**Kubernetes HPA: Custom Metrics for Effective CPU & Memory Scaling**

**Introduction and Summary:**

Kubernetes has become the de facto standard for container orchestration, providing a powerful platform for managing containerized applications at scale. One essential feature of Kubernetes is its autoscaling capabilities, which allows applications to scale up or down based on workload and performance metrics. In this article, we will explore the Horizontal Pod Autoscaler (HPA), a key component of Kubernetes autoscaling. We will delve into the basics of HPA, how it works, and how you can enhance its performance using custom metrics and resource limits. By the end of this article, you’ll have a solid understanding of HPA and how to configure it to optimize your Kubernetes deployments.

**Autoscaling in Kubernetes:**

Autoscaling is a critical feature of modern container orchestration systems, enabling applications to automatically adjust their resources based on demand and performance metrics. This dynamic scaling allows systems to maintain optimal performance and efficiency while minimizing operational costs.

In Kubernetes, autoscaling can be implemented at different levels:

1. Cluster Autoscaler: This component scales the entire Kubernetes cluster by adding or removing nodes from the cluster based on resource utilization and demand.
2. Horizontal Pod Autoscaler (HPA): The HPA adjusts the number of replicas for a specific deployment or stateful set based on pre-defined performance metrics such as CPU utilization, memory usage, or custom metrics.
3. Vertical Pod Autoscaler (VPA): The VPA automatically adjusts the CPU and memory requests and limits of individual containers within a pod, based on historical usage patterns and current resource demands.

**The importance of autoscaling:**

Autoscaling provides numerous benefits in maintaining an efficient and resilient system, including:

1. Resource optimization: Autoscaling ensures that your application uses the right amount of resources to meet its performance requirements, reducing the risk of over-provisioning or under-provisioning.
2. Cost efficiency: By automatically adjusting the resources according to the workload, you can minimize infrastructure costs, as you only pay for the resources you actually need.
3. Improved reliability: Autoscaling helps maintain the availability and performance of your applications by scaling out during periods of high demand and scaling in when demand decreases, preventing potential bottlenecks or system failures.
4. Enhanced user experience: By ensuring that your applications have the necessary resources to handle varying workloads, autoscaling can improve the overall user experience by reducing latency and maintaining consistent performance.

**Horizontal Pod Autoscaler (HPA) in Kubernetes**

The basic working mechanism of the Horizontal Pod Autoscaler (HPA) in Kubernetes involves monitoring, scaling policies, and the Kubernetes Metrics Server. Let’s break down each component:

1. **Monitoring**

HPA continuously monitors the metrics of the deployed pods in a Kubernetes cluster. By default, HPA monitors CPU utilization but can also be configured to monitor memory usage, custom metrics, or other per-pod metrics.

For per-pod resource metrics like CPU, HPA fetches metrics from the resource metrics API for each targeted pod. Based on target utilization or raw values, the controller computes a scaling ratio from the average of these values across all targeted pods. If some containers lack relevant resource requests, CPU utilization won’t be defined, and autoscaling won’t occur for that metric.

For per-pod custom metrics, the controller operates similarly but uses raw values instead of utilization values.

For object and external metrics, HPA fetches a single metric describing the object, compares it to the target value, and generates a scaling ratio. In the autoscaling/v2 API version, this value can be divided by the number of pods before comparison.

These metrics are collected and reported by the Kubernetes Metrics Server, which aggregates resource usage data from the kubelet running on each node.

