ISSN: 2634 - 8853

Journal of Engineering and Applied Sciences Technology

Review Article

Open Access

Dynamic Scaling of Application using Kubernetes Horizontal Pod Auto Scaling

Pallavi Priya Patharlagadda

USA

ABSTRACT

To make sure that the pods have enough resources to run effectively without going overboard, Pod Autoscaling monitors the pods' usage patterns and modifies their resource requests and restrictions accordingly. This keeps programs operating smoothly while maximizing resource use and reducing expenses. In this paper, we shall look into the various types of autoscaling and the uses of using horizontal autoscaling (HPA).

***Corresponding author**
Pallavi Priya Patharlagadda, USA.

Received: April 03, 2023; **Accepted:** April 10, 2023; **Published:** April 24, 2023

Introduction
Currently, using many container instances in parallel is common for application deployments. The most widely used platform for coordinating and overseeing these container clusters at scale is Kubernetes (K8s). Kubernetes clusters can effectively manage workloads that require a lot of resources by utilizing the pod auto-scaling feature, which is a unique resource definition.

Problem Statement
With the advent of cloud computing, majority of the cloud providers are providing services like IaaS (Infrstructure as a Service), PaaS (Platform as service) and Saas (Software as a service). These services are modeled as Pay as You Go which means that we only pay for what we use. If the application has more traffic, then the application replicas or pods need to be increased to meet the requirements. If the application is having less traffic then the number of application replicas can be reduced. This scaling can be done manually but it would take manual effort and time. Imagine a situation where the traffic spike happen in the middle of the night of early in the morning. We need to have a person monitoring it and then scale it immediately. This would come with some cost. This paper discusses on how we can solve this problem using Kubernetes Auto Scaling feature.

Kubernetes Auto Scaling:
With the help of Kubernetes autoscaling, a cluster can automatically scale up or down in response to demand by modifying pod resources or the number of nodes. Kubernetes can remove nodes or provide fewer resources to a pod as demand declines, and the cluster can add nodes or raise pod resources in response to demand fluctuations. This can enhance performance and optimize resource use and expenses.

Advantages of Pod Auto Scaling:
- **Enhanced performance:** Autoscaling makes sure your application has the resources it needs to manage rising traffic and prevent performance issues by dynamically raising the number of pods based on demand.
- **Cost efficiency:** By using auto-scaling, you only use the resources you require, which can help cut down on the expenses related to over- or under-provisioning resources.
- **High availability:** Auto scaling guarantees that your application will always be accessible, even in the event of unforeseen demand or spikes in traffic.
- **Scalability:** Autoscaling makes it simple and effective to add more resources as your application expands to meet growing demand.

Kubernetes autoscaling can be classified into three types,

Horizontal Pod Autoscaling (HPA): HPA is a Kubernetes feature that allows the number of pod replicas to be automatically scaled up or down in response to predefined metrics.

Vertical Pod Autoscaling (VPA): A Kubernetes feature that helps deploy pods of the proper size and prevents resource use issues on the cluster. Compared to other K8s autoscaling methods, VPA has a stronger connection to capacity planning.

Cluster Autoscaling: a kind of autoscaling generally provided by Kubernetes versions offered by cloud providers. When a pod is waiting to be scheduled or when the cluster has to get smaller to accommodate the existing number of pods, Cluster Autoscaling can add and remove worker nodes dynamically.

Horizontal Pod Auto Scaling (HPA)
Now, let us delve deep into Horizontal Pod Autoscaling (HPA), The HPA (Horizontal Pod Autoscaling) adds and removes pod replicas automatically. This allows for the automatic management of workload scaling when application usage varies.

