory
75 pods (0 pending 75 running 75 bound)

aks-nodepool1-40314261-vmss000000 cpu                                 9% (41 pods) Standard_D8ads_v5         -          - Ready
                                  memory                             99%
aks-system-surge-4gbnz            cpu                                32% (11 pods) Standard_D2als_v6/$0.080 On-Demand - Ready
                                  memory                             31%
aks-default-d6r7m                 cpu                                 5% (23 pods) Standard_D8as_v5/$0.344  On-Demand - Ready
                                  memory                             59%
```

k get nodeclaim -o wide



```
NAME                TYPE               CAPACITY    ZONE        NODE                     READY   AGE     NODEPOOL       NODECLASS   DRIFTED
default-d6r7m       Standard_D8as_v5   on-demand   eastus2-3   aks-default-d6r7m        True    4m40s   default        default
system-surge-4gbnz  Standard_D2als_v6  on-demand   eastus2-3   aks-system-surge-4gbnz   True    29m     system-surge   default
```

Step 5: Confirm all pods ready

k get deployments -o wide

```
NAME          READY   UP-TO-DATE   AVAILABLE   AGE   CONTAINERS   IMAGES   SELECTOR
test-deploy   40/40   40           40          32m   test         nginx    app=test
```

```
# Confirm app pods spread across aks-nodepool1 and aks-default
k get pods -l app=test -o wide
```

Step 6: Downscale so NAP re-optimizes

Scale to 1 replicas which leads to reduction of all nodes

k scale deploy test-deploy --replicas=1

Lab2 – Using custom node class

This lab uses custom AKSNodeClass, which is a template for VM with infra details. Also custom Nodepool, using that NodeClass, as the scheduler-facing policy for when and how to create nodes. So `AKSNodeClass` to describe *how to create Azure nodes*, and a `NodePool` to tell Karpenter *when/why/how many* nodes to create from that class.

Step 1: Create custom AKSNodeClass

```
$yamlContent = @"
apiVersion: karpenter.azure.com/v1beta1
kind: AKSNodeClass
metadata:
  name: nap-aksnodeclass
spec:
  imageFamily: AzureLinux
  osDiskSizeGB: 128
  tags:
    env: prod
"@
$yamlContent | Out-File -FilePath "nap-lab2-nodeclass.yaml" -Encoding utf8

k apply -f .\nap-lab2-nodeclass.yaml
```

Step 2 : Validate Node class

```
# Check that Image=Azure Linux, Tag Env=Prod, OSdisk=128Gb
k get aksnodeclass
k describe aksnodeclass nap-aksnodeclass
```

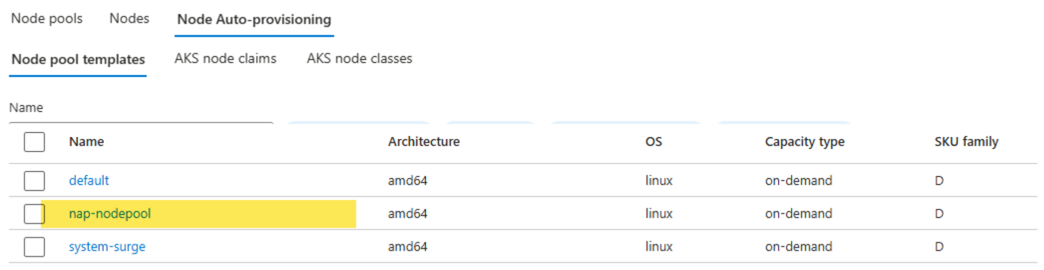
Step 3 : Create custom NodePool

Deploying the below YAML should get Nodepool created with weight=10 (higher PRI than default with weight=0)

```
$yamlContent = @"
apiVersion: karpenter.sh/v1
kind: NodePool
metadata:
  name: nap-nodepool
spec:
  template:
    spec:
      nodeClassRef:
        group: karpenter.azure.com
        kind: AKSNodeClass
        name: nap-aksnodeclass
      requirements:
        - key: kubernetes.io/arch
          operator: In
          values: ["amd64"]
        - key: kubernetes.io/os
          operator: In
          values: ["linux"]
        - key: karpenter.sh/capacity-type
          operator: In
          values: ["on-demand"]
        - key: karpenter.azure.com/sku-family
          operator: In
          values: ["D"]
      expireAfter: Never
  limits:
    cpu: 100
  disruption:
    consolidationPolicy: WhenEmptyOrUnderutilized
    consolidateAfter: 0s
  weight: 10
"@
```

