Omar Latrach

TECHNICAL ARTICLE

PUBLISHED · SEPTEMBER 24, 2026

Workload Autoscaling with the Horizontal Pod Autoscaler

Understand how the Horizontal Pod Autoscaler turns workload metrics into replica changes, why CPU requests matter, and how to configure, observe, and troubleshoot HPA behavior.

This is Learn post 14 of the 54-post Certified Kubernetes Administrator (CKA) preparation path. Earlier lessons established Deployments, ReplicaSets, resource requests, and scheduling. Now we connect them: a HorizontalPodAutoscaler (HPA) changes a workload's desired replica count when observed demand changes.

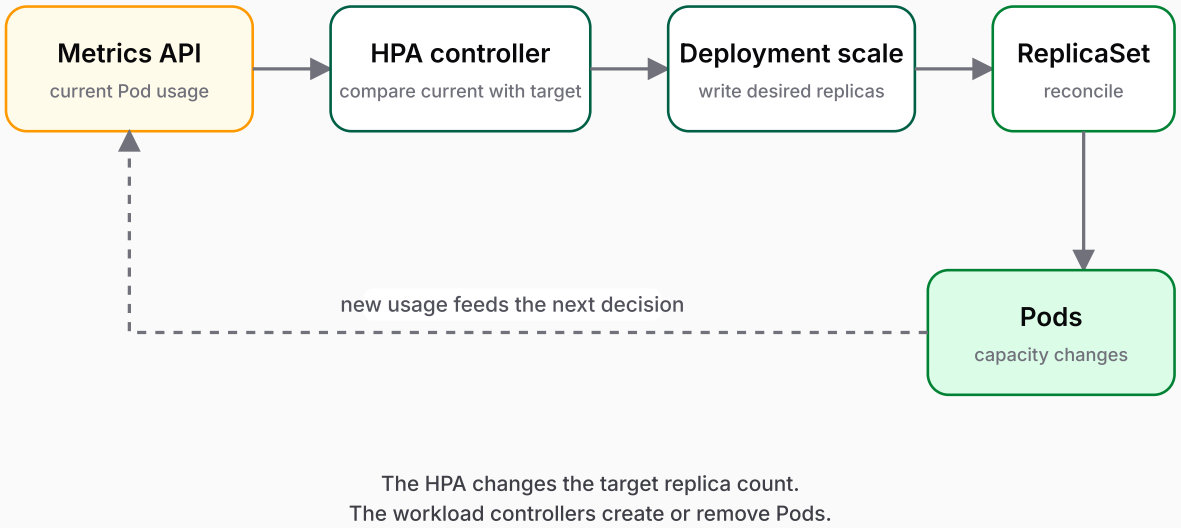
The important idea is not merely that Kubernetes can add Pods. It is which controller decides the new count, where its measurements come from, and which other controllers turn that number into running capacity.

What you'll learn

- Read the HPA as a control loop: observe a metric, compare it with a target, and write a desired replica count.
- Configure CPU-based autoscaling with the stable autoscaling/v2 API.
- Explain why CPU utilization depends on container requests rather than limits.
- Observe normal scale-up and scale-down behavior with kubectl get, top, describe, and events.
- Recognize the difference between replica autoscaling, Pod resource sizing, and cluster capacity.

The mental model: measure, decide, delegate

> The HPA does not create Pods. It measures demand, chooses a replica count, and delegates Pod creation or removal to the workload's existing controllers.



The HPA controls a target's replica count

Metrics drive the HPA decision; the Deployment and ReplicaSet still own Pod reconciliation.

The HPA controller runs in the control plane. On each control-loop iteration, it finds the resource named by scaleTargetRef, discovers that target's Pods through its selector, reads the configured metrics, calculates a replica recommendation, and updates the target's scale subresource.

For a Deployment, that write changes the desired replica count. The Deployment controller updates its ReplicaSet, and the ReplicaSet controller creates or deletes Pods. The scheduler then places any new Pods using the rules from posts 11–13.

What the HPA object expresses

An HPA specification answers four questions:

- Target: which scalable workload should receive replica updates?
- Metric: which signal should represent demand?
- Target value: what average level should the controller try to maintain?

- Bounds: how few or how many replicas may the controller request?

Deployments and StatefulSets expose the scale interface that HPA needs. A DaemonSet does not: its Pod count follows eligible nodes, not a freely chosen replica count.

CPU utilization is relative to requests

For a Resource metric with type Utilization, Kubernetes compares measured use with the requested resource. If a container uses 100 millicores and requests 200 millicores, its CPU utilization is 50%. A 500 millicore CPU limit does not change that percentage.

CPU utilization · plaintext

```
utilization = current CPU usage ÷ requested CPU

100m usage ÷ 200m request = 50% utilization
```

Because the request is the denominator, CPU utilization cannot be calculated correctly for a Pod when a relevant container has no CPU request. This is why the resource controls from post 10 are part of the autoscaling design, not separate decoration.

