Horizontal Pod Autoscaling

In Kubernetes, a *HorizontalPodAutoscaler* automatically updates a workload resource (such as a Deployment or StatefulSet), with the aim of automatically scaling the workload to match demand.

Horizontal scaling means that the response to increased load is to deploy more Pods. This is different from *vertical* scaling, which for Kubernetes would mean assigning more resources (for example: memory or CPU) to the Pods that are already running for the workload.

If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource (the Deployment, StatefulSet, or other similar resource) to scale back down.

Horizontal pod autoscaling does not apply to objects that can't be scaled (for example: a DaemonSet.)

The HorizontalPodAutoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The horizontal pod autoscaling controller, running within the Kubernetes control plane, periodically adjusts the desired scale of its target (for example, a Deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric you specify.

There is walkthrough example of using horizontal pod autoscaling.

How does a HorizontalPodAutoscaler work?

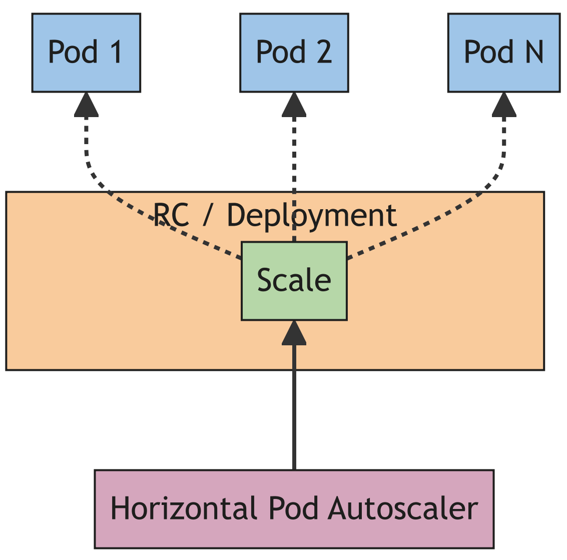

Figure 1. HorizontalPodAutoscaler controls the scale of a Deployment and its ReplicaSet

Kubernetes implements horizontal pod autoscaling as a control loop that runs intermittently (it is not a continuous process). The interval is set by the `--horizontal-pod-autoscaler-sync-period` parameter to the `kube-controller-manager` (and the default interval is 15 seconds).

Once during each period, the controller manager queries the resource utilization against the metrics specified in each HorizontalPodAutoscaler definition. The controller manager finds the target resource defined by the `scaleTargetRef`, then selects the pods based on the target resource's `.spec.selector` labels, and obtains the metrics from either the resource metrics API (for per-pod resource metrics), or the custom metrics API (for all other metrics).

- For per-pod resource metrics (like CPU), the controller fetches the metrics from the resource metrics API for each Pod targeted by the HorizontalPodAutoscaler. Then, if a target utilization value is set, the controller calculates the utilization value as a percentage of the equivalent resource request on the containers in each Pod. If a target raw value is set, the raw metric values are used directly. The controller then takes the mean of the utilization or the raw value (depending on the type of target specified) across all targeted Pods, and produces a ratio used to scale the number of desired replicas.

  Please note that if some of the Pod's containers do not have the relevant resource request set, CPU utilization for the Pod will not be defined and the autoscaler will not take any action for that metric. See the algorithm details section below for more information about how the autoscaling algorithm works.

- For per-pod custom metrics, the controller functions similarly to per-pod resource metrics, except that it works with raw values, not utilization values.

- For object metrics and external metrics, a single metric is fetched, which describes the object in question. This metric is compared to the target value, to produce a ratio as above. In the `autoscaling/v2` API version, this value can optionally be divided by the number of Pods before the comparison is made.

The common use for HorizontalPodAutoscaler is to configure it to fetch metrics from aggregated APIs (`metrics.k8s.io` , `custom.metrics.k8s.io` , or `external.metrics.k8s.io`). The `metrics.k8s.io` API is usually provided by an add-on named Metrics Server, which needs to be launched separately. For more information about resource metrics, see Metrics Server.

Support for metrics APIs explains the stability guarantees and support status for these different APIs.

The HorizontalPodAutoscaler controller accesses corresponding workload resources that support scaling (such as Deployments and StatefulSet). These resources each have a subresource named `scale` , an interface that allows you to dynamically set the number of replicas and examine each of their current states. For general information about subresources in the Kubernetes API, see Kubernetes API Concepts.