```
$yamlContent | Out-File -FilePath "nap-lab2-nodepool.yaml" -Encoding utf8
```

`k apply -f .\nap-lab2-nodepool.yaml`



Name	Architecture	OS	Capacity type	SKU family
default	amd64	linux	on-demand	D
nap-nodepool	amd64	linux	on-demand	D
system-surge	amd64	linux	on-demand	D

Confirm if running

```
kubectl get nodepool
kubectl describe nodepool nap-nodepool
```

Step 4 : Create additional replicas

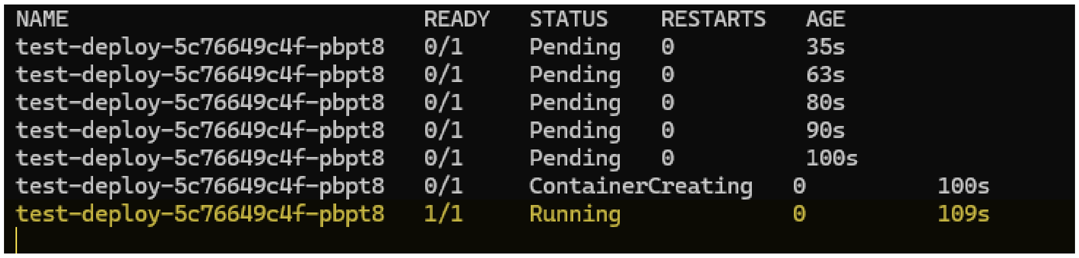
Apply deployment to create 1 replica of 10vCpu

```
$yamlContent = @"
apiVersion: apps/v1
kind: Deployment
metadata:
  name: test-deploy
spec:
  replicas: 1
  selector:
    matchLabels:
      app: test
  template:
    metadata:
      labels:
        app: test
    spec:
      containers:
        - name: test
          image: nginx
          resources:
            requests:
              cpu: "10000m"
"@
$yamlContent | Out-File -FilePath "nap-lab2-deployment.yaml" -Encoding utf8
```

```
k apply -f .\nap-lab2-deployment.yaml
```

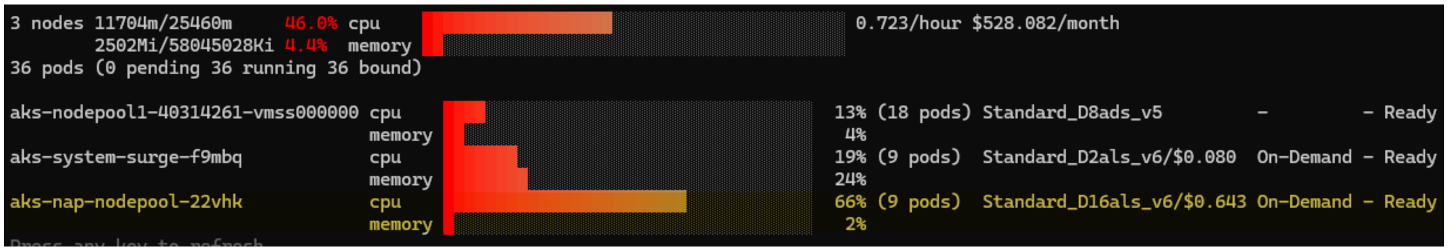
This lead to 1 replica of 10vCPU with Pending as Node not available. Finally goes to Running on new Node.

```
k get pods -w
```



```
NAME                          READY   STATUS    RESTARTS   AGE
test-deploy-5c76649c4f-pbpt8  0/1     Pending   0          35s
test-deploy-5c76649c4f-pbpt8  0/1     Pending   0          63s
test-deploy-5c76649c4f-pbpt8  0/1     Pending   0          80s
test-deploy-5c76649c4f-pbpt8  0/1     Pending   0          90s
test-deploy-5c76649c4f-pbpt8  0/1     Pending   0          100s
test-deploy-5c76649c4f-pbpt8  0/1     ContainerCreating   0          100s
test-deploy-5c76649c4f-pbpt8  1/1     Running             0          109s
```

AKS-Node-viewer shows the new node using **D16als_v6** which is best fit for Pod spec. Since nap-nodepool has higher weight than default, it was picked over default pool template.



