Managing Kubernetes Resource Limits

If you have ever worked with Kubernetes in production, you probably understand that feeling of needing to double check your resource configurations before finally deploying your workloads. Or worry about whether they were even configured in the first place. Proper resource consumption, after all, is a balancing act that can run awry.

Why is that? Well, once a pod is scheduled on a node, the pod starts immediately using compute resources based on the requirements defined. If gone unchecked, one simple misconfiguration can deprive the cluster's other running workloads from their needed resources, such as being Out Of Memory (OOM), and ultimately crash the cluster. Misconfigurations can also lead to the deployed pods going into the pending state if none of the worker nodes in the cluster are capable of meeting the pod's requirements.

On the flip side, the lack of oversight in resource allocation over time leads to significant financial waste, which could be channeled instead towards strategic engineering initiatives.

To keep our clusters healthy, it is critical to always specify resource requests and limits for computing resources. In this article, we'll give you a rundown of what you need to know about resource requests and limits. Let's get started.

How to set Resource Request & Limits

In the below example, we have a pod with two containers: `app` and `log-aggregator`. The sidecar container helps facilitate the proper execution of core business logic in the main container.

```
apiVersion: v1
kind: Pod
metadata:
 name: frontend
spec:
 containers:
 - name: app
   image: images.my-company.example/app:v4
   resources:
     requests:
       memory: "64Mi"
       cpu: "250m"
     limits:
       memory: "128Mi"
       cpu: "500m"
 - name: log-aggregator
   image: images.my-company.example/log-aggregator:v6
   resources:
     requests:
       memory: "64Mi"
       cpu: "250m"
     limits:
       memory: "128Mi"
       cpu: "500m"
```

As you can see, resource requests and limits are specified for each container. The pod also tracks this information in totality using two fields:

Resource.requested: equal to the sum of requested resources for all of its containers Resources.limited: equal to the sum of limited resources for all of its containers.

These pod fields are used to locate which node is best for scheduling the pod. The Kubernetes control plane's algorithms compare the resource requests to the available resources on each node in the cluster, and automatically assign a node to the pod being provisioned.

CPU Resource Requests and Limits

Limits and request for CPU resources are measured in millicores. If your container needs one full core to run, you would put the value “1000m.” If your container only needs ¼ of a core, you would put a value of “250m.”

That seems straight-forward enough. But what happens if you accidentally provide a value larger than the core count of your biggest node? Requesting 2000m from a cluster of quad-core machines is no problem; but 5000m? Your pod will never be scheduled.

On the other hand, let’s say that you quickly hit the CPU resource limit after your pod is deployed onto a worker node. Since CPU is considered as a compressible resource, instead of terminating the container, Kubernetes starts throttling it which could result in a slow-running application that is difficult to diagnose.

Memory Resource Requests and Limits

Limits and requests for Memory resources are defined in bytes. Most people typically provide a mebibyte value for memory (essentially, the same thing as a megabyte), but you can also use anything from bytes to petabytes.

Similar to over-requested CPU resources, placing a memory request larger than the amount of memory on your nodes causes the pod to not get scheduled. However, unlike how a container throttles its pod when CPU limits have been met, a pod might simply terminate when its memory limit is met.

If the pod is restartable, the kubelet will restart it, but keep in mind that containers which exceed their memory requests can cause the offending pod to get evicted whenever the node runs out of memory. An eviction happens when a cluster is running out of either memory or disk, which in turn causes kubelets to reclaim resources, kill containers, and declare pods as failed until the usage rises above the eviction threshold level.

Setting Requests & Limits Via Namespaces

Kubernetes namespaces help divide a cluster into multiple virtual clusters. You can then allocate your virtual clusters to specific applications, services, or teams. This is useful when you have several engineers or engineering teams working within the same large Kubernetes cluster.

After you have defined namespaces for all of your teams, we advise establishing consistent resource requirement thresholds for each namespace to ensure no resources are needlessly reserved. You can achieve this using Resource Quotas and Limit Ranges to allocate resources to those namespaces.

Resource Quotas

Resource quotas guarantee a fixed amount of compute resources for a given virtual cluster by reserving the quoted resources for exclusive use within a single, defined namespace.

Over-commitments stemming from generously defined resource quotas are a common reason for increased infrastructure cost. However, under-committed resource quotas risk degrading your application’s performance.