The simplified replica calculation is current replicas multiplied by the ratio of current metric to target metric, rounded up. For four replicas averaging 90% CPU against a 50% target, the recommendation is ceil(4 × 90 ÷ 50), or eight replicas.

Simplified HPA ratio · plaintext

```
desired replicas = ceil(current replicas × current metric ÷ target metric)

ceil(4 × 90 ÷ 50) = 8
```

The real controller also applies tolerance, readiness handling, missing-metric safeguards, configured scaling policies, and min/max bounds. Remember the ratio for intuition; use HPA status and events to understand an actual decision.

Check the metrics path first

CPU and memory resource metrics normally arrive through the metrics.k8s.io API, commonly provided by Metrics Server. HPA is built into Kubernetes, but the resource-metrics pipeline must be available in the cluster.

```bash
kubectl get apiservice v1beta1.metrics.k8s.io
kubectl top nodes
kubectl top pods -A
```

An Available=True APIService and successful top output show that resource metrics are queryable. A new Metrics Server installation may need a short collection interval before values appear. If these commands fail, creating an HPA will not repair the metrics path.

Representative APIService status · plaintext

```
NAME                     SERVICE                      AVAILABLE   AGE
v1beta1.metrics.k8s.io   kube-system/metrics-server   True        20m
```

Build an autoscalable workload

This demonstration uses a small HTTP workload designed to consume CPU while serving requests. The Service gives the load generator a stable name. The CPU request gives the HPA a utilization denominator.

web.yaml · yaml

```
apiVersion: v1
kind: Namespace
metadata:
  name: cka-hpa
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: web
  namespace: cka-hpa
spec:
  replicas: 1
  selector:
    matchLabels:
      app: web
  template:
    metadata:
      labels:
        app: web
    spec:
      containers:
        - name: web
          image: registry.k8s.io/hpa-example:latest
          ports:
            - containerPort: 80
          resources:
            requests:
              cpu: 200m
            limits:
              cpu: 500m
---
apiVersion: v1
kind: Service
metadata:
  name: web
  namespace: cka-hpa
spec:
  selector:
    app: web
  ports:
    - port: 80
      targetPort: 80
```

bash

```
kubectl apply -f web.yaml
kubectl rollout status deployment/web -n cka-hpa
kubectl get deployment,pods,service -n cka-hpa
```

Wait for the Deployment to become available before judging its metrics. A running but unready or newly started Pod can be treated conservatively by the autoscaler while its measurements settle.

Create the HPA

The imperative command is useful for speed and for generating a manifest. The percent sign means average CPU utilization relative to requested CPU, not a raw CPU quantity.

```bash
kubectl autoscale deployment web \
  --namespace=cka-hpa \
  --cpu=50% \
  --min=1 \
  --max=10 \
  --dry-run=client -o yaml
```

For a reusable definition, keep the stable autoscaling/v2 manifest. Its target says to change the scale of Deployment/web; its metric says to maintain average CPU utilization near 50%; its bounds permit one through ten replicas.

hpa.yaml · yaml

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

bash

```bash
kubectl apply -f hpa.yaml
kubectl get hpa web -n cka-hpa
kubectl describe hpa web -n cka-hpa
```

Representative steady state after metrics arrive · plaintext

```
NAME   REFERENCE       TARGETS      MINPODS   MAXPODS   REPLICAS   AGE
web    Deployment/web   cpu: 1%/50%   1        10        1          2m
```

Read TARGETS as current average over desired average. The workload is nearly idle, so one replica satisfies the target and the minimum bound prevents a further decrease.

Watch a normal scale-up

Create one load-generator Pod that repeatedly calls the Service. This is a worked observation: the request stream increases CPU use in the web Pods, which changes the next HPA calculation.

```bash
kubectl run load-generator \
  --namespace=cka-hpa \
  --image=busybox:1.36.1 \
  --restart=Never \
  -- /bin/sh -c 'while true; do wget -q -O- http://web > /dev/null; done'
```

In another terminal, watch the metric, the HPA's desired count, and the Deployment's reconciliation. Sampling and image startup take time, so the transitions are not instantaneous.

```bash
kubectl get hpa web -n cka-hpa --watch

# In a separate terminal:
kubectl get deployment web -n cka-hpa --watch
kubectl top pods -n cka-hpa
```

```
Representative HPA snapshots · plaintext

NAME   REFERENCE       TARGETS        MINPODS   MAXPODS   REPLICAS
web    Deployment/web   cpu: 1%/50%     1         10        1
web    Deployment/web   cpu: 240%/50%   1         10        1
web    Deployment/web   cpu: 48%/50%    1         10        5
```

At the high-usage snapshot, the simplified recommendation is ceil(1 × 240 ÷ 50), or five replicas. The HPA writes that desired count; the Deployment and ReplicaSet produce the additional Pods. Once requests are shared across them, average utilization approaches the target. Exact measurements and timing vary by cluster.

