

**Lab Manual- Cluster autoscaling in Azure Kubernetes Service (AKS) -(Lab1)**

**Prepared for**:

**Date:** 18th Dec 2023

**Prepared by:**

Document Name: Lab Manual **Document Number** AZLabn916

**Contributor:**

Contents

[1. Objective 3](#_Toc159323053)

[2. Cluster autoscaler 3](#_Toc159323054)

[3. Best practices and considerations 3](#_Toc159323055)

[4. AKS Cluster with autoscaler Metric 4](#_Toc159323056)

[5. AKS Cluster autoscaler with custom Metric 4](#_Toc159323057)

[6. Create AKS Cluster with autoscaler and Connect 6](#_Toc159323058)

[7. Update the cluster autoscaler settings 7](#_Toc159323059)

[8. Disable the cluster autoscaler on a cluster 9](#_Toc159323060)

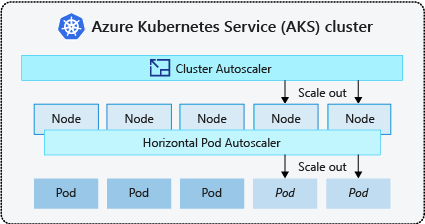
# Objective

The cluster autoscaler component watches for pods in your cluster that can't be scheduled because of resource constraints. When the cluster autoscaler detects issues, it scales up the number of nodes in the node pool to meet the application demand. It also regularly checks nodes for a lack of running pods and scales down the number of nodes as needed.

# Cluster autoscaler

Clusters often need a way to scale automatically to adjust to changing application demands, such as between workdays and evenings or weekends. AKS clusters can scale in the following ways:

* The **cluster autoscaler** periodically checks for pods that can't be scheduled on nodes because of resource constraints. The cluster then automatically increases the number of nodes. Manual scaling is disabled when you use the cluster autoscaler. For more information, see [How does scale up work?](https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#how-does-scale-up-work).
* The [**Horizontal Pod Autoscaler**](https://learn.microsoft.com/en-us/azure/aks/concepts-scale#horizontal-pod-autoscaler) uses the Metrics Server in a Kubernetes cluster to monitor the resource demand of pods. If an application needs more resources, the number of pods is automatically increased to meet the demand.
* The [**Vertical Pod Autoscaler**](https://learn.microsoft.com/en-us/azure/aks/vertical-pod-autoscaler) automatically sets resource requests and limits on containers per workload based on past usage to ensure pods are scheduled onto nodes that have the required CPU and memory resources.



# Best practices and considerations

* When implementing **availability zones with the cluster autoscaler**, we recommend using a single node pool for each zone. You can set the --balance-similar-node-groups parameter to True to maintain a balanced distribution of nodes across zones for your workloads during scale up operations. When this approach isn't implemented, scale down operations can disrupt the balance of nodes across zones.
* For **clusters with more than 400 nodes**, we recommend using Azure CNI or Azure CNI Overlay.
* To **effectively run workloads concurrently on both Spot and Fixed node pools**, consider using [priority expanders](https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#what-are-expanders). This approach allows you to schedule pods based on the priority of the node pool.
* Exercise caution when **assigning CPU/Memory requests on pods**. The cluster autoscaler scales up based on pending pods rather than CPU/Memory pressure on nodes.
* For **clusters concurrently hosting both long-running workloads, like web apps, and short/bursty job workloads**, we recommend separating them into distinct node pools with [Affinity Rules](https://learn.microsoft.com/en-us/azure/aks/operator-best-practices-advanced-scheduler#node-affinity)/[expanders](https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#what-are-expanders) or using [PriorityClass](https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/" \l "priorityclass) to help prevent unnecessary node drain or scale down operations.
* We **don't recommend making direct changes to nodes in autoscaled node pools**. All nodes in the same node group should have uniform capacity, labels, and system pods running on them.
* Nodes don't scale up if pods have a PriorityClass value below -10. Priority -10 is reserved for [overprovisioning pods](https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#how-can-i-configure-overprovisioning-with-cluster-autoscaler). For more information, see [Using the cluster autoscaler with Pod Priority and Preemption](https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md#how-does-cluster-autoscaler-work-with-pod-priority-and-preemption).
* **Don't combine other node autoscaling mechanisms**, such as Virtual Machine Scale Set autoscalers, with the cluster autoscaler.
* The cluster autoscaler **might be unable to scale down if pods can't move, such as in the following situations**:
  + A directly created pod not backed by a controller object, such as a Deployment or ReplicaSet.
  + A pod disruption budget (PDB) that's too restrictive and doesn't allow the number of pods to fall below a certain threshold.