Algorithm details

From the most basic perspective, the HorizontalPodAutoscaler controller operates on the ratio between desired metric value and current metric value:

```
desiredReplicas = ceil[currentReplicas * ( currentMetricValue / desiredMetricValue )]
```

For example, if the current metric value is `200m` , and the desired value is `100m` , the number of replicas will be doubled, since `200.0 / 100.0 == 2.0` If the current value is instead `50m` , you'll halve the number of replicas, since `50.0 / 100.0 == 0.5` . The control plane skips any scaling action if the ratio is sufficiently close to 1.0 (within a globally-configurable tolerance, 0.1 by default).

When a `targetAverageValue` or `targetAverageUtilization` is specified, the `currentMetricValue` is computed by taking the average of the given metric across all Pods in the HorizontalPodAutoscaler's scale target.

Before checking the tolerance and deciding on the final values, the control plane also considers whether any metrics are missing, and how many Pods are `Ready` . All Pods with a deletion timestamp set (objects with a deletion timestamp are in the process of being shut down / removed) are ignored, and all failed Pods are discarded.

If a particular Pod is missing metrics, it is set aside for later; Pods with missing metrics will be used to adjust the final scaling amount.

When scaling on CPU, if any pod has yet to become ready (it's still initializing, or possibly is unhealthy) *or* the most recent metric point for the pod was before it became ready, that pod is set aside as well.

Due to technical constraints, the HorizontalPodAutoscaler controller cannot exactly determine the first time a pod becomes ready when determining whether to set aside certain CPU metrics. Instead, it considers a Pod "not yet ready" if it's unready and transitioned to ready within a short, configurable window of time since it started. This value is configured with the `--horizontal-pod-autoscaler-initial-readiness-delay` flag, and its default is 30 seconds. Once a pod has become ready, it considers any transition to ready to be the first if it occurred within a longer, configurable time since it started. This value is configured with the `--horizontal-pod-autoscaler-cpu-initialization-period` flag, and its default is 5 minutes.

The `currentMetricValue / desiredMetricValue` base scale ratio is then calculated using the remaining pods not set aside or discarded from above.

If there were any missing metrics, the control plane recomputes the average more conservatively, assuming those pods were consuming 100% of the desired value in case of a scale down, and 0% in case of a scale up. This dampens the magnitude of any potential scale.

Furthermore, if any not-yet-ready pods were present, and the workload would have scaled up without factoring in missing metrics or not-yet-ready pods, the controller conservatively assumes that the not-yet-ready pods are consuming 0% of the desired metric, further dampening the magnitude of a scale up.

After factoring in the not-yet-ready pods and missing metrics, the controller recalculates the usage ratio. If the new ratio reverses the scale direction, or is within the tolerance, the controller doesn't take any scaling action. In other cases, the new ratio is used to decide any change to the number of Pods.

Note that the *original* value for the average utilization is reported back via the HorizontalPodAutoscaler status, without factoring in the not-yet-ready pods or missing metrics, even when the new usage ratio is used.

If multiple metrics are specified in a HorizontalPodAutoscaler, this calculation is done for each metric, and then the largest of the desired replica counts is chosen. If any of these metrics cannot be converted into a desired replica count (e.g. due to an error fetching the metrics from the metrics APIs) and a scale down is suggested by the metrics which can be fetched, scaling is skipped.

This means that the HPA is still capable of scaling up if one or more metrics give a `desiredReplicas` greater than the current value.

Finally, right before HPA scales the target, the scale recommendation is recorded. The controller considers all recommendations within a configurable window choosing the highest recommendation from within that window. This value can be configured using the `--horizontal-pod-autoscaler-downscale-stabilization` flag, which defaults to 5 minutes. This means that scaledowns will occur gradually, smoothing out the impact of rapidly fluctuating metric values.

API Object

The Horizontal Pod Autoscaler is an API resource in the Kubernetes `autoscaling` API group. The current stable version can be found in the `autoscaling/v2` API version which includes support for scaling on memory and custom metrics. The new fields introduced in `autoscaling/v2` are preserved as annotations when working with `autoscaling/v1`.

When you create a HorizontalPodAutoscaler API object, make sure the name specified is a valid DNS subdomain name. More details about the API object can be found at HorizontalPodAutoscaler Object.