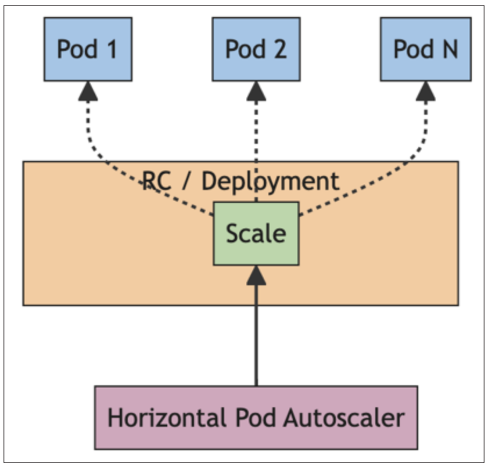
HPA can be helpful for workloads that are stateful as well as stateless. Under the supervision of the Kubernetes controller manager, HPA functions as a control loop. By default, the HPA loop lasts for 15 seconds, but the controller manager can supply a flag to change that value. –horizontal-pod-autoscaler-sync-period is the flag.

The controller manager evaluates real resource use against the metrics specified for every HPA following each loop period. It gets these from the resource metrics API, or the custom metrics API if you tell it to auto-scale based on resources per pod (such as CPU usage).

HPA makes use of metrics to determine auto-scaling, as follows: Resource metrics refers to the typical resource management statistics that the Kubernetes metric server provides, such as memory and CPU use metrics.

Custom metrics refers to request throughput, latency, dependency, queue depth etc. Cluster administrators can install a metrics collector, gather the required application metrics, and expose them to the Kubernetes metrics server with the help of the custom metrics API.

How does a Horizontal Pod Autoscaler work?



According to the goal metric value, the HPA resource updates the deployment resource, as depicted in the diagram. Following that, the number of replica pods operating will either increase or decrease via the pod controller (Deployment).
Thrashing is one issue that can arise in these situations if there is no contingency. When the workload stops reacting to previous autoscaling actions before the HPA completes its next one, this is known as thrashing. The HPA control loop prevents thrashing by selecting the greatest pod count suggestion in the last five minutes.

The code excerpt illustrates the process of setting up an HPA object and a Kubernetes deployment such that the deployment's pods can automatically scale in response to CPU stress. This is demonstrated step-by-step with commentary:

- The kubectl autoscale command
- The HPA YAML resource file

Create a namespace for HPA testing,

```
kubectl create ns hpa-ns
namespace/hpa-ns created
```

Create a deployment for HPA testing

```
cat example-app.yaml
apiVersion: apps/v1
kind: Deployment
metadata:
name: hpa-demo
namespace: hpa-ns
spec:
selector:
matchLabels:
run: hpa-demo
replicas: 1
template:
metadata:
labels:
run: hpa-demo
spec:
containers:
- name: hpa-demo
image: k8s.gcr.io/hpa-example
ports:
- containerPort: 80
resources:
limits:
cpu: 500m
requests:
cpu: 200m
 ---
apiVersion: v1
kind: Service
metadata:
name: hpa-demo
namespace: hpa-ns
labels:
run: hpa-demo
spec:
ports:
- port: 80
selector:
run: hpa-demo

kubectl create -f example-app.yaml
deployment.apps/hpa-demo created
service/hpa-demo created
```

Verify that the pod is operating and that the deployment has been made.

```
kubectl get deploy -n hpa-ns
NAME        READY   UP-TO-DATE   AVAILABLE   AGE
hpa-demo 1/1     1                      1              22s
```

Use the kubectl autoscale command to construct the HPA once the deployment has successfully launched. To keep the total CPU consumption to 50%, this HPA will maintain a minimum of 1 and a maximum of 5 replica pods during the deployment.

```
kubectl -n hpa-ns autoscale deployment hpa-demo --cpu-percent=50 --min=1 --max=5
  horizontalpodautoscaler.autoscaling/hpa-demo autoscaled
```

The following Kubernetes resource would be created using the declarative version of the same command.

```
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
name: hpa-demo
namespace: hpa-ns
spec:
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: hpa-demo
minReplicas: 1
maxReplicas: 5
targetCPUUtilizationPercentage: 50
```

Examine the current state of HPA

```
kubectl -n hpa-ns get hpa
NAME REFERENC TARGETS MINPODS MAXPODS
REPLICAS AGE
 hpa-demo Deployment/hpa-demo 0%/50% 1 5 1
17s
```