# AKS Cluster with autoscaler Metric

The metrics used by the AKS (Azure Kubernetes Service) cluster autoscaler to make scaling decisions are primarily based on the resource utilization of the nodes in the cluster. The cluster autoscaler monitors these metrics and adjusts the number of nodes in the cluster accordingly to maintain adequate resource availability.

The key metrics considered by the AKS cluster autoscaler include:

1. **Node CPU Utilization**: This metric measures the CPU usage of the nodes in the cluster. If the average CPU utilization across nodes exceeds a certain threshold (typically set to 70-80%), the cluster autoscaler may decide to scale up the cluster by adding more nodes.
2. **Node Memory Utilization**: Similar to CPU utilization, this metric measures the memory usage of the nodes in the cluster. If the average memory utilization across nodes exceeds a certain threshold, the cluster autoscaler may decide to scale up the cluster by adding more nodes.
3. **Pod Scheduling**: The cluster autoscaler also considers pending pod requests and scheduling constraints. If there are pending pods that cannot be scheduled due to resource constraints, the autoscaler may decide to scale up the cluster to accommodate the pending pods.
4. **Custom Metrics**: Additionally, you can configure the cluster autoscaler to scale based on custom metrics specific to your application or workload. For example, you may define custom metrics based on application performance metrics, queue length, or other relevant factors.

The cluster autoscaler continuously monitors these metrics and makes scaling decisions based on predefined rules and thresholds. By scaling the cluster up or down dynamically, the autoscaler helps optimize resource utilization and maintain the desired level of performance and availability for the applications running in the AKS cluster.

# AKS Cluster autoscaler with custom Metric

To configure the AKS (Azure Kubernetes Service) cluster autoscaler to scale based on custom metrics specific to your application or workload, you can leverage custom metrics and horizontal pod autoscaler (HPA) configurations in Kubernetes. Here's how you can achieve this:

1. **Set Up Custom Metrics**:

First, you need to expose custom metrics from your application or monitoring system. This could involve instrumenting your application code to emit custom metrics or using monitoring solutions like Prometheus or Azure Monitor to collect custom metrics.

1. **Configure Horizontal Pod Autoscaler (HPA)**:

Once you have custom metrics available, you can configure an HPA in Kubernetes to scale your application based on these metrics. You define a custom HPA resource with your custom metrics as the scaling target.

For example, let's say you have a custom metric named **custom\_metric**, and you want to scale your deployment based on this metric. You can create an HPA like this:

apiVersion: autoscaling/v2beta2

kind: HorizontalPodAutoscaler

metadata:

  name: myapp-hpa

spec:

  scaleTargetRef:

    apiVersion: apps/v1

    kind: Deployment

    name: myapp-deployment

  minReplicas: 1

  maxReplicas: 10

  metrics:

  - type: External

    external:

      metricName: custom\_metric

      targetAverageValue: 1000m

1. In this example, the HPA scales the deployment named **myapp-deployment** based on the custom metric **custom\_metric**. The **targetAverageValue** specifies the desired value for the custom metric at which the deployment should be scaled.
2. **Enable Custom Metrics in AKS Cluster Autoscaler**:

By default, the AKS cluster autoscaler supports scaling based on CPU and memory metrics. To enable scaling based on custom metrics, you need to deploy and configure a custom metrics adapter that translates custom metrics into a format understandable by the HPA and the cluster autoscaler.

You can use the Kubernetes Metrics Server with a custom metrics adapter, such as the Prometheus Adapter or Azure Monitor Metrics Adapter, to expose custom metrics to the HPA and the cluster autoscaler.

1. **Deploy Custom Metrics Adapter**:

Deploy the custom metrics adapter in your AKS cluster. You can find instructions for deploying specific custom metrics adapters in the respective documentation.

