**Scaling Applications in Azure Kubernetes environment**

* **Vertical pod Autoscaling**
* **Horizontal Pod Autoscaling**
* **Keda**

**Vertical Pod Autoscaling (preview) in Azure Kubernetes Service (AKS)**

* When configured, it automatically sets resource requests and limits on containers per workload based on past usage. VPA makes certain pods are scheduled onto nodes that have the required CPU and memory resources.
* It analyzes and adjusts processor and memory resources to right size your applications. VPA isn't only responsible for scaling up, but also for scaling down based on their resource use over time.
* A Pod is evicted if it needs to change its resource requests if its scaling mode is set to AUTO or RECREATE
* Set CPU and memory constraints for individual containers by specifying a resource policy.
* Improve cluster resource utilization and frees up CPU and memory for other pods.
* VPA is a feature that allows you to auto adjust the CPU and memory resources allocated to your pods based on their actual usage.
* unlike the horizontal pod scaling which scales the number of replicas of a pod.
* Resources requests are minimum amount of CPU and memory that a container needs to function.
* while resource limits are the maximum amount of CPU and memory that a container needs.
* The VPA object continuously monitors the resource usage of containers in a pod and adjusts their resource requests and limits based on that usage.

**List of Contents :**

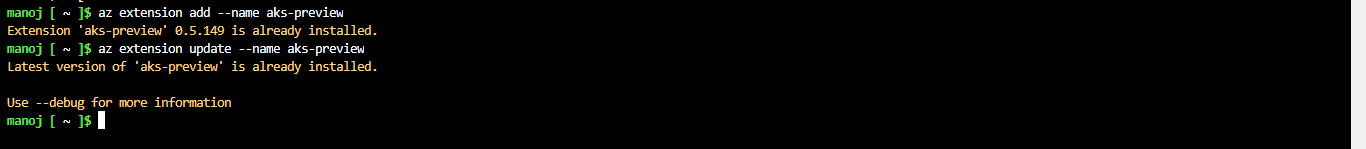
1. Install the aks-preview extension and update to the latest version of the extension released.
2. Register the AKS-VPApreview feature flag by using the az feature register command.
3. Deploy, upgrade, or disable the Vertical Pod Autoscaler on your cluster.
   1. enable VPA on a new cluster.
   2. Optionally, to enable VPA on an existing cluster.
4. verify that the Vertical Pod Autoscaler pods have been created successfully.

## Test your Vertical Pod Autoscaler installation.

* 1. Create a file named hamster.yaml and copy in the following manifest of the Vertical Pod.
  2. Deploy the hamster.yaml Vertical Pod Autoscaler example using the kubectl apply command.

1. Set Pod Autoscaler requests automatically.
   1. Enable VPA for your cluster.
   2. Create a file named azure-autodeploy.yaml.
   3. Create the pod with the kubectl create.
   4. Create a file named azure-vpa-auto.yaml.
   5. Apply the manifest to the cluster using the kubectl apply command.
   6. To get detailed information about the Vertical Pod Autoscaler and its recommendations for CPU and memory, use the kubectl get command.
2. Limitations of VPA.
3. **Install the aks-preview extension and update to the latest version of the extension released.**

* az extension add --name aks-preview
* az extension update --name aks-preview



1. Register the AKS-VPApreview feature flag by using the az feature register command.

* az feature register --namespace "Microsoft.ContainerService" --name "AKS-VPAPreview".
* az feature show --namespace "Microsoft.ContainerService" --name "AKS-VPAPreview".
* az provider register --namespace Microsoft.ContainerService

A black screen with white text

Description automatically generated

1. **Deploy, upgrade, or disable the Vertical Pod Autoscaler on your cluster.**

* az group create -g brilliorg1 -l eastus

A screen shot of a computer

Description automatically generated

* az aks create -n brilliocluster1 -g brilliorg1 --enable-vpa
* Optionally, to enable VPA on an existing cluster, use the --enable-vpa

A screen shot of a computer

Description automatically generated

1. **verify that the Vertical Pod Autoscaler pods have been created successfully.**

* az aks get-credentials --name brilliocluster1 --resource-group brilliorg1
* kubectl get pods -n kube-system

