**Autoscaling**

**Horizontal Pod Autoscaler (HPA) vs. Vertical Pod Autoscaler (VPA)**

Both HPA and VPA are used to scale workloads in Kubernetes, but they serve different purposes.

A screenshot of a computer

AI-generated content may be incorrect.

**📌 Real-Time Use Case for HPA**

**Scenario: Auto-scaling an Nginx Deployment Based on CPU Usage**

Imagine you have an **Nginx-based web application** running in Kubernetes.

* When traffic increases, CPU usage spikes.
* To handle the load, **HPA increases the number of pods** dynamically.
* When the load decreases, it **removes pods** to optimize resource usage.

kubectl apply -f <https://github.com/kubernetes-sigs/metrics-server/releases/latest/download/components.yaml>

kubectl get deployment -n kube-system | grep metrics-server

**Horizontal Scaling**

**HorizontalPod – Autoscaler**

kubectl apply -f deployment.yaml

kubectl expose deployment cpu-demo --port=80 --type=LoadBalancer

kubectl apply -f hpa.yaml

kubectl get hpa

kubectl get pods

**Generate Load to Trigger Scaling (for test) - Use a temporary pod to stress the service:**

kubectl run -i --tty load-generator --image=busybox /bin/sh

while true; do wget -q -O- http://cpu-demo.default.svc.cluster.local; done

**Horizontal Scaling**

---

apiVersion: "apps/v1"

kind: "Deployment"

metadata:

name: "buggycpu"

namespace: "default"

labels:

app: "buggycpu"

spec:

replicas: 1

selector:

matchLabels:

app: "buggycpu"

template:

metadata:

labels:

app: "buggycpu"

spec:

containers:

- name: buggycpu

image: sumanth17121988/buggyapp:v1

resources:

limits:

cpu: "1"

memory: "1024Mi"

requests:

cpu: "500m"

memory: "512Mi"

---

apiVersion: "autoscaling/v2"

kind: "HorizontalPodAutoscaler"

metadata:

name: "buggycpu-hpa"

namespace: "default"

labels:

app: "buggycpu"

spec:

scaleTargetRef:

kind: "Deployment"

name: "buggycpu"

apiVersion: "apps/v1"

minReplicas: 1

maxReplicas: 5

metrics:

- type: "Resource"

resource:

name: "cpu"

target:

type: "Utilization"

averageUtilization: 80

The default HPA (autoscaling/v2) behavior is **to average across all containers**, so the pod may not scale even if one container is overloaded.

You're working on a **multi-container pod** where only one container (buggycpu) is consuming high CPU

You're solving this by:

* Using **ContainerResource type metric** (introduced in autoscaling/v2) to target **CPU usage of a specific container** (buggycpu).
* This allows HPA to **scale based on individual container utilization**, not the whole pod.

**✅ What We’ll Do**

1. Create a **multi-container pod**:
   * buggycpu container (simulates high CPU)
   * idle container (almost no CPU usage)
2. Set **resource requests/limits** for each container.
3. Create **HPA with ContainerResource metric**, target buggycpu.
4. Validate that autoscaling triggers **only based on buggycpu** container’s CPU

**🧭 What is VPA?**

Vertical Pod Autoscaler automatically adjusts the CPU and memory requests/limits of your pods based on usage over time.

* Ideal for long-running pods
* Improves resource efficiency
* Not meant to be used together with HPA (if HPA is scaling based on CPU/Memory)

**🔧 Step-by-Step: Set Up Vertical Pod Autoscaler (VPA) in AKS**

kubectl apply -f <https://github.com/kubernetes/autoscaler/releases/latest/download/vertical-pod-autoscaler.yaml>

**🎯 Vertical Pod Autoscaler (VPA) — Works at Pod Level, Not Node Level**

**✅ VPA is Pod-Level Autoscaling**

* It adjusts the **CPU and memory requests/limits of individual pods**.
* It helps right-size **container resource requests** based on historical usage.
* It does **not** add or remove pods or nodes.

Example behavior:

* If your pod consistently uses more CPU than it requested, VPA might increase the CPU request from 100m to 300m.
* This improves **scheduling efficiency** and avoids **OOMKilled** or **throttling**.