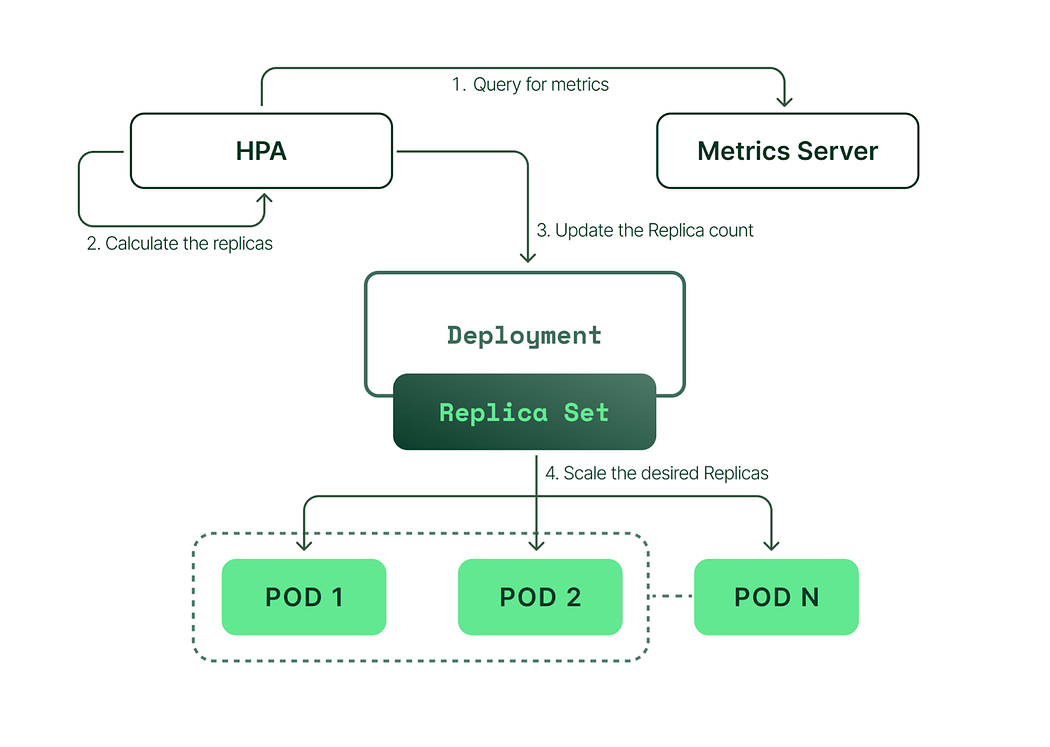


Image source: [Kubecost](https://www.kubecost.com/kubernetes-autoscaling/kubernetes-hpa/" \t "_blank) | An illustration of how the Horizontal Pod Autoscaler (HPA) works in Kubernetes.

**2. Scaling Policies:**

When configuring HPA, you define scaling policies that determine how the autoscaler should react to changes in metrics. These policies include:

* Target metric value: This is the desired value for the metric you want HPA to maintain. For example, you might set a target CPU utilization of 50% to ensure that your pods are neither overburdened nor underutilized.
* Min and max replicas: These values define the minimum and maximum number of replicas HPA can scale your deployment to. This prevents excessive scaling, which could lead to overloading the cluster or consuming too many resources.

**3. Scaling Decisions:**

HPA uses the collected metrics and the defined scaling policies to make scaling decisions. If the monitored metric exceeds the target value, HPA will increase the number of replicas in the deployment or stateful set to distribute the load more evenly. Conversely, if the metric falls below the target value, HPA will reduce the number of replicas to save resources.

**4. Kubernetes Metrics Server:**

The Kubernetes Metrics Server is a cluster-wide aggregator of resource usage data. It collects data from the kubelet on each node and provides metrics to the HPA and other components that require resource usage information. The Metrics Server is an essential component for enabling autoscaling and other features that rely on real-time metrics in Kubernetes.

In summary, the Horizontal Pod Autoscaler in Kubernetes works by continuously monitoring pod metrics, applying scaling policies based on target values and replica limits, and making scaling decisions to maintain optimal resource utilization. The Kubernetes Metrics Server plays a crucial role in providing the necessary data for HPA to make informed decisions.

**Custom Metrics in HPA**

Custom metrics are user-defined performance indicators that extend the default resource metrics (e.g., CPU and memory) supported by the Horizontal Pod Autoscaler (HPA) in Kubernetes. By default, HPA bases its scaling decisions on pod resource requests, which represent the minimum resources required for the pod to run. However, this approach might not be ideal for optimal performance. Instead, it’s often more beneficial to scale based on resource limits, as this ensures your application doesn’t reach its maximum resource constraints. Custom metrics enable more granular and application-specific autoscaling decisions, leading to better resource utilization and system performance.

**Why Custom Metrics Are Necessary:**

While the default metrics provided by Kubernetes, such as CPU and memory usage based on resource requests, are useful for many scenarios, they may not be sufficient for all applications. Scaling based on resource limits ensures that your application can handle varying workloads without hitting its maximum allowed resources. Custom metrics allow you to tailor the HPA’s scaling behavior based on your application’s specific needs, enabling more precise and efficient autoscaling.

**Using Custom Metrics in HPA:**

To use custom metrics in HPA, you need to:

1. Ensure your cluster is set up to support custom metrics. This typically involves deploying a custom metrics API server and configuring the necessary monitoring tools, such as Prometheus.
2. Define custom metrics in your application code, if needed, and expose them through an appropriate endpoint.
3. Configure HPA to use the custom metrics by specifying them in the HPA manifest.

**Examples of Custom Metrics and Their Use Cases:**

1. Request rate: For applications where the number of incoming requests has a significant impact on resource consumption, you can define a custom metric based on request rate. This enables HPA to scale the number of replicas based on the actual workload rather than just CPU or memory usage.