A screenshot of a computer

Description automatically generated

## **Test your Vertical Pod Autoscaler installation.**

* vi hamster.yaml

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 # nobody

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

A screen shot of a computer

Description automatically generated

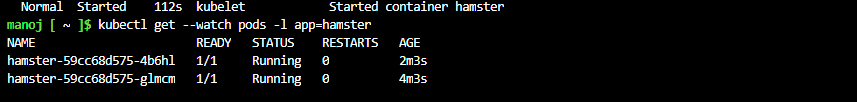
* Use the kubectl describe command on one of the pods to view its CPU and memory reservation.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated



* The pod has 100 millicpu and 50 Mibibytes of memory reserved in this example. For this sample application, the pod needs less than 100 millicpu to run, so there's no CPU capacity available. The pods also reserves much less memory than needed. The Vertical Pod Autoscaler vpa-recommender deployment analyzes the pods hosting the hamster application to see if the CPU and memory requirements are appropriate. If adjustments are needed, the vpa-updater relaunches the pods with updated values.
* Wait for the vpa-updater to launch a new hamster pod, which should take a few minutes.
* When a new hamster pod is started, describe the pod running the kubectl describe command and view the updated CPU and memory reservations.

A computer screen shot of a black screen

Description automatically generated

A screenshot of a computer

Description automatically generated

* This pod was under-resourced, and the Vertical Pod Autoscaler corrected the estimate with a much more appropriate value.
* To view updated recommendations from VPA, run the kubectl describe command to describe the hamster-vpa resource information.

A black background with white text

Description automatically generated

A computer screen with white text

Description automatically generated

1. **Set Pod Autoscaler requests automatically**

* az aks update -n brilliocluster1 -g brilliorg1 --enable-vpa

A screen shot of a computer

Description automatically generated

* vi azure-autodeploy.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: vpa-auto-deployment

spec:

replicas: 2

selector:

matchLabels:

app: vpa-auto-deployment

template:

metadata:

labels:

app: vpa-auto-deployment

spec:

containers:

- name: mycontainer

image: mcr.microsoft.com/oss/nginx/nginx:1.15.5-alpine

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 create -f azure-autodeploy.yaml

This manifest describes a deployment that has two Pods. Each Pod has one container that requests 100 milliCPU and 50 MiB of memory.

* kubectl get pods

A black background with a black square

Description automatically generated

Create a file named azure-vpa-auto.yaml, and copy in the following manifest that describes a VerticalPodAutoscaler.

* vi azure-vpa-auto.yaml

apiVersion: autoscaling.k8s.io/v1

kind: VerticalPodAutoscaler

metadata:

name: vpa-auto

spec:

targetRef:

apiVersion: "apps/v1"

kind: Deployment

name: vpa-auto-deployment

updatePolicy:

updateMode: "Auto"

* kubectl create -f azure-vpa-auto.yaml

A screen shot of a computer

Description automatically generated

Get detailed information about one of your running Pods by using the kubectl get command

* kubectl get pod hamster-59cc68d575-4b6hl --output yaml.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

To get detailed information about the Vertical Pod Autoscaler and its recommendations for CPU and memory, use the kubectl get command.

* kubectl get vpa vpa-auto --output yaml

A screenshot of a computer program

Description automatically generated

A black background with a black square

Description automatically generated

* The results show the target attribute specifies that for the container to run optimally, it doesn't need to change the CPU or the memory target. Your results may vary where the target CPU and memory recommendation are higher.
* The Vertical Pod Autoscaler uses the lowerBound and upperBound attributes to decide whether to delete a Pod and replace it with a new Pod. If a Pod has requests less than the lower bound or greater than the upper bound, the Vertical Pod Autoscaler deletes the Pod and replaces it with a Pod that meets the target attribute.

1. **Limitations of VPA**.

7.1VPA is not aware of Kubernetes cluster infrastructure variables such as node size in terms of memory and CPU. Therefore, it doesn't know whether a recommended pod size will fit your node. This means that the resource requests recommendation may be too large to fit any node, and therefore pods may go to a pending state because the resource request can’t be met. Some cloud providers such as GKE provide a [cluster autoscaler](https://cloud.google.com/kubernetes-engine/docs/concepts/cluster-autoscaler) to spin up more worker nodes addressing pod pending issues, but if the Kubernetes environment has no cluster autoscaler feature, then pods will remain pending, causing downtime.