```
3 nodes 11704m/25460m     46.0% cpu                                          0.723/hour $528.082/month
        2502Mi/58045028Ki 4.4%  memory
36 pods (0 pending 36 running 36 bound)

aks-nodepool1-40314261-vmss000000 cpu                                   13% (18 pods) Standard_D8ads_v5         -          - Ready
                                  memory                                 4%
aks-system-surge-f9mbq            cpu                                   19% (9 pods)  Standard_D2als_v6/$0.080  On-Demand - Ready
                                  memory                                24%
aks-nap-nodepool-22vhk            cpu                                   66% (9 pods)  Standard_D16als_v6/$0.643 On-Demand - Ready
                                  memory                                 2%
Press any key to refresh
```

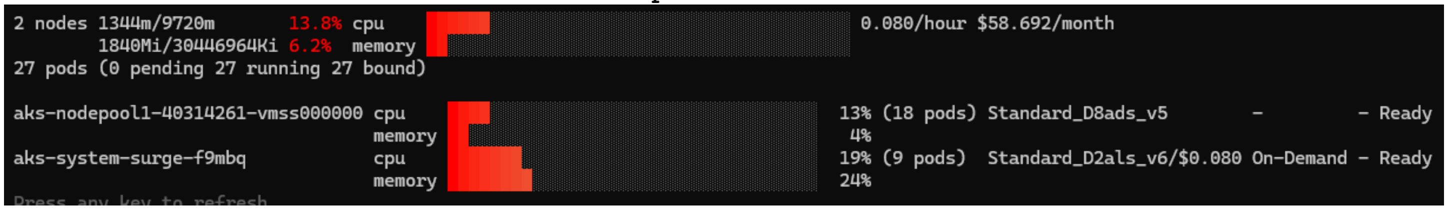
Step 5 : Watch events take place

```
k get events -A --field-selector source=karpenter -w
```

Step 6 : Delete Deployment

```
k delete -f .\nap-lab2-deployment.yaml
```

```
# Watch deletion of the node created for the pod
```



```
2 nodes 1344m/9720m       13.8% cpu                                          0.080/hour $58.692/month
        1840Mi/30446964Ki 6.2%  memory
27 pods (0 pending 27 running 27 bound)

aks-nodepool1-40314261-vmss000000 cpu                                   13% (18 pods) Standard_D8ads_v5        -          - Ready
                                  memory                                 4%
aks-system-surge-f9mbq            cpu                                   19% (9 pods)  Standard_D2als_v6/$0.080 On-Demand - Ready
                                  memory                                24%
Press any key to refresh
```

Cleanup

(Stop! Only do this at the end of all Modules)

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

Module 6 – Health Lab

The flow below uses one tiny web app (nginx) + one toolbox pod (busybox) to exercise scheduling, controllers, service discovery, Konnectivity, policy, image pulls, and observability

Step 1: Setup

```
$NS = "health-lab"
$RESOURCE_GROUP = "your rg"
$AKS_NAME = "your rg"

# Create the lab namespace
kubectl get ns $NS 1>$null 2>$null; if ($LASTEXITCODE -ne 0) { kubectl create ns $NS | Out-Null }
kubectl config set-context --current --namespace=$NS
```

Step 2: Deploy app

Deploy "probe" app (to monitor health, controllers, image pulls, service discovery, observability)

```
@"
apiVersion: apps/v1
kind: Deployment
metadata: { name: web, namespace: $NS }
spec:
  replicas: 2
  selector: { matchLabels: { app: web } }
  template:
    metadata: { labels: { app: web } }
    spec:
      containers:
      - name: nginx
        image: nginx:1.25-alpine
        ports: [{ containerPort: 80 }]
        resources:
          requests: { cpu: "50m", memory: "64Mi" }
          limits:   { cpu: "200m", memory: "128Mi" }
---
apiVersion: v1
kind: Service
metadata: { name: web, namespace: $NS }
spec:
  selector: { app: web }
  ports: [{ port: 80, targetPort: 80 }]
  type: ClusterIP
"@ | kubectl apply -f -

kubectl -n $NS delete pod bb --ignore-not-found | Out-Null
kubectl -n $NS run bb --image=busybox:1.36 --restart=Never --command -- sleep 3600
```

Step 3 : Check Node & Pod health (readiness, pressure, taints)

Two replicas must schedule and become Ready, so any node issues show up immediately.

```
kubectl get nodes -o wide
kubectl -n $NS get deploy/web -o wide
kubectl -n $NS get pods -l app=web -o wide
kubectl -n $NS describe deploy/web | Select-String -Context 2,2 "Conditions|Events"
```

Pass if nodes are Ready, web shows AVAILABLE 2/2, pods Running, no crashloops.

Step 4 : Check Controllers & Rollout behavior (Deployment/ReplicaSet, events)

The app helps here to scale/roll and watch controller health + events.