**Deploy the Vertical Pod Autoscaler on a new cluster**

az aks create --name <cluster-name> --resource-group <resource-group-name> --enable-vpa --generate-ssh-keys

**Update an existing cluster to use the Vertical Pod Autoscaler**

az aks update --resource-group internal-training --name aksdemo --enable-vpa

**Disable the Vertical Pod Autoscaler on an existing cluster**

az aks update --name <cluster-name> --resource-group <resource-group-name> --disable-vpa

<https://learn.microsoft.com/en-us/azure/aks/use-vertical-pod-autoscaler>

**🔄 1. Auto Mode**

**✅ Description:**

* **Automatically updates** pod CPU and memory by evicting & restarting pods with **new recommendations.**

**🔥 Real-time Use:**

* Good for **background services**, not recommended for **latency-sensitive** or **stateful** apps.

apiVersion: autoscaling.k8s.io/v1

kind: VerticalPodAutoscaler

metadata:

name: nginx-vpa

spec:

targetRef:

apiVersion: "apps/v1"

kind: Deployment

name: nginx

updatePolicy:

updateMode: "Auto"

**🚨 Caution:**

* Pod will be **evicted and restarted** based on thresholds.
* Disruptive if not combined with **PodDisruptionBudget**.

==

apiVersion: "autoscaling.k8s.io/v1"

kind: VerticalPodAutoscaler

metadata:

name: hamster-vpa

spec:

targetRef:

apiVersion: "apps/v1"

kind: Deployment

name: hamster

resourcePolicy:

containerPolicies:

- containerName: '\*'

minAllowed:

cpu: 100m

memory: 50Mi

maxAllowed:

cpu: 1

memory: 500Mi

controlledResources: ["cpu", "memory"]

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: hamster

spec:

selector:

matchLabels:

app: hamster

replicas: 2

template:

metadata:

labels:

app: hamster

spec:

securityContext:

runAsNonRoot: true

runAsUser: 65534

containers:

- name: hamster

image: registry.k8s.io/ubuntu-slim:0.1

resources:

requests:

cpu: 100m

memory: 50Mi

command: ["/bin/sh"]

args:

- "-c"

- "while true; do timeout 0.5s yes >/dev/null; sleep 0.5s; done"

kubectl apply -f hamster.yaml

kubectl get pods -l app=hamster

kubectl describe pod hamster-<example-pod>

kubectl get --watch pods -l app=hamster

kubectl describe pod hamster-<example-pod>

kubectl describe vpa/hamster-vpa

**Kubernetes v1.33: New features in DRA**

**https://kubernetes.io/docs/concepts/scheduling-eviction/dynamic-resource-allocation/**

Kubernetes [Dynamic Resource Allocation](https://kubernetes.io/docs/concepts/scheduling-eviction/dynamic-resource-allocation/) (DRA) was originally introduced as an alpha feature in the v1.26 release, and then went through a significant redesign for Kubernetes v1.31.

**What is In-Place Pod Resize?**

Before Kubernetes 1.33, changing a pod’s CPU or memory resources meant updating the pod’s specification, which triggered a pod recreation. This process often led to downtime, especially for stateful applications like databases or message queues, where restarts could disrupt connections or lose in-memory state.

**Key Benefits**

* **No Downtime:** Adjust resources for stateful workloads without interrupting running processes (in many cases).
* **Better Resource Utilization:** Scale down over-provisioned pods or scale up to handle bursts without recreating pods.
* **Faster Scaling:** In-place updates are quicker than pod recreation, improving responsiveness

**CLUSTER – AUTOSCALER**

**🚀 Cluster Autoscaler — Overview**

The **Cluster Autoscaler (CA)** automatically adjusts the number of **nodes** in a Kubernetes cluster based on resource demand.

**⚙️ How It Works**

**🔼 Scale-Up**

* Triggered when **pods are pending** due to insufficient CPU/memory.
* CA evaluates every **10 seconds** by default (can be changed).
* If no nodes can accommodate pending pods → **new nodes are added** to the node pool.