7.2VPA does not support StatefulSets yet. The problem is scaling pods in StatefulSet is not simple. Neither starting nor restarting can be done the way it’s done for a Deployment or ReplicaSet. Instead, the pods in StatefulSet are managed in a well-defined order. For example, a Postgres DB StatefulSet will first deploy the master pod and then deploy the slave or replication pods. The master pod can’t be simply replaced with just any other pod.

7.3In Kubernetes, the pod spec is immutable. This means that the pod spec can't be updated in place. To update or change the pod resource request, VPA needs to evict the pod and re-create it. This will disrupt your workload. As a result, running VPA in auto mode isn’t a viable option for many use cases. Instead, it is used for recommendations that can be applied manually during a maintenance window.

7.4VPA won't work with HPA using the same CPU and memory metrics because it would cause a race condition. Suppose HPA and VPA both use CPU and memory metrics for scaling decisions. HPA will try to scale out (horizontally) based on CPU and memory, while at the same time, VPA will try to scale the pods up (vertically). Therefore if you need to use both HPA and VPA together, you must configure HPA to use a custom metric such as web requests.

7.5VPA is not yet ready for JVM-based workloads. This shortcoming is due to its limited visibility into memory usage for Java virtual machine workloads

7.6The performance of VPA is untested on large-scale clusters. Therefore, performance issues may occur when using VPA at scale. This is another reason why it’s not recommended to use VPA within large production environments.

7.7VPA doesn’t consider network and I/O. This is an important issue since ignoring I/O throughout (for writing to disk), and network bandwidth usage can cause application slow-downs and outages.

7.8VPA uses limited historical data. [VPA requires eight days of historical data storage](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md#history-storage) before it’s initiated. The limited use of only eight days of data would miss monthly, quarterly, annual, and seasonal fluctuations that could cause bottlenecks during peak usage.

7.9VPA requires configuration for each cluster. If you manage a dozen or more clusters, you would have to manage separate configurations for each cluster. More sophisticated optimization tools provide a governance workflow for approving and unifying configurations across multiple clusters.

7.10VPA policies lack flexibility. VPA uses a resource policy to control resource computations, and an update policy to control how to apply changes to Pods. The policy functionality is however limited. For example, [the resource policy sets a higher and a lower value](https://github.com/kubernetes/community/blob/master/contributors/design-proposals/autoscaling/vertical-pod-autoscaler.md#resource-policy) calculated based on historical CPU and memory measurements aggregated into percentiles (e.g., 95 percentile) and you can’t choose a more sophisticated [machine learning algorithm](https://www.densify.com/product) to predict usage.

**Horizontal Pod Autoscaling with Cluster Autoscaling**

* Kubernetes uses the horizontal pod autoscaler (HPA) to monitor the resource demand and automatically scale the number of pods. By default, the HPA checks the Metrics API every 15 seconds for any required changes in replica count, and the Metrics API retrieves data from the Kubelet every 60 seconds. So, the HPA is updated every 60 seconds. When changes are required, the number of replicas is increased or decreased accordingly. The HPA works with AKS clusters that have deployed the Metrics Server for Kubernetes 1.8+.
* When you configure the HPA for a given deployment, you define the minimum and maximum number of replicas that can run.
* To minimize race events, a delay value is set. This value defines how long the HPA must wait after a scale event before another scale event can be triggered.
* To respond to changing pod demands, the Kubernetes cluster autoscaler adjusts the number of nodes based on the requested compute resources in the node pool. By default, the cluster autoscaler checks the Metrics API server every 10 seconds for any required changes in node count. If the cluster autoscaler determines that a change is required, the number of nodes in your AKS cluster is increased or decreased accordingly. The cluster autoscaler works with Kubernetes RBAC-enabled AKS clusters that run Kubernetes 1.10.x or higher.