Scale-down is deliberately cautious

Stop the demand and continue watching. Metrics fall quickly, but the default scale-down stabilization window considers recent recommendations for five minutes. This delay helps avoid repeatedly removing and recreating Pods around a noisy threshold.

```bash
kubectl delete pod load-generator -n cka-hpa
kubectl get hpa web -n cka-hpa --watch
```

The autoscaling/v2 behavior field can customize scale-up and scale-down policies and stabilization windows. For CKA administration, first recognize the default cautious scale-down and inspect the stored behavior before assuming that a replica count is stuck.

```bash
kubectl explain horizontalpodautoscaler.spec.behavior --recursive
kubectl get hpa web -n cka-hpa -o yaml
```

Read status and conditions, not only replica count

kubectl describe hpa joins specification, measurements, controller conditions, and scaling events. Three condition types provide a compact diagnosis:

- AbleToScale: the controller can read and update the target's scale.
- ScalingActive: the controller obtained a valid metric and calculated a recommendation.
- ScalingLimited: the recommendation was constrained, commonly by minReplicas or maxReplicas.

```bash
kubectl describe hpa web -n cka-hpa
kubectl get hpa web -n cka-hpa \
  -o custom-columns='NAME:.metadata.name,CURRENT:.status.currentReplicas,DESIRED:.status.desiredReplicas,LAST-SCALE:.status.lastScaleTime'
kubectl get events -n cka-hpa --sort-by=.metadata.creationTimestamp
```

A ScalingLimited=True condition is not automatically an error. At idle, a recommendation below one replica is correctly limited by minReplicas: 1. Under heavy load, a recommendation above ten is correctly limited by maxReplicas: 10.

Do not confuse three different scaling problems

Replica count

HPA changes how many Pod replicas a scalable workload requests. Repeated manual kubectl scale commands fight that control loop because both actors write the same desired count.

Pod size

Requests and limits size each Pod and affect scheduling and runtime enforcement. HPA does not rewrite those values. It uses requests when percentage-based resource targets need a baseline.

Cluster capacity

HPA can request more replicas than the current nodes can schedule. In that case, its desired replica count increases while new Pods remain Pending. HPA scales a workload; it does not add nodes or bypass resource, taint, affinity, and topology constraints.

A compact troubleshooting sequence

Start at the HPA, then follow the same relationship as the control loop. Each command answers a different question.

```bash
# 1. Does the HPA have a metric and a recommendation?
kubectl get hpa web -n cka-hpa
kubectl describe hpa web -n cka-hpa

# 2. Does the metrics API return current Pod data?
kubectl top pods -n cka-hpa

# 3. Do all workload containers declare the requested CPU?
kubectl get deployment web -n cka-hpa \
  -o jsonpath='{range .spec.template.spec.containers[*]}{.name}{" request="}{.resources.requests.cpu}{"\n"}{end}'

# 4. Did the target receive the desired replica count?
kubectl get deployment web -n cka-hpa

# 5. If replicas exist but Pods are not running, why?
kubectl get pods -n cka-hpa -o wide
kubectl get events -n cka-hpa --sort-by=.metadata.creationTimestamp
```

TARGETS showing unknown usually points toward unavailable metrics, a missing resource request, or Pods whose metrics are not yet usable. A desired count at maxReplicas means the HPA is working but bounded. Pending replicas move the investigation from autoscaling to scheduling and capacity.

Beyond one CPU metric

autoscaling/v2 can use memory, container resource metrics, custom metrics, external metrics, and multiple metrics. With multiple metrics, the controller calculates a recommendation for each usable metric and chooses the largest recommendation. Custom and external metrics require cluster-specific API adapters.

CPU is the clearest first model because adding replicas often spreads CPU-bound request load. Memory may not decrease when replicas are added, so choose a signal whose relationship to replica count actually represents capacity. Detailed monitoring belongs later in post 36; here the key is that HPA consumes a metric API rather than inventing measurements itself.

What to remember

- HPA measures and writes desired replicas; workload controllers create or remove Pods.
- Percentage CPU targets use requests as their denominator, not limits.
- Metrics availability, valid requests, and a scalable target are prerequisites for useful decisions.
- minReplicas and maxReplicas are safety bounds; stabilization and tolerance prevent noisy changes.
- HPA can ask for Pods that the cluster cannot schedule. Replica demand and node capacity are separate concerns.

Official references

Horizontal Pod Autoscaling explains the controller loop, metric sources, algorithm safeguards, and scaling behavior.

HorizontalPodAutoscaler walkthrough provides the official end-to-end CPU autoscaling demonstration.

autoscaling/v2 API reference defines the HPA specification, status, metric targets, and behavior fields.

Clean up the demonstration

```bash
kubectl delete namespace cka-hpa
```

Deleting the namespace removes the Deployment, Service, HPA, load generator if it is still present, and all demonstration Pods.