Stability of workload scale

When managing the scale of a group of replicas using the HorizontalPodAutoscaler, it is possible that the number of replicas keeps fluctuating frequently due to the dynamic nature of the metrics evaluated. This is sometimes referred to as *thrashing*, or *flapping*. It's similar to the concept of *hysteresis* in cybernetics.

Autoscaling during rolling update

Kubernetes lets you perform a rolling update on a Deployment. In that case, the Deployment manages the underlying ReplicaSets for you. When you configure autoscaling for a Deployment, you bind a HorizontalPodAutoscaler to a single Deployment. The HorizontalPodAutoscaler manages the `replicas` field of the Deployment. The deployment controller is responsible for setting the `replicas` of the underlying ReplicaSets so that they add up to a suitable number during the rollout and also afterwards.

If you perform a rolling update of a StatefulSet that has an autoscaled number of replicas, the StatefulSet directly manages its set of Pods (there is no intermediate resource similar to ReplicaSet).

Support for resource metrics

Any HPA target can be scaled based on the resource usage of the pods in the scaling target. When defining the pod specification the resource requests like `cpu` and `memory` should be specified. This is used to determine the resource utilization and used by the HPA controller to scale the target up or down. To use resource utilization based scaling specify a metric source like this:

```yaml
type: Resource
resource:
  name: cpu
  target:
    type: Utilization
    averageUtilization: 60
```

With this metric the HPA controller will keep the average utilization of the pods in the scaling target at 60%. Utilization is the ratio between the current usage of resource to the requested resources of the pod. See Algorithm for more details about how the utilization is calculated and averaged.

> **Note:** Since the resource usages of all the containers are summed up the total pod utilization may not accurately represent the individual container resource usage. This could lead to situations where a single container might be running with high usage and the HPA will not scale out because the overall pod usage is still within acceptable limits.

Container resource metrics

FEATURE STATE: `Kubernetes v1.27 [beta]`

The HorizontalPodAutoscaler API also supports a container metric source where the HPA can track the resource usage of individual containers across a set of Pods, in order to scale the target resource. This lets you configure scaling thresholds for the containers that matter most in a particular Pod. For example, if you have a web application and a logging sidecar, you can scale based on the resource use of the web application, ignoring the sidecar container and its resource use.

If you revise the target resource to have a new Pod specification with a different set of containers, you should revise the HPA spec if that newly added container should also be used for scaling. If the specified container in the metric source is not present or only present in a subset of the pods then those pods are ignored and the recommendation is recalculated. See Algorithm for more details about the calculation. To use container resources for autoscaling define a metric source as follows:

```yaml
type: ContainerResource
containerResource:
  name: cpu
  container: application
  target:
    type: Utilization
    averageUtilization: 60
```

In the above example the HPA controller scales the target such that the average utilization of the cpu in the `application` container of all the pods is 60%.

Note:
If you change the name of a container that a HorizontalPodAutoscaler is tracking, you can make that change in a specific order to ensure scaling remains available and effective whilst the change is being applied. Before you update the resource that defines the container (such as a Deployment), you should update the associated HPA to track both the new and old container names. This way, the HPA is able to calculate a scaling recommendation throughout the update process.

Once you have rolled out the container name change to the workload resource, tidy up by removing the old container name from the HPA specification.

Scaling on custom metrics

FEATURE STATE: `Kubernetes v1.23 [stable]`

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

Provided that you use the `autoscaling/v2` API version, you can configure a HorizontalPodAutoscaler to scale based on a custom metric (that is not built in to Kubernetes or any Kubernetes component). The HorizontalPodAutoscaler controller then queries for these custom metrics from the Kubernetes API.

See Support for metrics APIs for the requirements.

Scaling on multiple metrics

FEATURE STATE: `Kubernetes v1.23 [stable]`

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

Provided that you use the `autoscaling/v2` API version, you can specify multiple metrics for a HorizontalPodAutoscaler to scale on. Then, the HorizontalPodAutoscaler controller evaluates each metric, and proposes a new scale based on that metric. The HorizontalPodAutoscaler takes the maximum scale recommended for each metric and sets the workload to that size (provided that this isn't larger than the overall maximum that you configured).

Support for metrics APIs

By default, the HorizontalPodAutoscaler controller retrieves metrics from a series of APIs. In order for it to access these APIs, cluster administrators must ensure that:

- The API aggregation layer is enabled.