**🔽 Scale-Down**

* Nodes are **removed** if underutilized (e.g., <50% CPU & memory usage).

Great! You're using a CPU-intensive app (buggyapp:v1) and an HPA targeting 80% CPU utilization — perfect for testing **HPA-based burst-to-ACI with nodeAffinity and taints**.

Now, let's **modify your YAML** to:

1. 🎯 Prefer AKS nodes first
2. ✅ Tolerate ACI (virtual-node-aci-linux) for overflow pods
3. 🧠 Use soft affinity (preferred, not required)

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: buggycpu

namespace: default

labels:

app: buggycpu

spec:

replicas: 1

selector:

matchLabels:

app: buggycpu

template:

metadata:

labels:

app: buggycpu

spec:

imagePullSecrets:

- name: acr-secret

affinity:

nodeAffinity:

preferredDuringSchedulingIgnoredDuringExecution:

- weight: 100

preference:

matchExpressions:

- key: node-type

operator: In

values:

- aks

tolerations:

- key: "virtual-kubelet.io/provider"

operator: "Equal"

value: "azure"

effect: "NoSchedule"

containers:

- name: buggycpu

image: skrisacr.azurecr.io/buggyapp:v1

resources:

limits:

cpu: "400m"

memory: "256Mi"

requests:

cpu: "400m"

memory: "256Mi"

---

apiVersion: autoscaling/v2

kind: HorizontalPodAutoscaler

metadata:

name: buggycpu-hpa

namespace: default

labels:

app: buggycpu

spec:

scaleTargetRef:

kind: Deployment

name: buggycpu

apiVersion: apps/v1

minReplicas: 1

maxReplicas: 10

metrics:

- type: Resource

resource:

name: cpu

target:

type: Utilization

averageUtilization: 80

kubectl get hpa

kubectl get pods -o wide -w

**🧠 Why This Happens**

Azure Container Instances (ACI), used by virtual nodes in AKS, has **stricter requirements** than standard Kubernetes:

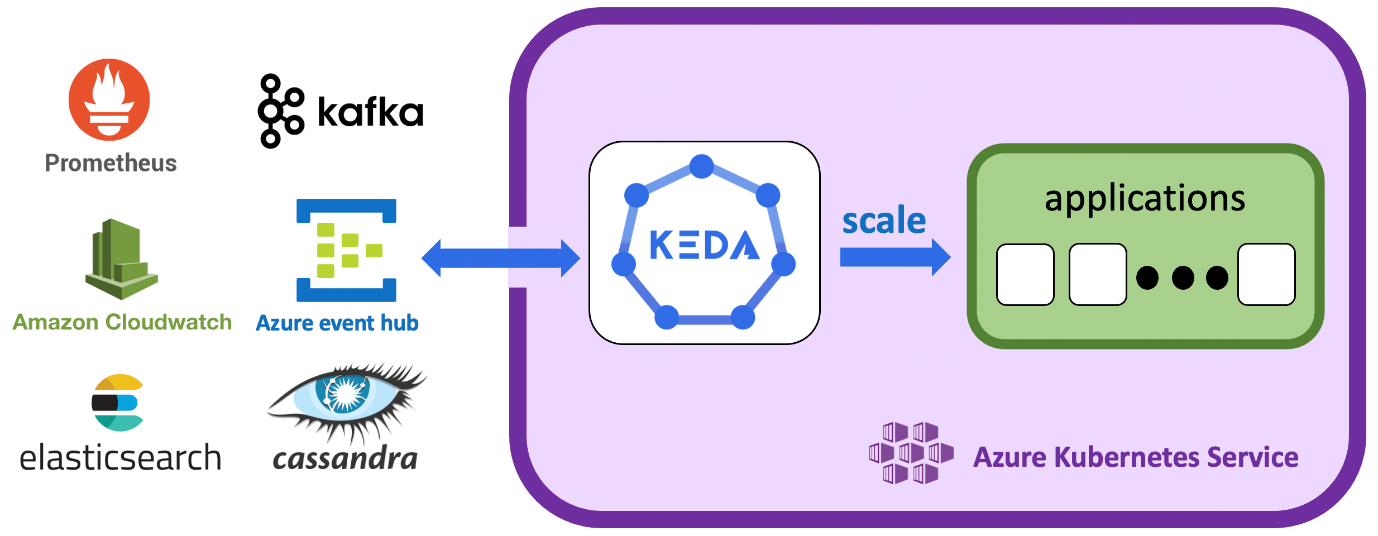
⚠️ In ACI, **CPU limits must be less than or equal to requests**.