```
# Scale up (simulate real rollout pressure)
kubectl -n $NS scale deploy/web --replicas=4
kubectl -n $NS rollout status deploy/web --timeout=90s
kubectl -n $NS get rs -o wide
kubectl -n $NS get events --sort-by=.lastTimestamp | Select-String -Context 0,2 "web"

# Scale down on completion
kubectl -n $NS scale deploy/web --replicas=1
kubectl get pods # to confirm
```

Pass if rollout completes, RS shows correct desired/ready, no repeating failures in events.

Step 5 : Check Service discovery & East-West connectivity (CoreDNS + kube-proxy/eBPF)

The app helps to check for a stable service endpoint (web) that can be hit from another pod.

```
# DNS lookup + HTTP fetch from toolbox pod
kubectl -n $NS exec bb -- sh -c "nslookup web.$NS.svc.cluster.local || true"
kubectl -n $NS exec bb -- sh -c "wget -qO- http://web"
```

Validate if DNS resolves to ClusterIP and wget returns nginx HTML.

To skip DNS entirely, just keep the wget http://web check; it hits service IP via cluster DNS name cached by BusyBox or use curl http://$CLUSTER_IP by reading kubectl -n $NS get svc web -o jsonpath='{.spec.clusterIP}'

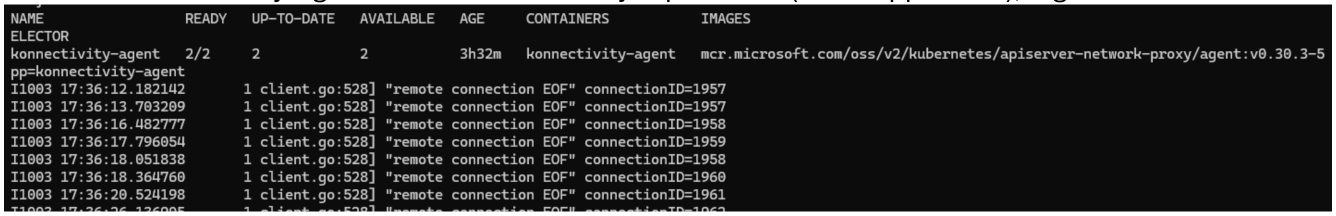
```
kubectl exec bb -- sh -c "wget -qO- http://$(kubectl -n $NS get svc web -o jsonpath='{.spec.clusterIP}')"
```

Step 6 : Check Control plane ↔ node path (Konnectivity)

The app helps to check if API↔node path is broken, your **rollout/status/scale** above would flake; here's the direct health check.

```
$hasKon = kubectl -n kube-system get deploy/konnectivity-agent --ignore-not-found
if ($hasKon) {
  kubectl -n kube-system get deploy/konnectivity-agent -o wide
  kubectl -n kube-system logs -l app=konnectivity-agent --tail=10
} else {
  Write-Host "Konnectivity not deployed (may be normal for your CNI/APIServer integration)."
}
```

Validate if konnectivity-agent exists **and** has readyReplicas > 0 (when applicable), logs are clean.

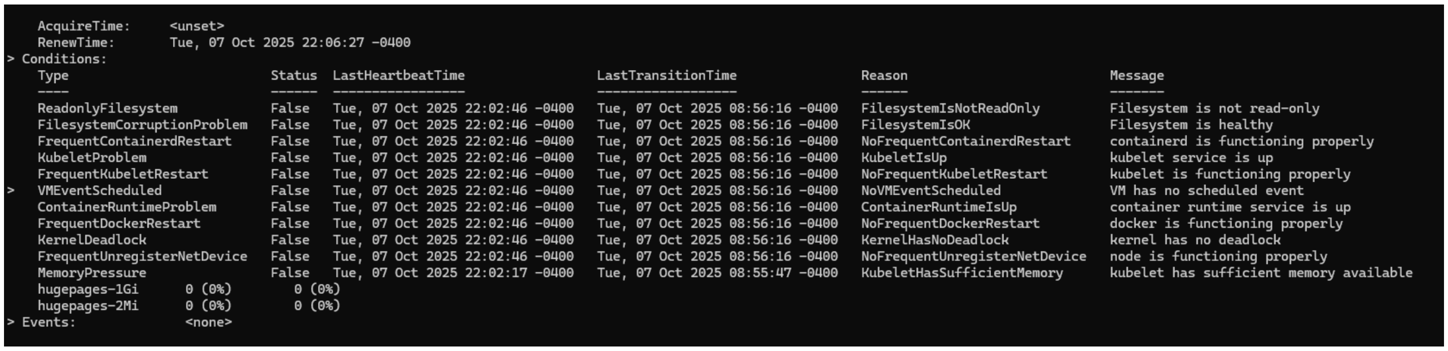