**Table of Contents:**

1. Create a resource group.
2. Create or update existing Kubernetes cluster.
3. Deploy sample application.
4. Enable HPA in the cluster.
5. Generating the load onto the application.
6. Monitoring the functionality of autoscaling.
7. Removing the load generator.
8. **Create a resource group.**

**az group create -g myresourcegroup -l eastus**

1. **Create or update existing Kubernetes cluster.**

Enable the cluster autoscaler on a new cluster.

az aks update \

--resource-group myResourceGroup \

--name myAKSCluster \

--enable-cluster-autoscaler \

--min-count 1 \

--max-count 3

### Enable the cluster autoscaler on an existing cluster.

az aks update \

--resource-group myResourceGroup \

--name myAKSCluster \

--enable-cluster-autoscaler \

--min-count 1 \

### --max-count 3

1. **Deploy sample application.**

Create application file(deploy.yaml).

apiVersion: apps/v1

kind: Deployment

metadata:

name: webapp-deployment

spec:

selector:

matchLabels:

app: webapp

replicas: 2

template:

metadata:

labels:

app: webapp

spec:

containers:

- name: webapp

image: houssemdocker/webapp:111

ports:

- containerPort: 80

resources:

limits:

cpu: 300m

memory: "100Mi"

requests:

cpu: 100m

memory: "50Mi"

---

apiVersion: v1

kind: Service

metadata:

name: webapp-svc

labels:

app: webapp

spec:

ports:

- port: 80

selector:

app: webapp

A black background with white text

Description automatically generated

1. **Enable HPA in the cluster.**

Create hpa manifest file for the application(hpa.yaml).

apiVersion: autoscaling/v2beta2

kind: HorizontalPodAutoscaler

metadata:

name: webapp-hpa

spec:

minReplicas: 3

maxReplicas: 50

scaleTargetRef:

apiVersion: apps/v1

kind: Deployment

name: webapp-deployment

metrics:

- type: Resource

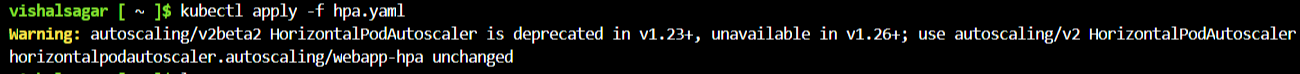
resource:

name: cpu

target:

type: Utilization

averageUtilization: 70



1. **Generating the load onto the application.**

Create the load-generator file (load-generator).

# load-generator-deploy.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: load-generator

spec:

selector:

matchLabels:

run: load-generator

replicas: 100 # 2

template:

metadata:

labels:

run: load-generator

spec:

containers:

- name: load-generator

image: busybox

args: [/bin/sh, -c, 'while true; do wget -q -O- http://webapp-svc; done']



1. **Monitoring the functionality of autoscaling.**

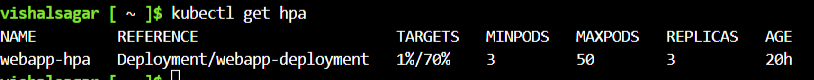
When No-Load on application:

Application is running the minimum replicas.

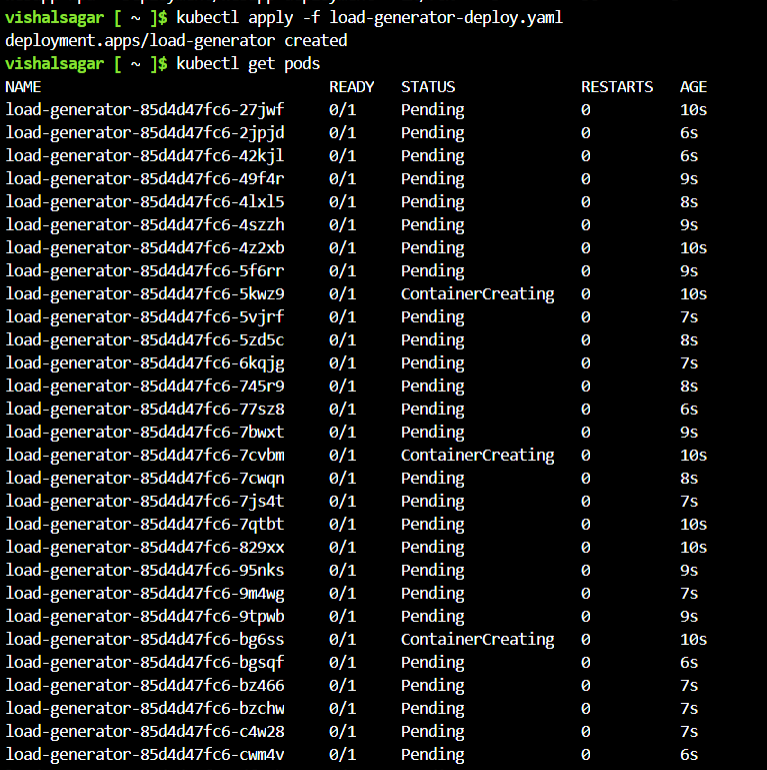
**A screen shot of a computer program

Description automatically generated**

we can the max and min pods below

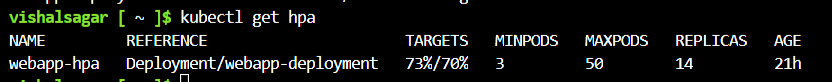
****

When load applied on application:

****

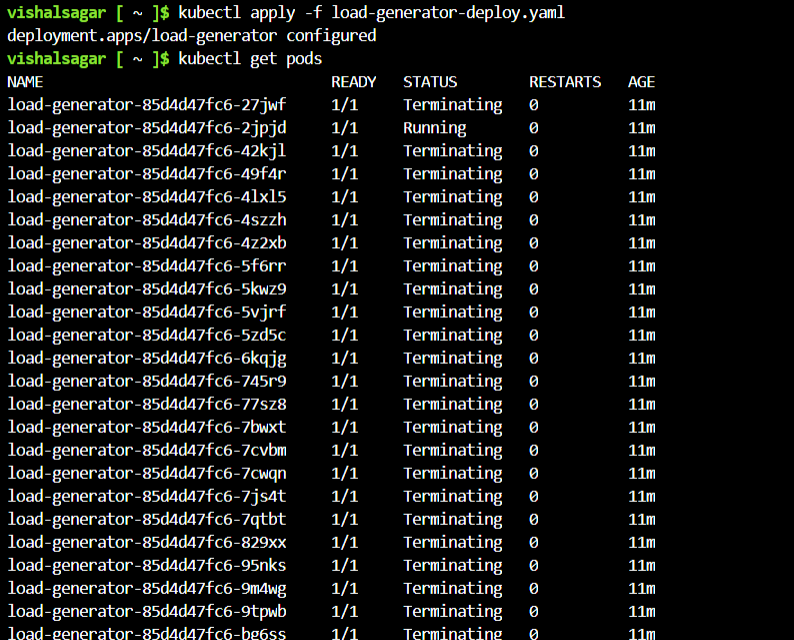
After Appling the load, you can the number of pods is autoscaling gradually.

A black background with white text

Description automatically generated

1. **Removing the load generator.**

When the load removed from the application, you can see the number of pods are terminating and scaling down to minimum number of pods.



After removing the load, you can see the no of pods scaling down.

A black screen with white text

Description automatically generated

Reference document:

[Use the cluster autoscaler in Azure Kubernetes Service (AKS) - Azure Kubernetes Service | Microsoft Learn](https://learn.microsoft.com/en-us/azure/aks/cluster-autoscaler)

[Kubernetes on Azure tutorial - Scale application - Azure Kubernetes Service | Microsoft Learn](https://learn.microsoft.com/en-us/azure/aks/tutorial-kubernetes-scale?tabs=azure-cli)

**Integrate KEDA with AKS cluster.**

**Keda: -**

KEDA is a Kubernetes-based Event Driven Autoscaler. KEDA lets you drive the scaling of any container in Kubernetes based on the load to be processed, by querying metrics from systems such as Prometheus. Integrate KEDA with your Azure Kubernetes Service (AKS) cluster to scale your workloads based on Prometheus metrics.

KEDA has a wide range of [scalers](https://keda.sh/scalers) that can both detect if a deployment should be activated or deactivated, and feed custom metrics for a specific event source. For example, Prometheus, kafka, Postgres and many more.

**Architecture**

The diagram below shows how KEDA works in conjunction with the Kubernetes Horizontal Pod Autoscaler, external event sources, and Kubernetes’ [etcd](https://etcd.io/) data store:

A diagram of a computer network

Description automatically generated

**KEDA performs three key roles within Kubernetes**:

1. **Agent** — KEDA activates and deactivates Kubernetes [Deployments](https://kubernetes.io/docs/concepts/workloads/controllers/deployment) to scale to and from zero on no events. This is one of the primary roles of the keda-operator container that runs when you install KEDA.
2. **Metrics** — KEDA acts as a [Kubernetes metrics server](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/#support-for-custom-metrics) that exposes rich event data like queue length or stream lag to the Horizontal Pod Autoscaler to drive scale out. It is up to the Deployment to consume the events directly from the source. This preserves rich event integration and enables gestures like completing or abandoning queue messages to work out of the box. The metric serving is the primary role of the keda-operator-metrics-apiserver container that runs when you install KEDA.
3. **Admission Webhooks** - Automatically validate resource changes to prevent misconfiguration and enforce best practices by using an [admission controller](https://kubernetes.io/docs/reference/access-authn-authz/admission-controllers/). As an example, it will prevent multiple ScaledObjects to target the same scale target.