Use case: An API gateway that needs to handle varying levels of incoming traffic.

2. Queue length: For applications that process tasks from a queue, you can create a custom metric based on the queue length. This allows HPA to scale the application based on the backlog of tasks, ensuring that processing capacity matches the workload.

Use case: A background job processing service that consumes tasks from a message queue.

3. Application-specific metrics: You may have unique performance indicators specific to your application, such as the number of active user sessions or the rate of database transactions. Creating custom metrics based on these indicators can help HPA make more informed scaling decisions tailored to your application’s behavior.

Use case: An e-commerce platform that experiences fluctuations in user activity and needs to scale its services accordingly.

In summary, custom metrics in HPA enable more precise and application-specific autoscaling by extending the default resource metrics supported by Kubernetes. By leveraging custom metrics, you can optimize resource utilization and performance for a wider range of applications and use cases.

## Configuring HPA with CPU and Memory Limits

**Setting CPU and memory limits for your application is crucial for several reasons:**

1. Resource management: By specifying resource limits, you prevent individual pods or containers from consuming excessive resources, which could affect other workloads running on the same cluster.
2. Predictable performance: Setting limits ensures that your application has enough resources to perform optimally under varying workloads, minimizing the chances of performance degradation.
3. Cost optimization: By limiting resource usage, you can avoid unnecessary expenses on cloud resources or on-premises hardware.
4. Efficient autoscaling: Properly configured resource limits enable the Horizontal Pod Autoscaler (HPA) to make better scaling decisions, ensuring that your application scales up or down based on actual resource needs.

**Step-by-Step Guide on Configuring HPA with Custom Metrics and Resource Limits:**

1. Set up Prometheus. My recommendation is using the [kube-prometheus-stack Helm Chart](https://artifacthub.io/packages/helm/prometheus-community/kube-prometheus-stack" \t "_blank), which deploys cAdvisor and other necessary components.
2. Create custom metrics in Prometheus to monitor CPU and memory usage based on resource limits. Add the following examples to your Prometheus configuration:

CPU Usage Limits custom metric example:

- record: pod:cpu\_usage\_percentage:ratio  
 expr: |  
 sum by (pod, namespace) (  
 node\_namespace\_pod\_container:container\_cpu\_usage\_seconds\_total:sum\_irate{cluster="",namespace!="",pod!=""}  
 )  
 /  
 sum by (pod, namespace) (  
 kube\_pod\_container\_resource\_limits{cluster="",job="kube-state-metrics",namespace!="",pod!="",resource="cpu"}  
 ) \* 100

Memory Usage Limits custom metric example:

- record: pod:memory\_usage\_percentage:ratio  
 expr: |  
 sum by (pod, namespace) (  
 container\_memory\_working\_set\_bytes{cluster="",container!="",image!="",job="kubelet",metrics\_path="/metrics/cadvisor",namespace!="",pod!=""}  
 )  
 /  
 sum by (pod, namespace) (  
 kube\_pod\_container\_resource\_limits{cluster="",job="kube-state-metrics",namespace!="",pod!="",resource="memory"}  
 ) \* 100

3. Configure the [Prometheus Adapter Chart](https://artifacthub.io/packages/helm/prometheus-community/prometheus-adapter) that will replace the default [Kubernetes Metrics-Server](https://github.com/kubernetes-sigs/metrics-server). Follow these steps:

4. Configure the Prometheus Adapter to use Prometheus metrics:

prometheus:  
 url: http://prometheus.monitoring.svc  
 port: 9090

5. Add custom metrics to the Prometheus Adapter (these metrics will be found in Prometheus):

rules:  
 default: true  
 custom:  
 - seriesQuery: 'pod:memory\_usage\_percentage:ratio{namespace!="",pod!=""}'  
 resources:  
 overrides:  
 namespace: {resource: "namespace"}  
 pod: {resource: "pod"}  
 metricsQuery: '<<.Series>>{<<.LabelMatchers>>} / 100'  
   
 - seriesQuery: 'pod:cpu\_usage\_percentage:ratio{namespace!="",pod!=""}'  
 resources:  
 overrides:  
 namespace: {resource: "namespace"}  
 pod: {resource: "pod"}  
 metricsQuery: '<<.Series>>{<<.LabelMatchers>>} / 100'

You can find more information about Prometheus Adapter rules in [official documentation](https://github.com/kubernetes-sigs/prometheus-adapter/blob/master/docs/config-walkthrough.md).