✅ Option 1: Make Limits = Requests (Safe for ACI)

✅ Option 2: Remove Limits (Use Only Requests)

===KEDA===

**🚀 Why Autoscaling Based on CPU and Memory is Not Enough in Modern Microservices and How KEDA Solves It**



In the dynamic world of microservices, traditional autoscaling metrics like CPU and memory utilization often fall short. Here's why:

**The Real Problem with CPU/Memory-Based Autoscaling**

1. Irrelevant Metrics: CPU and memory are not always the best indicators of application load. In many cases, traffic patterns, such as queue length, request rate, or event volume, are the real drivers of resource needs.
2. Delayed Scaling: Autoscaling based on CPU and memory can lag behind actual demand, leading to degraded user experiences during traffic spikes.
3. Inefficiency: Services that depend on external systems like message queues (Kafka, RabbitMQ) or APIs often experience bottlenecks unrelated to CPU/memory, making these metrics unreliable for scaling decisions.

**Why We Need Autoscaling Based on Traffic**

Autoscaling based on traffic metrics like queue length, HTTP requests per second, or Kafka lag ensures:

* Proactive Scaling: Resources scale up or down based on real demand, not just system pressure.
* Improved Reliability: Services handle traffic spikes gracefully without over-allocating resources.
* Cost Efficiency: Resources are allocated precisely where and when they are needed.

This is where KEDA (Kubernetes Event-driven Autoscaler) comes into play.

**What is KEDA?**

KEDA is an open-source Kubernetes-based tool that enables event-driven scaling. It supports scaling workloads based on custom metrics, such as message queue lag, HTTP request counts, or database queries, by integrating directly with event sources.

**2. Install KEDA**

**Install KEDA using Helm:**

**helm repo add kedacore https://kedacore.github.io/charts**

**helm repo update**

**helm install keda kedacore/keda --namespace keda --create-namespace**

**kubectl get pods -n keda**

**How KEDA Improved Reliability and Scalability**

1. **Dynamic Scaling**: Pods scaled up during traffic spikes and scaled down during idle times, ensuring no resource wastage.
2. **Proactive Load Handling**: By scaling based on Kafka lag, we eliminated processing delays, improving the user experience.
3. **Cost Efficiency**: Optimized resource usage saved costs compared to over-provisioning resources based on CPU/memory.
4. **Better Reliability**: No more application crashes or degraded performance during peak loads.

Scaling based on CPU and memory is outdated in today's fast-paced, event-driven microservices world. KEDA enables you to focus on what truly matters: **scaling based on real demand.**

If you're struggling with inefficient autoscaling strategies, consider exploring KEDA. It's a game-changer for ensuring reliability, scalability, and cost-effectiveness.

Have questions or need help implementing KEDA? Let's connect and discuss!

**✅ Step 1: Install Prometheus with ServiceMonitor Support**

**1.1 Create Namespace**

kubectl create namespace monitoring

helm repo add prometheus-community https://prometheus-community.github.io/helm-charts

helm repo update

helm install prom-stack prometheus-community/kube-prometheus-stack \

--namespace monitoring \

--set grafana.enabled=false

kubectl get pods -n monitoring

**🧭 Step 2: Deploy Spring Boot App with Prometheus Annotations**

<https://github.com/Sumanth17-git/SRETraining/tree/main/springboot-microservice-logging>

**Update your Spring Boot deployment (springboot-app) and expose /actuator/prometheus.**

apiVersion: apps/v1

kind: Deployment

metadata:

name: springboot-app

labels:

app: springboot-app

spec:

replicas: 2 # Updated replicas for high availability

selector:

matchLabels:

app: springboot-app

template:

metadata:

labels:

app: springboot-app

annotations:

prometheus.io/scrape: "true"

prometheus.io/port: "8881" # Updated port

prometheus.io/path: "/actuator/prometheus"

spec:

containers:

- name: springboot-app

image: sumanth17121988/springbootmetric:1

ports:

- name: metrics-port # Named port for Prometheus

containerPort: 8881 # Application container port

resources:

limits:

cpu: "500m"

memory: "256Mi"

requests:

cpu: "250m"

memory: "128Mi"

livenessProbe:

httpGet:

path: /actuator/health/liveness

port: 8881

initialDelaySeconds: 10

periodSeconds: 10

readinessProbe:

httpGet:

path: /actuator/health/readiness

port: 8881

initialDelaySeconds: 5

periodSeconds: 5

---

apiVersion: v1

kind: Service

metadata:

name: springboot-app-service

labels:

prometheus: monitored # Label to match ServiceMonitor

spec:

selector:

app: springboot-app

ports:

- name: metrics-port # Named port for Prometheus

protocol: TCP

port: 8881 # Exposed service port

targetPort: 8881 # Container port

type: LoadBalancer # Expose the service externally

---

apiVersion: monitoring.coreos.com/v1

kind: ServiceMonitor

metadata:

name: springboot-app-monitor

labels:

release: prom-stack # Label to match the Prometheus release name

spec:

selector:

matchLabels:

prometheus: monitored # Matches the label on the Service

namespaceSelector:

matchNames:

- default # Ensure this matches the namespace

endpoints:

- port: "metrics-port" # Named port as a string

path: /actuator/prometheus # Path to scrape metrics

interval: 30s # Scraping interval

**kubectl apply -f deployment\_prometheus.yaml**

**http://<ipaddress>:8881/actuator/prometheus**

**📡 Step 3: Validate Metrics in Prometheus**

1. **Get Prometheus URL:**

**kubectl get svc -n monitoring prom-stack-kube-prometheus-prometheus**

**kubectl edit svc -n monitoring prom-stack-kube-prometheus-prometheus**

**change it to LoadBalancer.**

**A screenshot of a computer

AI-generated content may be incorrect.**

**⚙️ Step 4: Install KEDA**

**helm repo add kedacore https://kedacore.github.io/charts**

**helm repo update**

**kubectl create ns keda**

**helm install keda kedacore/keda --namespace keda**

**kubectl get all -n keda**

**vi scaledobject.yaml**

apiVersion: keda.sh/v1alpha1

kind: ScaledObject

metadata:

name: springboot-keda-scaler

namespace: default

spec:

scaleTargetRef:

name: springboot-app

minReplicaCount: 1

maxReplicaCount: 5

pollingInterval: 30 # Poll every 30s

cooldownPeriod: 60 # Wait 60s before scaling down

triggers:

- type: prometheus

metadata:

serverAddress: http://prom-stack-kube-prometheus-prometheus.monitoring.svc.cluster.local:9090

metricName: api\_json\_requests\_count\_total

query: |

sum(rate(api\_json\_requests\_count\_total{application="SpringbootMicroservice"}[1m]))

threshold: "1"

**kubectl apply -f keda-scaler.yaml**

**kubectl get hpa**

**Step 6: Generate Load with k6**

**🧰 Step 1: Install k6 (if not already)**

**Ubuntu/Debian:**

sudo apt update

sudo apt install -y gnupg software-properties-common

curl -s https://dl.k6.io/key.gpg | sudo apt-key add -

echo "deb https://dl.k6.io/deb stable main" | sudo tee /etc/apt/sources.list.d/k6.list

sudo apt update

sudo apt install k6

**script.js**

import http from 'k6/http';

import { sleep } from 'k6';

export let options = {

vus: 10, // 10 virtual users

duration: '5m', // Run for 5 minutes

};

export default function () {

http.get('http://20.253.196.6:8881/api/json');

sleep(1); // Wait 1 second between requests

}

k6 run script.js

kubectl get hpa

kubectl get pods -l app=springboot-app -o wide

kubectl get pods -w

kubectl logs -n keda -l app=keda-operator -f

kubectl get hpa -w