- The corresponding APIs are registered:
    - For resource metrics, this is the `metrics.k8s.io` API, generally provided by metrics-server. It can be launched as a cluster add-on.
    - For custom metrics, this is the `custom.metrics.k8s.io` API. It's provided by "adapter" API servers provided by metrics solution vendors. Check with your metrics pipeline to see if there is a Kubernetes metrics adapter available.
    - For external metrics, this is the `external.metrics.k8s.io` API. It may be provided by the custom metrics adapters provided above.

For more information on these different metrics paths and how they differ please see the relevant design proposals for the HPA V2, custom.metrics.k8s.io and external.metrics.k8s.io.

For examples of how to use them see the walkthrough for using custom metrics and the walkthrough for using external metrics.

Configurable scaling behavior

FEATURE STATE: `Kubernetes v1.23 [stable]`

(the `autoscaling/v2beta2` API version previously provided this ability as a beta feature)

If you use the `v2` HorizontalPodAutoscaler API, you can use the `behavior` field (see the API reference) to configure separate scale-up and scale-down behaviors. You specify these behaviours by setting `scaleUp` and / or `scaleDown` under the `behavior` field.

You can specify a *stabilization window* that prevents flapping the replica count for a scaling target. Scaling policies also let you control the rate of change of replicas while scaling.

Scaling policies

One or more scaling policies can be specified in the `behavior` section of the spec. When multiple policies are specified the policy which allows the highest amount of change is the policy which is selected by default. The following example shows this behavior while scaling down:

```yaml
behavior:
  scaleDown:
    policies:
    - type: Pods
      value: 4
      periodSeconds: 60
    - type: Percent
      value: 10
      periodSeconds: 60
```

`periodSeconds` indicates the length of time in the past for which the policy must hold true. The maximum value that you can set for `periodSeconds` is 1800 (half an hour). The first policy *(Pods)* allows at most 4 replicas to be scaled down in one minute. The second policy *(Percent)* allows at most 10% of the current replicas to be scaled down in one minute.

Since by default the policy which allows the highest amount of change is selected, the second policy will only be used when the number of pod replicas is more than 40. With 40 or less replicas, the first policy will be applied. For instance if there are 80 replicas and the target has to be scaled down to 10 replicas then during the first step 8 replicas will be reduced. In the next iteration when the number of replicas is 72, 10% of the pods is 7.2 but the number is rounded up to 8. On each loop of the autoscaler controller the number of pods to be change is re-calculated based on the number of current replicas. When the number of replicas falls below 40 the first policy *(Pods)* is applied and 4 replicas will be reduced at a time.

The policy selection can be changed by specifying the `selectPolicy` field for a scaling direction. By setting the value to `Min` which would select the policy which allows the smallest change in the replica count. Setting the value to `Disabled` completely disables scaling in that direction.

Stabilization window

The stabilization window is used to restrict the flapping of replica count when the metrics used for scaling keep fluctuating. The autoscaling algorithm uses this window to infer a previous desired state and avoid unwanted changes to workload scale.

For example, in the following example snippet, a stabilization window is specified for `scaleDown`.

```yaml
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
```

When the metrics indicate that the target should be scaled down the algorithm looks into previously computed desired states, and uses the highest value from the specified interval. In the above example, all desired states from the past 5 minutes will be considered.

This approximates a rolling maximum, and avoids having the scaling algorithm frequently remove Pods only to trigger recreating an equivalent Pod just moments later.

Default Behavior

To use the custom scaling not all fields have to be specified. Only values which need to be customized can be specified. These custom values are merged with default values. The default values match the existing behavior in the HPA algorithm.

```yaml
behavior:
  scaleDown:
    stabilizationWindowSeconds: 300
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
  scaleUp:
    stabilizationWindowSeconds: 0
    policies:
    - type: Percent
      value: 100
      periodSeconds: 15
    - type: Pods
      value: 4
      periodSeconds: 15
    selectPolicy: Max
```

For scaling down the stabilization window is *300* seconds (or the value of the `--horizontal-pod-autoscaler-downscale-stabilization` flag if provided). There is only a single policy for scaling down which allows a 100% of the currently running replicas to be removed which means the scaling target can be scaled down to the minimum allowed replicas. For scaling up there is no stabilization window. When the metrics indicate that the target should be scaled up the target is scaled up immediately. There are 2 policies where 4 pods or a 100% of the currently running replicas may at most be added every 15 seconds till the HPA reaches its steady state.