Since the operating application is not under any load at the moment, the desired and existing pod counts are equal to 1.

```
kubectl -n hpa-ns get hpa hpa-demo -o yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
name: hpa-demo
namespace: hpa-ns
resourceVersion: "402396524"
selfLink: /apis/autoscaling/v1/namespaces/hpa-ns/
horizontalpodautoscalers/hpa-demo
uid: 6040eea9-0c2b-47de-9725-cfb78f17fe32
spec:
maxReplicas: 5
minReplicas: 1
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: hpa-demo
targetCPUUtilizationPercentage: 50
status:
currentCPUUtilizationPercentage: 0
currentReplicas: 1
desiredReplicas: 1
```

Now run the load test and see the HPA status again

```
kubectl -n hpa-ns run -i --tty load-generator --rm --image=busybox
--restart=Never -- /bin/sh -c "while sleep 0.01; do wget -q -O-
http://hpa-demo; done"
If you don't see a command prompt, try pressing enter.
OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!O
K!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK!OK
!OK!OK!OK!OK!OK!OK!OK!
kubectl -n hpa-ns get hpa
NAME REFERENCE TARGETS MINPODS
MAXPODS REPLICAS AGE
hpa-demo Deployment/hpa-demo 211%/50% 1 5 4
10m
```

Use CTRL-C to halt the load, and then check the HPA status once again to see whether everything has returned to normal and if there is just one replica operating.

```
kubectl -n hpa-ns get hpa

NAME REFERENCE TARGETS MINPODS MAXPODS
REPLICAS AGE
hpa-demo Deployment/hpa-demo 0%/50% 1 5 1
20h

kubectl -n hpa-ns get hpa hpa-demo -o yaml
apiVersion: autoscaling/v1
kind: HorizontalPodAutoscaler
metadata:
name: hpa-demo
namespace: hpa-ns
resourceVersion: "402402364"
selfLink: /apis/autoscaling/v1/namespaces/hpa-ns/
horizontalpodautoscalers/hpa-demo
uid: 6040eea9-0c2b-47de-9725-cfb78f17fe32
spec:
maxReplicas: 5
minReplicas: 1
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: hpa-demo
targetCPUUtilizationPercentage: 50
status:
currentCPUUtilizationPercentage: 0
currentReplicas: 1
desiredReplicas: 1
lastScaleTime: "2023-02-25T10:34:23Z"
```

Clean up the resources

```
kubectl delete ns hpa-ns --cascade
 namespace "hpa-ns" deleted
```

The parameter targetCPUUtilizationPercentage, which assesses the average CPU utilization of the pods, is then the only option available in this API version 1.

API version 2 of autoscaling features a modified syntax and the addition of metrics in place of targetCPUUtilizationPercentage, enabling a more versatile configuration of metrics.

```
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
name: php-apache
spec:
scaleTargetRef:
apiVersion: apps/v1
kind: Deployment
name: hpa-demo
minReplicas: 1
maxReplicas: 10
 metrics:
- type: Resource
resource:
name: cpu
target:
type: Utilization
averageUtilization: 50
```

The CPU utilization metric is a resource metric, since it is represented as a percentage of a resource specified on pod containers. Memory is the other resource metric that can be provided.

By utilizing a target, you may also give resource metrics as direct values rather than as percentages of the intended value by using target.type of AverageValue instead of Utilization, and setting the corresponding target. averageValue instead of target. averageUtilization. Below is an example.

```
metrics:
- type: Resource
resource:
name: memory
target:
type: AverageValue
averageValue: 750Mi
```

Pod metrics are unique metrics. These metrics provide information about Pods. To calculate the replica count, these metrics are averaged over all Pods and compared to a target number. They work similarly to resource metrics, with the exception that they only support a target type of AverageValue. Advanced cluster monitoring configuration is necessary for these cluster-specific KPIs. Below is the sample snippet for pod metrics.

```
type: Pods
pods:
metric:
name: packets-per-second
target:
type: AverageValue
averageValue: 50k
```