You can also use resource quotas to limit the number of objects created by a given type or by its resource consumption within the namespace.

Example

Let’s create a resource quota on a number of pods.

First, we’ll specify a hard limit of 2 pods. This means one can only create 2 pods in the default namespace.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: ResourceQuota
metadata:
 name: pod-examples
spec:
 hard:
   pods: "2"
EOF
```

The next step is to run the above command in the terminal to create a resource quota named pod-examples.

Now that we’ve created the resource quota, let’s test it. Run the following command three times (each time with a different pod name) to create three different pods:

```
kubectl run pod1 --image=nginx
```

You'll notice that the request to create the 3rd and final pod will fail, displaying the error below:

```
Error from server (Forbidden): pods "pod3" is forbidden: exceeded
quota: pod-demo, requested: pods=1, used: pods=2, limited: pods=2
```

The resource quota has successfully prevented a third pod from being run.

Limit Ranges

With Resource Quotas, we saw how you can place a restriction on resource consumption based on namespaces. However, the objects that are created within the defined namespace can still utilize all of the resources available within that namespace. This can cause one object to utilize all of the available resources, essentially starving out the others.

Limit ranges overcome this problem by setting minimum and maximum limits on CPU, RAM, and storage requests per PVC within a namespace at the object level.

Example

Let's set a limit that defines the minimum and maximum memory allocation. We'll specify a maximum limit of 1GB and a minimum limit of 500MBs.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: LimitRange
metadata:
 name: min-max-memory-demo
spec:
 limits:
 - max:
     memory: 1Gi
   min:
     memory: 500Mi
   type: Container
EOF
```

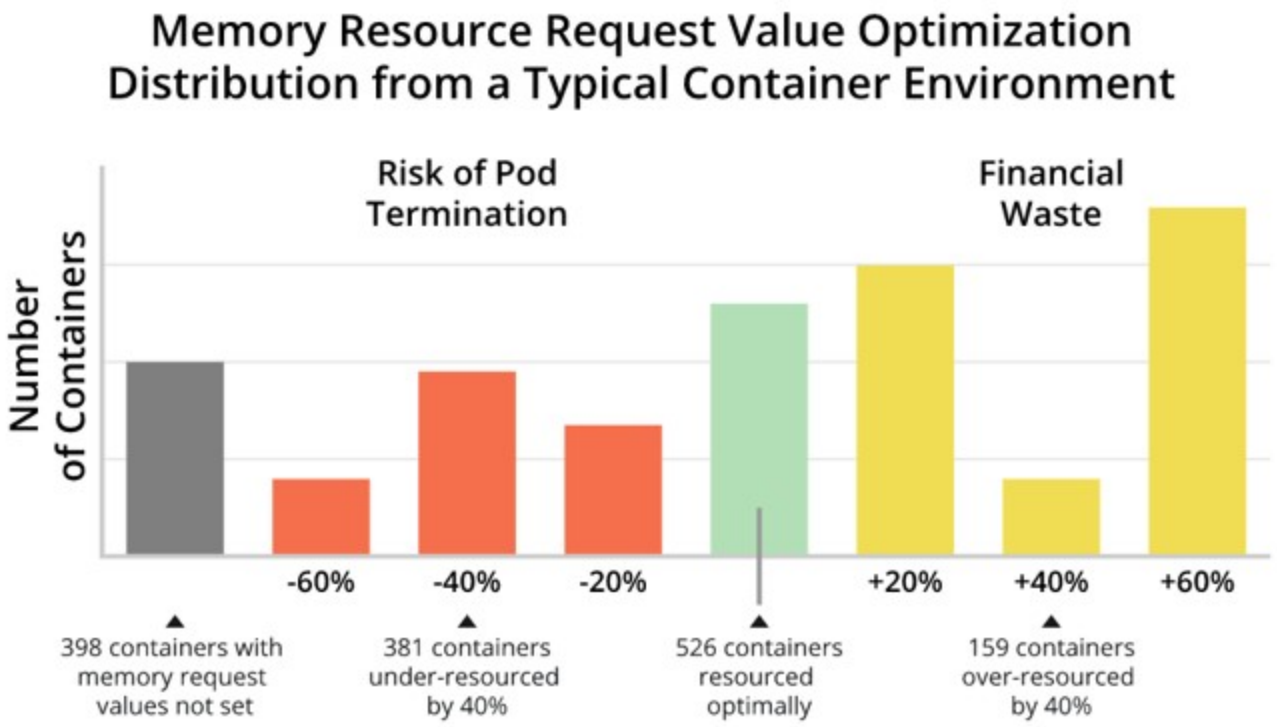
The above command creates a LimitRange named min-max-memory-demo in the default namespace.