```
NAME                 READY   UP-TO-DATE   AVAILABLE   AGE     CONTAINERS           IMAGES
ELECTOR
konnectivity-agent   2/2     2            2           3h32m   konnectivity-agent   mcr.microsoft.com/oss/v2/kubernetes/apiserver-network-proxy/agent:v0.30.3-5
pp=konnectivity-agent
I1003 17:36:12.182142       1 client.go:528] "remote connection EOF" connectionID=1957
I1003 17:36:13.703209       1 client.go:528] "remote connection EOF" connectionID=1957
I1003 17:36:16.482777       1 client.go:528] "remote connection EOF" connectionID=1958
I1003 17:36:17.796054       1 client.go:528] "remote connection EOF" connectionID=1959
I1003 17:36:18.051838       1 client.go:528] "remote connection EOF" connectionID=1958
I1003 17:36:18.364760       1 client.go:528] "remote connection EOF" connectionID=1960
I1003 17:36:20.524198       1 client.go:528] "remote connection EOF" connectionID=1961
```

Step 7 : Check Kubelet / Node Problem Detector (conditions & events)

The app helps to check if web pods run across nodes. Any node issues surface as conditions/events that can be correlated to the app.

```
# Quick condition scan for one node (repeat for others as needed)
$node = kubectl get nodes -o jsonpath='{.items[0].metadata.name}'
kubectl describe node $node | Select-String -Context 2,5 "Conditions|Events"
```



```
    AcquireTime:     <unset>
    RenewTime:       Tue, 07 Oct 2025 22:06:27 -0400
> Conditions:
    Type                          Status  LastHeartbeatTime                 LastTransitionTime                Reason                          Message
    ----                          ------  -----------------                 ------------------                ------                          -------
    ReadonlyFilesystem            False   Tue, 07 Oct 2025 22:02:46 -0400   Tue, 07 Oct 2025 08:56:16 -0400   FilesystemIsNotReadOnly         Filesystem is not read-only
    FilesystemCorruptionProblem   False   Tue, 07 Oct 2025 22:02:46 -0400   Tue, 07 Oct 2025 08:56:16 -0400   FilesystemIsOK                  Filesystem is healthy
    FrequentContainerdRestart     False   Tue, 07 Oct 2025 22:02:46 -0400   Tue, 07 Oct 2025 08:56:16 -0400   NoFrequentContainerdRestart     containerd is functioning properly
    KubeletProblem                False   Tue, 07 Oct 2025 22:02:46 -0400   Tue, 07 Oct 2025 08:56:16 -0400   KubeletIsUp                     kubelet service is up
    FrequentKubeletRestart        False   Tue, 07 Oct 2025 22:02:46 -0400   Tue, 07 Oct 2025 08:56:16 -0400   NoFrequentKubeletRestart        kubelet is functioning properly
>   VMEventScheduled              False   Tue, 07 Oct 2025 22:02:46 -0400   Tue, 07 Oct 2025 08:56:16 -0400   NoVMEventScheduled              VM has no scheduled event
    ContainerRuntimeProblem       False   Tue, 07 Oct 2025 22:02:46 -0400   Tue, 07 Oct 2025 08:56:16 -0400   ContainerRuntimeIsUp            container runtime service is up
    FrequentDockerRestart         False   Tue, 07 Oct 2025 22:02:46 -0400   Tue, 07 Oct 2025 08:56:16 -0400   NoFrequentDockerRestart         docker is functioning properly
    KernelDeadlock                False   Tue, 07 Oct 2025 22:02:46 -0400   Tue, 07 Oct 2025 08:56:16 -0400   KernelHasNoDeadlock             kernel has no deadlock
    FrequentUnregisterNetDevice   False   Tue, 07 Oct 2025 22:02:46 -0400   Tue, 07 Oct 2025 08:56:16 -0400   NoFrequentUnregisterNetDevice   node is functioning properly
    MemoryPressure                False   Tue, 07 Oct 2025 22:02:17 -0400   Tue, 07 Oct 2025 08:55:47 -0400   KubeletHasSufficientMemory      kubelet has sufficient memory available
    hugepages-1Gi      0 (0%)        0 (0%)
    hugepages-2Mi      0 (0%)        0 (0%)
> Events:              <none>
```

```
# Look for OOMKilling, FrequentKubeletRestart, DiskPressure, etc.
kubectl get events -A --sort-by=.lastTimestamp | Select-String -Context 0,1
"Kubelet|NodeProblem|OOM|DiskPressure|MemoryPressure"
```

Pass if: no permanent-problem conditions; events aren't repeatedly warning/erroring for the nodes hosting web.

Step 8 : Check Disk I/O & throttling (workbook-style signal + inline fio)

The app gives you a **steady baseline**; fio adds a controlled burst to see if node/disk throttles under load.
If you only want a quick "show me numbers" run and see the output live:

```
kubectl -n $NS run fio --image=alpine --restart=Never -it --rm -- sh -c `
 'apk add --no-cache fio && fio --name=randrw --filename=/tmp/testfile --size=128m --time_based --
runtime=10 --rw=randrw --bs=4k --iodepth=1'
```

Interpretation of the output below: from a 10s run below shows a flat ceiling (e.g., ~6k IOPS / ~23 MiB/s at 4k) with microsecond low-ms latencies usually means **disk SKU limits**, not throttling. Long tail spikes (tens to hundreds of ms) or a drop after a few minutes suggests throttling/burst credit exhaustion.

```
read: IOPS=6020, BW=23.5MiB/s (24.7MB/s)(235MiB/10001msec)
       lat (usec): min=65, max=20293, avg=149.67, stdev=166.51
       99.00th=[  289], 99.50th=[  343], 99.90th=[ 1188], 99.95th=[ 1549],
write: IOPS=6008, BW=23.5MiB/s (24.6MB/s)(235MiB/10001msec); 0 zone resets
       lat (usec): min=3, max=214, avg= 4.61, stdev= 3.76
       99.00th=[   11], 99.50th=[   12], 99.90th=[   31], 99.95th=[   90],
```

Step 9 : Check Admission controls (Azure Policy/Gatekeeper)

To enable controls, install the Azure Policy Add-on for AKS

```
az aks enable-addons --addons azure-policy --name $AKS_NAME --resource-group $RESOURCE_GROUP
az aks update -g aks01day2-rg -n aks01day2 --network-policy <cilium or azure depending on the
dataplane in use>
```

Note: Enabling the add-on only deploys the Gatekeeper controllers.
For actual enforcement (DENY on non-compliant resources), you must also configure policies/constraints through Azure Policy or Gatekeeper templates. This setup can take ~20 minutes to sync.

The app helps to prove policies enforce/allow as intended.

```
# Gatekeeper/Policy pods healthy?
if (-not (kubectl get ns gatekeeper-system -o json 2>$null)) {
    Write-Host "Gatekeeper namespace not found. Gatekeeper not installed."
} else {
    $pods = kubectl -n gatekeeper-system get pods --no-headers 2>$null
    if ([string]::IsNullOrWhiteSpace($pods)) {
        Write-Host "Gatekeeper namespace exists but no pods are running."
    } else {
        Write-Host "Gatekeeper is installed. Pods:"
        $pods
    }
}

# Probe: Non-compliant Pod (no labels)
# Will only be denied if constraints/policies have been set up to require them.
@"
apiVersion: v1
kind: Pod
metadata:
  name: policy-probe
  namespace: $NS
spec:
  containers:
  - name: c
    image: busybox:1.36
    command: ["sh","-c","sleep 10"]
"@ | kubectl apply -f - | Write-Host

kubectl -n $NS delete pod policy-probe --ignore-not-found | Out-Null
```

Validate if Gatekeeper pods are Ready, and the webhook behavior matches your configured policy (DENY or ALLOW).

Step 10 : Image pull/ACR auth (fixing ImagePullBackOff)

The app should help to swap the image from existing to one in your ACR and observe pulls.

```
# Replace nginx with an image from your ACR to validate AcrPull (edit
<youracr>.azurecr.io/yourrepo:tag)

kubectl -n $NS set image deploy/web nginx=<youracr>.azurecr.io/yourrepo:tag
kubectl -n $NS rollout status deploy/web --timeout=90s
if ($pods = kubectl -n $NS get pods | Select-String "ImagePullBackOff") { $pods } else { Write-Host
"No ImagePullBackOff detected." }
```

```
# If you do see ImagePullBackOff, verify kubelet identity role assignment:
# $kubeletClientId = (Get-AzAks -ResourceGroupName $RESOURCE_GROUP -Name
$AKS_NAME).IdentityProfile.kubeletidentity.clientId
# $acr = Get-AzContainerRegistry -ResourceGroupName <rg> -Name <acr>
# New-AzRoleAssignment -ObjectId $kubeletClientId -Scope $acr.Id -RoleDefinitionName "AcrPull"
```

Pass if: rollout succeeds without ImagePullBackOff.

See below example for quick test