Instead of describing Pods, object metrics describe another object within the same namespace. Value and AverageValue target types are supported by object metrics. Value compares the target directly to the statistic that the API returned. Before being compared to the target, the value obtained from the custom metrics API using AverageValue is divided by the number of Pods. Below is the example for requests-per-second Object metric.

```
type: Object
object:
metric:
name: requests-per-second
describedObject:
apiVersion: networking.k8s.io/v1
kind: Ingress
name: main-route
target:
type: Value
value: 2k
```

The HorizontalPodAutoscaler will take into account each metric individually if you supply more than one of these metric blocks. For every metric, the HorizontalPodAutoscaler will compute suggested replica counts; it will then select the metric with the highest replica count. Below is the sample.

```
metrics:
- type: Resource
resource:
name: cpu
target:
type: Utilization
averageUtilization: 50
- type: Pods
pods:
metric:
name: packets-per-second
target:
type: AverageValue
averageValue: 1k
```

With the above configuration the HorizontalPodAutoscaler would try to make sure each pod was serving 1000 packets per second and using about 50% of the CPU that was requested.

Advantages of Horizontal Pod Auto Scaling:

One potent feature in Kubernetes that makes it possible to use resources more effectively, particularly during moments of peak consumption, is Horizontal Pod Autoscaling. By enabling the system to automatically add or remove resources as needed, it removes the need for human interaction and guarantees that your applications can handle the demand. By doing this, you can prevent problems like slowdowns and unavailability and guarantee that your apps operate smoothly even when they are under heavy pressure.

Best Practices for Kubernetes HPA

- To leverage capabilities like as HPA, which are part of Kubernetes, you must design the application with horizontal scaling in mind. Adding native support for concurrent pod execution through a microservice architecture is necessary to accomplish this.
- Instead of connecting the HPA resource straight to a ReplicaSet controller or Replication controller, use it on a Deployment object.
- To enable version control over HPA resources, create them using the declarative form. This method makes it easier to monitor configuration changes over time.
- When utilizing HPA, be sure to provide the resource demands for the pods. Requests for resources will allow HPA to scale pods in the best possible way.

Limitations of Kubernetes HPA

- HPA and Vertical Pod Autoscaler based on CPU or Memory measurements are incompatible. When HPA activates VPA, it must employ one or more custom metrics to prevent scaling conflicts with VPA, as VPA can only scale depending on CPU and memory values. To use custom metrics, HPA needs a custom metrics adaptor, which is available from each cloud provider.
- Only stateless programs that allow for the simultaneous operation of numerous instances can use HPA. HPA is also applicable to stateful sets that depend on replica pods. HPA does not apply to apps that cannot operate in numerous pods.
- Applications using HPA (and VPA) run the risk of experiencing delays and disruptions because these algorithms do not account for IOPS, networks, or storage.
- Finding waste in the Kubernetes cluster caused by the requested resources at the container level that are reserved but not used is still the administrators' responsibility after HPA. Kubernetes does not handle the detection of inefficient container usage; machine learning-powered third-party software is necessary.

Conclusion

Strong tools for controlling application scalability and resource efficiency are provided by HPA, one of Kubernetes' autoscaling capabilities. Your apps will stay responsive and economical under different loads if you are aware of their methods, use cases, and configuration. Remember to monitor and adjust settings to meet your unique workload requirements and attain peak performance in your Kubernetes environment as you implement these autoscaling solutions. HPA reduces manual intervention and provides with cost optimization [1-7].

References

1. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/.
2. https://www.densify.com/kubernetes-autoscaling/kubernetes-hpa/.
3. https://docs.avisi.cloud/blog/2023/03/20/auto-scaling-in-kubernetes-part-1/.
4. https://www.dbi-services.com/blog/kubernetes-deployment-autoscaling-using-memory-cpu/.
5. https://kubernetes.io/docs/concepts/cluster-administration/cluster-autoscaling/.
6. https://bluexp.netapp.com/blog/cvo-blg-kubernetes-scaling-the-comprehensive-guide-to-scaling-apps.
7. https://www.kubecost.com/kubernetes-autoscaling/kubernetes-hpa/.