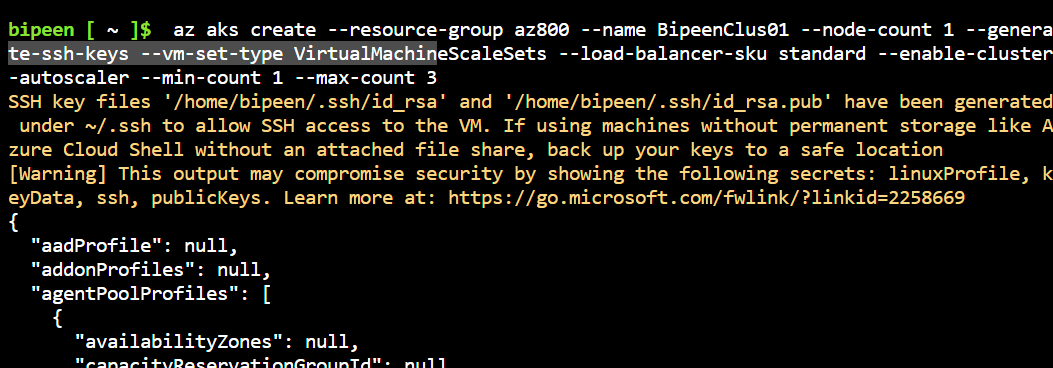
1. **Update AKS Node Pool Configuration**:

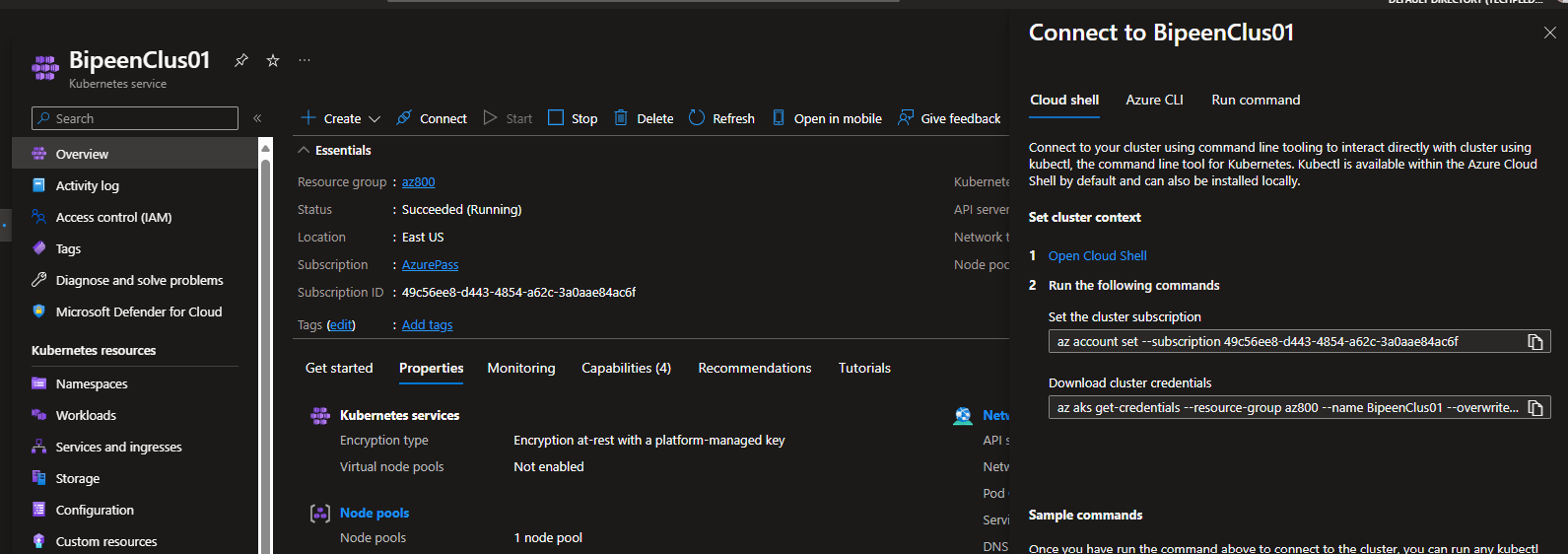
Ensure that the AKS node pool associated with your application has autoscaling enabled. You can configure the minimum and maximum number of nodes in the node pool to scale dynamically based on the workload demands.