List of contents: -

1. Creating the AKS Cluster

2. Deploying the Prometheus and Application in Keda-demo namespace

3. Deploying the Keda in the Keda Namespace

4. Scaling the application using Prometheus metrics trigger

5. Applying load on the Endpoint and checking the Scaled workloads.

1. Creating the AKS Cluster

Kubectl create cluster.

2. Deploying the Prometheus and Application in Keda-demo namespace

Create the namespace:

kubectl create ns keda-demo

Switch to the namespace

kubectl config set-context --current --namespace=keda-demo

Deploying the Prometheus from the below Prometheus manifest file.

prometheus.yaml

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRole

metadata:

name: prometheus

rules:

- apiGroups: [""]

resources:

- services

verbs: ["get", "list", "watch"]

- nonResourceURLs: ["/metrics"]

verbs: ["get"]

---

apiVersion: v1

kind: ServiceAccount

metadata:

name: keda-demo

---

apiVersion: rbac.authorization.k8s.io/v1

kind: ClusterRoleBinding

metadata:

name: prometheus

roleRef:

apiGroup: rbac.authorization.k8s.io

kind: ClusterRole

name: prometheus

subjects:

- kind: ServiceAccount

name: keda-demo

namespace: keda-demo

---

apiVersion: v1

kind: ConfigMap

metadata:

name: prom-conf

labels:

name: prom-conf

data:

prometheus.yml: |-

global:

scrape\_interval: 5s

evaluation\_interval: 5s

scrape\_configs:

- job\_name: 'go-prom-job'

kubernetes\_sd\_configs:

- role: service

relabel\_configs:

- source\_labels: [\_\_meta\_kubernetes\_service\_label\_run]

regex: go-prom-app-service

action: keep

---

apiVersion: apps/v1

kind: Deployment

metadata:

name: prometheus-deployment

spec:

replicas: 1

selector:

matchLabels:

app: prometheus-server

template:

metadata:

labels:

app: prometheus-server

spec:

serviceAccountName: keda-demo

containers:

- name: prometheus

image: prom/prometheus

args:

- "--config.file=/etc/prometheus/prometheus.yml"

- "--storage.tsdb.path=/prometheus/"

ports:

- containerPort: 9090

volumeMounts:

- name: prometheus-config-volume

mountPath: /etc/prometheus/

- name: prometheus-storage-volume

mountPath: /prometheus/

volumes:

- name: prometheus-config-volume

configMap:

defaultMode: 420

name: prom-conf

- name: prometheus-storage-volume

emptyDir: {}

---

apiVersion: v1

kind: Service

metadata:

name: prometheus-service

spec:

ports:

- port: 9090

protocol: TCP

selector:

app: prometheus-server

Apply the Prometheus manifest:

kubectl apply -f prometheus.yaml

Once the pod is up and running, let's see if it also works:

kubectl port-forward svc/prometheus-service 9090

The Prometheus dashboard endpoint shown below: -

A screenshot of a computer

Description automatically generated

Deploy the sample application in Keda-demo namespace:

go-deployment.yaml

apiVersion: apps/v1

kind: Deployment

metadata:

name: go-prom-app

namespace: keda-demo

spec:

selector:

matchLabels:

app: go-prom-app

template:

metadata:

labels:

app: go-prom-app

spec:

containers:

- name: go-prom-app

image: djam97/keda

imagePullPolicy: Always

ports:

- containerPort: 8080

---

apiVersion: v1

kind: Service

metadata:

name: go-prom-app-service

namespace: keda-demo

labels:

run: go-prom-app-service

spec:

ports:

- port: 8080

protocol: TCP

selector:

app: go-prom-app

Deploy sample application:

kubectl apply -f go-deployment.yaml

If the pod is up, verify if it is working.

kubectl port-forward svc/go-prom-app-service 8080

If you visit http://localhost:8080 you should see Hello, you've requested: /.