```
kubectl -n $NS set image deploy/web nginx=nginx:stable-alpine

# Wait for rollout to complete
kubectl -n $NS rollout status deploy/web --timeout=90s

# Check for any ImagePullBackOff errors
if ($pods = kubectl -n $NS get pods | Select-String "ImagePullBackOff") {
    $pods
} else {
    Write-Host "No ImagePullBackOff detected. Deployment successful."
}
```

Workload observability (Container Insights / Prometheus / Grafana)
the web pods offer a quiet, predictable metric stream for dashboards.

What to look at (brief):

- From Grafana dashboards K8S > Compute > Pod, check CPU/memory usage & **CPU throttling** percent for pods/containers hosting web.
- From Insights, check Pod restarts over time (should be 0 for web).
- **Node Disk IO workbook** while/after the fio run.

(Portal steps vary by your add-ons; with managed Prometheus/Grafana, open the standard Kubernetes dashboards and filter by namespace="$NS" and pod=~"web.".)*

Resource reservation considerations

Mi = Mebibyte (binary, 1 MiB = 1,048,576 bytes = 2^{20}), while **MB** = Megabyte (decimal, 1 MB = 1,000,000 bytes = 10^6)
Gi = Gibibyte (binary, 1 GiB = 1,073,741,824 bytes = 2^{30}), while **GB** = Gigabyte (decimal, 1 GB = 10^9 bytes)

The goal
AKS reserves resources from the nodes to ensure proper functionality and to avoid competing for resources between pods, system daemons (like kubelet and containerd), and node's operating system

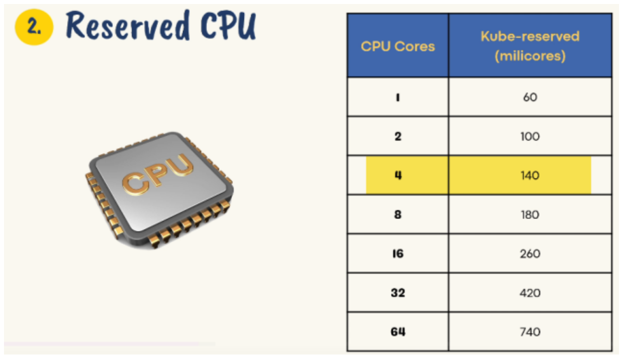
- Resource reservations lead to a discrepancy between the total and allocatable resources
- It is important to be aware of resource reservations, especially when requests and limits are set
- The resource reservations depend on multiple factors, especially node size
- The resource reservations is set with the help of "kubeReserved" kubelet parameter

```
kubectl describe node | Select-String "Hostname:|cpu:|Capacity:|Allocatable:|Kubelet Version|pods:"
```

```
Hostname:    aks-syspool-41046857-vmss00001h
Capacity:
  cpu:                      4
  pods:                     30
Allocatable:
  cpu:                      3860m
  pods:                     30
  Kubelet Version:                  v1.32.6
Non-terminated Pods:                (30 in total)

Hostname:    aks-userpool-41046857-vmss00002m
Capacity:
  cpu:                      4
  pods:                     30
Allocatable:
  cpu:                      3860m
  pods:                     30
  Kubelet Version:                  v1.32.6
Non-terminated Pods:                (29 in total)

Hostname:    aks-userpool-41046857-vmss00002n
Capacity:
  cpu:                      4
  pods:                     30
Allocatable:
  cpu:                      3860m
  pods:                     30
  Kubelet Version:                  v1.32.6
Non-terminated Pods:                (14 in total)
```

```
kubectl describe node | Select-String "Hostname:|memory:|Capacity:|Allocatable:|Kubelet Version|pods:"
```

```
Hostname:    aks-syspool-41046857-vmss00001h
Capacity:
  memory:                   16384160Ki
  pods:                     30
Allocatable:
  memory:                   15616160Ki
  pods:                     30
  Kubelet Version:                  v1.32.6
Non-terminated Pods:                (30 in total)
Hostname:    aks-userpool-41046857-vmss00002m
Capacity:
  memory:                   16384160Ki
  pods:                     30
Allocatable:
  memory:                   15616160Ki
  pods:                30
  Kubelet Version:              v1.32.6
Non-terminated Pods:            (29 in total)
Hostname:    aks-userpool-41046857-vmss00002n
Capacity:
  memory:              16384160Ki
  pods:                30
Allocatable:
  memory:              15616160Ki
  pods:                30
  Kubelet Version:              v1.32.6
Non-terminated Pods:            (14 in total)


kubectl proxy

$response = Invoke-WebRequest -Uri "http://localhost:8001/api/v1/nodes/aks-syspool-41046857-vmss00001h/proxy/configz"
$json = $response.Content | ConvertFrom-Json
$json.kubeletconfig.kubeReserved
cpu  memory pid
---  ------ ---
140m 650Mi  1000