Example: change downscale stabilization window

To provide a custom downscale stabilization window of 1 minute, the following behavior would be added to the HPA:

```yaml
behavior:
  scaleDown:
    stabilizationWindowSeconds: 60
```

Example: limit scale down rate

To limit the rate at which pods are removed by the HPA to 10% per minute, the following behavior would be added to the HPA:

```yaml
behavior:
  scaleDown:
    policies:
    - type: Percent
      value: 10
      periodSeconds: 60
```

To ensure that no more than 5 Pods are removed per minute, you can add a second scale-down policy with a fixed size of 5, and set `selectPolicy` to minimum. Setting `selectPolicy` to `Min` means that the autoscaler chooses the policy that affects the smallest number of Pods:

```yaml
behavior:
  scaleDown:
    policies:
    - type: Percent
      value: 10
      periodSeconds: 60
    - type: Pods
      value: 5
      periodSeconds: 60
    selectPolicy: Min
```

Example: disable scale down

The `selectPolicy` value of `Disabled` turns off scaling the given direction. So to prevent downscaling the following policy would be used:

```yaml
behavior:
  scaleDown:
    selectPolicy: Disabled
```

Support for HorizontalPodAutoscaler in kubectl

HorizontalPodAutoscaler, like every API resource, is supported in a standard way by `kubectl`. You can create a new autoscaler using `kubectl create` command. You can list autoscalers by `kubectl get hpa` or get detailed description by `kubectl describe hpa`. Finally, you can delete an autoscaler using `kubectl delete hpa`.

In addition, there is a special `kubectl autoscale` command for creating a HorizontalPodAutoscaler object. For instance, executing `kubectl autoscale rs foo --min=2 --max=5 --cpu-percent=80` will create an autoscaler for ReplicaSet *foo*, with target CPU utilization set to `80%` and the number of replicas between 2 and 5.

Implicit maintenance-mode deactivation

You can implicitly deactivate the HPA for a target without the need to change the HPA configuration itself. If the target's desired replica count is set to 0, and the HPA's minimum replica count is greater than 0, the HPA stops adjusting the target (and sets the `ScalingActive` Condition on itself to `false`) until you reactivate it by manually adjusting the target's desired replica count or HPA's minimum replica count.

Migrating Deployments and StatefulSets to horizontal autoscaling

When an HPA is enabled, it is recommended that the value of `spec.replicas` of the Deployment and / or StatefulSet be removed from their manifest(s). If this isn't done, any time a change to that object is applied, for example via `kubectl apply -f deployment.yaml`, this will instruct Kubernetes to scale the current number of Pods to the value of the `spec.replicas` key. This may not be desired and could be troublesome when an HPA is active.

Keep in mind that the removal of `spec.replicas` may incur a one-time degradation of Pod counts as the default value of this key is 1 (reference Deployment Replicas). Upon the update, all Pods except 1 will begin their termination procedures. Any deployment application afterwards will behave as normal and respect a rolling update configuration as desired. You can avoid this degradation by choosing one of the following two methods based on how you are modifying your deployments:

Client Side Apply (this is the default) | Server Side Apply

1. `kubectl apply edit-last-applied deployment/<deployment_name>`
2. In the editor, remove `spec.replicas`. When you save and exit the editor, `kubectl` applies the update. No changes to Pod counts happen at this step.
3. You can now remove `spec.replicas` from the manifest. If you use source code management, also commit your changes or take whatever other steps for revising the source code are appropriate for how you track updates.
4. From here on out you can run `kubectl apply -f deployment.yaml`

What's next

If you configure autoscaling in your cluster, you may also want to consider using cluster autoscaling to ensure you are running the right number of nodes.

For more information on HorizontalPodAutoscaler:

- Read a walkthrough example for horizontal pod autoscaling.
- Read documentation for `kubectl autoscale`.
- If you would like to write your own custom metrics adapter, check out the boilerplate to get started.
- Read the API reference for HorizontalPodAutoscaler.

Feedback

Was this page helpful?

Yes No

Last modified February 18, 2024 at 2:59 PM PST: Add concept page about cluster autoscaling (b39e01b971)