The application endpoint shown below:

A white rectangular object with a white border

Description automatically generated

3. Deploying the Keda in the Keda Namespace

We can now deploy the KEDA operator. KEDA provides multiple ways to deploy their operator, but for now we will use the k8s manifest.

kubectl apply -f

<https://github.com/kedacore/keda/releases/download/v2.9.0/keda-2.9.0.yaml>.

Now there should be two pods in the namespace KEDA you can check it with the following command:

kubectl get pods -n keda

A screen shot of a computer

Description automatically generated

4. Scaling the application using Prometheus metrics trigger

Now that we have our go application up we have a manifest that will scale our application. Keda offers many triggers that can scale our application, but of course we will use the [Prometheus trigger](https://keda.sh/docs/2.4/scalers/prometheus/).

In a new file called scaled-object.yaml add the following content:

apiVersion: keda.sh/v1alpha1

# Custom CRD provisioned by the Keda operator

kind: ScaledObject

metadata:

name: prometheus-scaledobject

spec:

scaleTargetRef:

# target our deployment

name: go-prom-app

# Interval to when to query Prometheus

pollingInterval: 15

# The period to wait after the last trigger reported active

# before scaling the deployment back to 1

cooldownPeriod: 30

# min replicas keda will scale to

# if you have an app that has an dependency on pubsub

# this would be a good use case to set it to zero

# why keep your app running if your topic has no messages?

minReplicaCount: 1

# max replicas keda will scale to

maxReplicaCount: 20

advanced:

# HPA config

# Read about it here: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/

horizontalPodAutoscalerConfig:

behavior:

scaleDown:

stabilizationWindowSeconds: 30

policies:

- type: Percent

value: 50

periodSeconds: 30

scaleUp:

stabilizationWindowSeconds: 0

policies:

- type: Percent

value: 50

periodSeconds: 10

triggers:

- type: prometheus

metadata:

# address where keda can reach our prometheus on

serverAddress: http://prometheus-service.keda-demo.svc.cluster.local:9090

# metric on what we want to scale

metricName: http\_requests\_total

# if treshold is reached then Keda will scale our deployment

threshold: "100"

query: sum(rate(http\_requests[2m]))

Let's apply the Scaled-object manifest:

kubectl apply -f scaled-object.yaml

Verify

kubectl get --namespace keda-demo pods,hpa,scaledobjects

A black screen with white text

Description automatically generated

We can see there is only one pod running in the cluster where there is no load on the api.

Azure dashboard view.

A screenshot of a computer

Description automatically generated

5. Applying load on the Endpoint and checking the Scaled workloads.

Applying the Load on the end point.

By using the below command, we can generate load on to the cluster.

kubectl run -i --tty load-generator --rm --image=busybox --restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O- http://20.253.3.65:8080/; done"

A black screen with a black background

Description automatically generated

It can take a minute before the application actually starts scaling. After a while you should see the number of pods increased in the cluster. As shown below.

A screen shot of a computer

Description automatically generated

We can see that our prometheus-scaledobject is ready so let’s scale our application! Remember our application scales on the metric http\_requests\_total and our threshold is only 100 so we should be able reach that threshold. For this we can use a simple busybox image to put load on the application.

Azure dashboard view no of replicas increased to 9

A screenshot of a computer

Description automatically generated

Remove the Load on the end point.

Now let’s also look at the scale down process. Stop putting load on the application and let’s just watch the pods.

After removing the load, the number of pods scaled down to original state i.e minimum no of pods to 1.

A black screen with white text

Description automatically generated

Now let’s also look at the scale down process. Stop putting load on the application and let’s just watch the pods. This process should go from 10 -> 5 - > 2 -> 1. This is basically how KEDA goes to work!

Reference:-

[Install the Kubernetes Event-driven Autoscaling (KEDA) add-on by using Azure CLI - Azure Kubernetes Service | Microsoft Learn](https://learn.microsoft.com/en-us/azure/aks/keda-deploy-add-on-cli)

[Using KEDA and Prometheus to scale your k8s workloads | Djamaile Rahamat](https://djamaile.dev/blog/using-keda-and-prometheus/#deploying-keda)

[Deploying KEDA | KEDA](https://keda.sh/docs/2.11/deploy/)