Now, let's test the LimitRange by attempting to create a pod that possesses a container whose memory resource limit exceeds 1GB and its memory request is below 500MB.

```
cat <<EOF | kubectl apply -f -
apiVersion: v1
kind: Pod
metadata:
 name: memory-demo-pod
spec:
 containers:
 - name: memory-demo-container
   image: nginx
   resources:
     limits:
       memory: "1.5Gi"
     requests:
       memory: "100Mi"
EOF
```

When you create a pod with memory less than the min-memory or greater than the max memory, the pod creation will fail and display the following error:

```
Error from server (Forbidden): error when creating "STDIN": pods
"memory-demo-pod" is forbidden: [minimum memory usage per Container is
500Mi, but request is 100Mi, maximum memory usage per Container is 1Gi,
but limit is 1536Mi]
```



Common Challenges

Under and Overcommitments

Even though resources like CPU and memory can be allocated and restricted—either by object type, count, or namespaces—these settings can still lead to slack (a word used by K8s administrators to describe wasted spend) and overcommitments.

For example, in a scenario where insufficient memory resources are allocated, the application would be Out Of Memory (OOM) and crash. Without proper planning, the provisioning of a new pod used for testing could lead to a production application outage.

One way to solve this problem is to use pod priority and preemption. Pod priority defines the importance of a given pod relative to other pods. When a higher priority pod cannot get scheduled, the scheduler attempts to preempt (evict) a lower priority pod.

To take advantage of pod priorities, simply create a PriorityClass (a non-namespace object). Here's an example:

```
apiVersion: v1
kind: PriorityClass
metadata:
  name: critical
value: 9999999
globalDefault: false
description: "Use this class for critical service pods only."
```

You can also disable preemption of a high PriorityClass by setting the preemptionPolicy to "Never."

```
apiVersion: v1
kind: PriorityClass
metadata:
  name: high-priority
value: 7777777
globalDefault: false
preemptionPolicy: Never
description: "Use this class for critical service pods only."
```

Pod priorities are mapped back to a pod using the priorityCassName field.

```
apiVersion: v1
kind: Pod
metadata:
  name: a-pod
spec:
  containers:
  - name: a1-container
    image: nginx
    imagePullPolicy: IfNotPresent
    resources:
       limits:
         memory: “1Gi”
         cpu: “500Mi”
- name: a2-container
    image: nginx
    imagePullPolicy: IfNotPresent
    resources:
       limits:
         memory: “1Gi”
         cpu: “500Mi”
  priorityClassName: critical
```

Pod priority and preemption allows you to confidently schedule your most critical workloads without worrying about over-provisioning your clusters.

Chargeback and Utilization Reports

It’s difficult to create a chargeback or utilization report for individual application owners on a shared cluster. This is because a Kubernetes cluster may have hundreds or thousands of pods (and other resources).

One way to solve this is by using labels that represent your cost centers. Labels are key/value pairs you can assign to objects, such as pods and namespaces.

Applying labels enables:

Building a logical, hierarchical organization of objects Performing bulk operations on subsets of objects

Below are some examples of pod labels and container labels.

```
apiVersion: v1
kind: Pod
metadata:
  name: a-pod
  labels:
      env: staging
    team: infra
spec:
  containers:
  - name: a1-container
    image: nginx
    imagePullPolicy: IfNotPresent
    resources:
       limits:
         memory: “1Gi”
         cpu: “500Mi”
- name: a2-container
    image: nginx
    imagePullPolicy: IfNotPresent
    resources:
       limits:
         memory: “1Gi”
         cpu: “500Mi”
  priorityClassName: critical
```

You can take this idea a step further by using annotations on containers, however unlike labels, annotations cannot be used to select and identify objects.

Another way to solve this problem is to align your application workloads with the right computing resource pools. For example, if your cluster is on one of the major cloud platforms and has been deployed using kOps, then it supports both CloudLabels and NodeLabels. Once these labels are used in kOps, they can be applied to all the instances that are part of the Cluster. These tags on the instance or virtual machines can then be used to do cost allocation and chargeback.