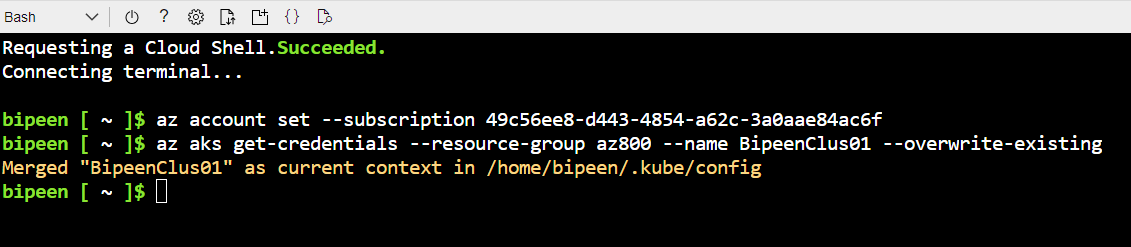
By following these steps, you can configure the AKS cluster autoscaler to scale based on custom metrics specific to your application or workload. Adjust the HPA configuration and custom metrics adapter based on your requirements and monitoring setup.

# Create AKS Cluster with autoscaler and Connect

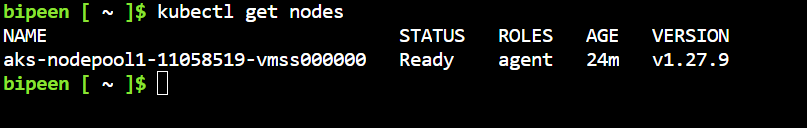
az aks create --resource-group az800 --name BipeenClus01 --node-count 1 --generate-ssh-keys --vm-set-type VirtualMachineScaleSets --load-balancer-sku standard --enable-cluster-autoscaler --min-count 1 --max-count 3