Capacity from above is  cpu: 4vcpu - 0.14 (140m) = 3.86vcpu or 3860m core (allocatable above)

This confirms below chart
```


2. Reserved CPU

CPU Cores	Kube-reserved (milicores)
1	60
2	100
4	140
8	180
16	260
32	420
64	740

Kube-reserved optimization

Reserved memory (K8s versions 1.29 and higher)

The memory reservation is the sum of:
- kubelet daemon
- rate of memory reservation

The **kubelet daemon** requires at least 100 Mi of allocatable memory + The **rate of memory reservation** is the lesser value of:
- 20MB * Max Pods supported on the Node + 50MB
- 25% of the total system memory resources B

2 more GB are reserved for system process in Windows nodes

Manual Scale

```
kubectl get deployment/store-front
kubectl scale --replicas=2 deployment/store-front
kubectl get deployment store-front -o wide; kubectl get pods -l app=store-front -o wide
```

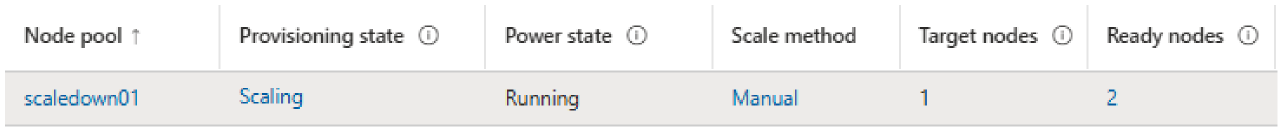
Stop vs Deallocate Nodes

Mode	Pros	Cons
Delete (Default)	• No extra cost after removal (OS/data disks deleted) • Fresh nodes provisioned during scale-up • Simpler, default behavior	• Slower scale-up (new nodes must be created) • Container images lost (pods need to re-pull) • No image caching benefit
Deallocate	• Faster scale-up (nodes resumed, not recreated) • Preserves container images (saves re-pull time) • Good for workloads needing quick recovery	• Still incurs storage charges for OS & data disks • Nodes appear **NotReady** after scale-down • Ephemeral OS disks not supported • Extra management overhead • Switching to Delete removes deallocated nodes anyway

1. Create Nodepool

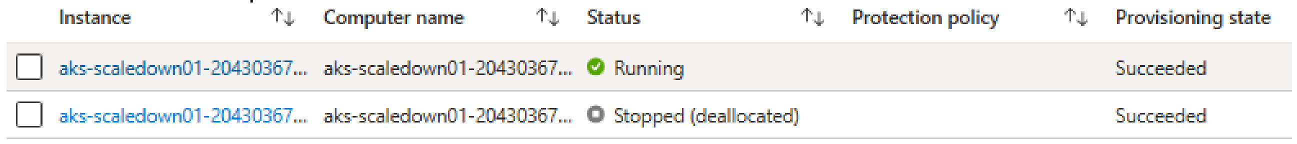
```
az aks nodepool add --node-count 2  --scale-down-mode  Deallocate --node-osdisk-type Managed `
      --name scaledown01 --cluster-name aks01day2 --resource-group aks01day2-rg
```

2. Scale down from UI to 1



Node pool ↑	Provisioning state ⓘ	Power state ⓘ	Scale method	Target nodes ⓘ	Ready nodes ⓘ
scaledown01	Scaling	Running	Manual	1	2

3. Confirm from respective VMSS that it shows as deallocated and not deleted



Instance ↑↓	Computer name ↑↓	Status ↑↓	Protection policy ↑↓	Provisioning state
aks-scaledown01-20430367...	aks-scaledown01-20430367...	✓ Running		Succeeded
aks-scaledown01-20430367...	aks-scaledown01-20430367...	Stopped (deallocated)		Succeeded

4. Confirm nodes show as Not Ready

```
> k get nodes
NAME                                 STATUS     ROLES    AGE    VERSION
aks-scaledown01-20430367-vmss000000  Ready      <none>   9m35s  v1.32.6
aks-scaledown01-20430367-vmss000001  NotReady   <none>   9m34s  v1.32.6
```

5. Change nodepool to delete and scale down to 1

```
az aks nodepool update --scale-down-mode  Delete --name scaledown01 `
      --cluster-name aks01day2 --resource-group aks01day2-rg
```

6. Delete nodepool

```
az aks nodepool delete --name scaledown01 --cluster-name aks01day2 --resource-group aks01day2-rg
```