6. For resource metrics, you can customize queries to collect CPU and memory:

resource:  
 cpu:  
 containerQuery: |  
 sum by (<<.GroupBy>>) (  
 rate(container\_cpu\_usage\_seconds\_total{container!="",<<.LabelMatchers>>}[3m])  
 )  
 nodeQuery: |  
 sum by (<<.GroupBy>>) (  
 rate(node\_cpu\_seconds\_total{mode!="idle",mode!="iowait",mode!="steal",<<.LabelMatchers>>}[3m])  
 )  
 resources:  
 overrides:  
 node:  
 resource: node  
 namespace:  
 resource: namespace  
 pod:  
 resource: pod  
 containerLabel: container  
 memory:  
 containerQuery: |  
 sum by (<<.GroupBy>>) (  
 avg\_over\_time(container\_memory\_working\_set\_bytes{container!="",<<.LabelMatchers>>}[3m])  
 )  
 nodeQuery: |  
 sum by (<<.GroupBy>>) (  
 avg\_over\_time(node\_memory\_MemTotal\_bytes{<<.LabelMatchers>>}[3m])  
 -  
 avg\_over\_time(node\_memory\_MemAvailable\_bytes{<<.LabelMatchers>>}[3m])  
 )  
 resources:  
 overrides:  
 node:  
 resource: node  
 namespace:  
 resource: namespace  
 pod:  
 resource: pod  
 containerLabel: container  
 window: 3m

7. Deploy Prometheus Adapter

helm repo add --force-update prometheus-community https://prometheus-community.github.io/helm-charts  
  
helm upgrade --install -n monitoring \  
prometheus-adapter prometheus-community/prometheus-adapter --version 4.1.1 -f metrics-server.yaml

8. Checking if the custom metric was successful applied

kubectl get --raw "/apis/custom.metrics.k8s.io/v1beta1/" | jq

9. For demonstration, deploy NGINX with resource limits and requests defined:

helm repo add --force-update bitnami https://charts.bitnami.com/bitnami  
  
helm upgrade --install \  
--set resources.limits.cpu=100m \  
--set resources.limits.memory=128Mi \  
--set resources.requests.cpu=50m \  
--set resources.requests.memory=64Mi \  
nginx bitnami/nginx --version 13.2.29

10. Apply HPA with Custom Metrics

apiVersion: autoscaling/v2  
kind: HorizontalPodAutoscaler  
metadata:  
 name: nginx-hpa  
spec:  
 scaleTargetRef:  
 apiVersion: apps/v1  
 kind: Deployment  
 name: nginx  
 minReplicas: 1  
 maxReplicas: 10  
 metrics:  
 - type: Pods  
 pods:  
 metric:  
 name: pod:memory\_usage\_percentage:ratio  
 target:   
 type: Utilization  
 averageValue: 0.8 # 80%  
 - type: Pods  
 pods:  
 metric:  
 name: pod:cpu\_usage\_percentage:ratio  
 target:   
 type: Utilization  
 averageValue: 0.8 # 80%

kubectl apply -f demo/nginx-hpa.yaml

In Helm Chart, an HPA template would look like this:  
templates/hpa.yaml

{{- if .Values.autoscaling.enabled }}  
apiVersion: autoscaling/v2  
kind: HorizontalPodAutoscaler  
metadata:  
 name: {{ include "app.fullname" . }}  
 labels:  
 {{- include "app.labels" . | nindent 4 }}  
spec:  
 scaleTargetRef:  
 apiVersion: apps/v1  
 kind: Deployment  
 name: {{ include "app.fullname" . }}  
 minReplicas: {{ .Values.autoscaling.minReplicas }}  
 maxReplicas: {{ .Values.autoscaling.maxReplicas }}  
 metrics:  
 {{- if .Values.autoscaling.targetCPUUtilizationPercentage }}  
 - type: Pods  
 pods:  
 metric:  
 name: pod:cpu\_usage\_percentage:ratio  
 averageValue: {{- div .Values.autoscaling.targetCPUUtilizationPercentage 100 -}}  
 {{- end }}  
 {{- if .Values.autoscaling.targetMemoryUtilizationPercentage }}  
 - type: Pods  
 pods:  
 metric:  
 name: pod:memory\_usage\_percentage:ratio  
 averageValue: {{- div .Values.autoscaling.targetMemoryUtilizationPercentage 100 -}}  
 {{- end }}  
{{- end }}

values.yaml

autoscaling:  
 enabled: true  
 minReplicas: 1  
 maxReplicas: 2  
 targetCPUUtilizationPercentage: 80  
 targetMemoryUtilizationPercentage: 80

11. Check if HPA is working:

kubectl get hpa nginx-hpa