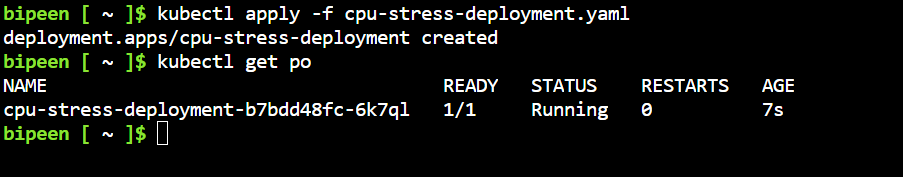


kubectl get nodes

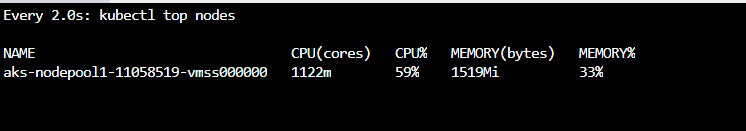


Clone the Code from : <https://github.com/bipeensinha/AKS-Scaleing>

kubectl get kubectl apply -f cpu-stress-deployment.yaml





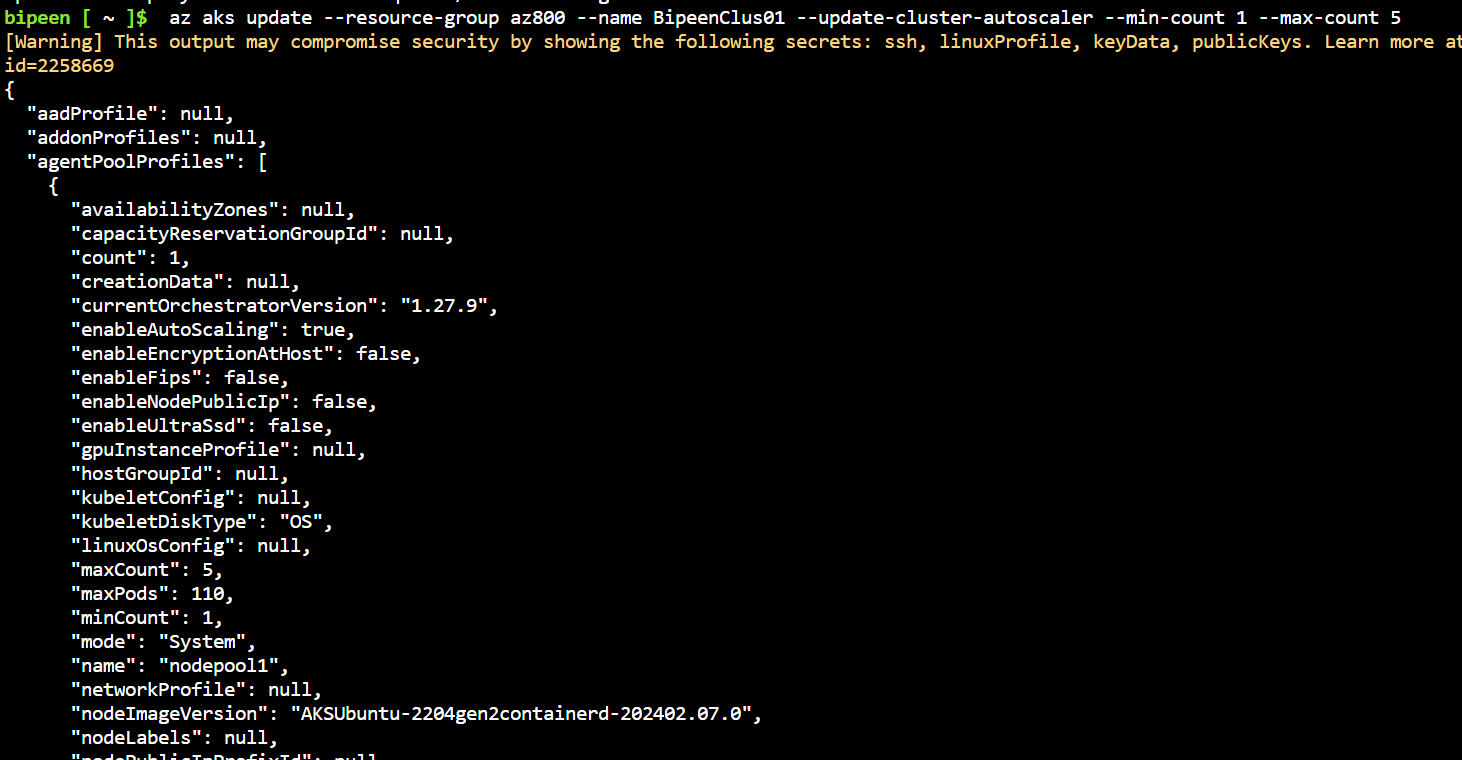


# Update the cluster autoscaler settings

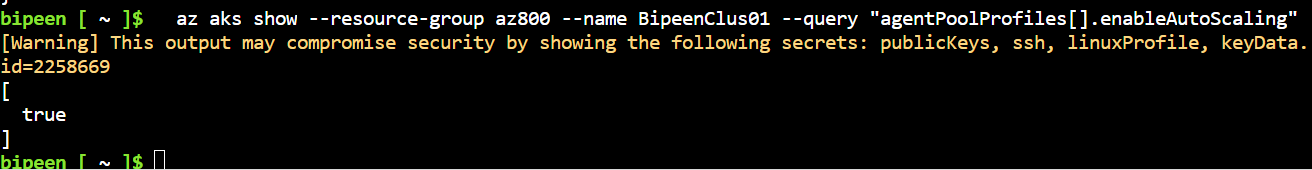
As your application demands change, you might need to adjust the cluster autoscaler node count to scale efficiently.

* Change the node count using the [az aks update](https://learn.microsoft.com/en-us/cli/azure/aks" \l "az-aks-update) command and update the cluster autoscaler using the --update-cluster-autoscaler parameter and specifying your updated node --min-count and --max-count.

az aks update --resource-group az800 --name BipeenClus01 --update-cluster-autoscaler --min-count 1 --max-count 5

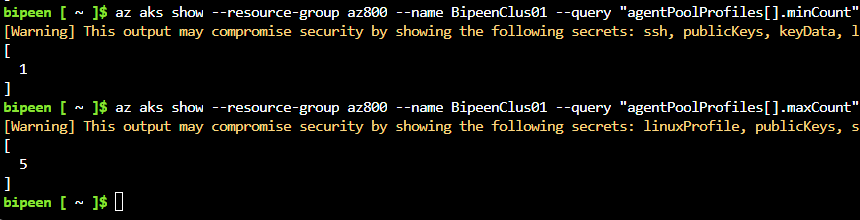


az aks show --resource-group az800 --name BipeenClus01 --query "agentPoolProfiles[].enableAutoScaling"



az aks show --resource-group az800 --name BipeenClus01 --query "agentPoolProfiles[].minCount"

az aks show --resource-group az800 --name BipeenClus01 --query "agentPoolProfiles[].maxCount"



# Disable the cluster autoscaler on a cluster

* Disable the cluster autoscaler using the [az aks update](https://github.com/Azure/azure-cli-extensions/tree/master/src/aks-preview) command and the --disable-cluster-autoscaler parameter.
* az aks update --resource-group az800 --name BipeenClus01 --disable-cluster-autoscaler



