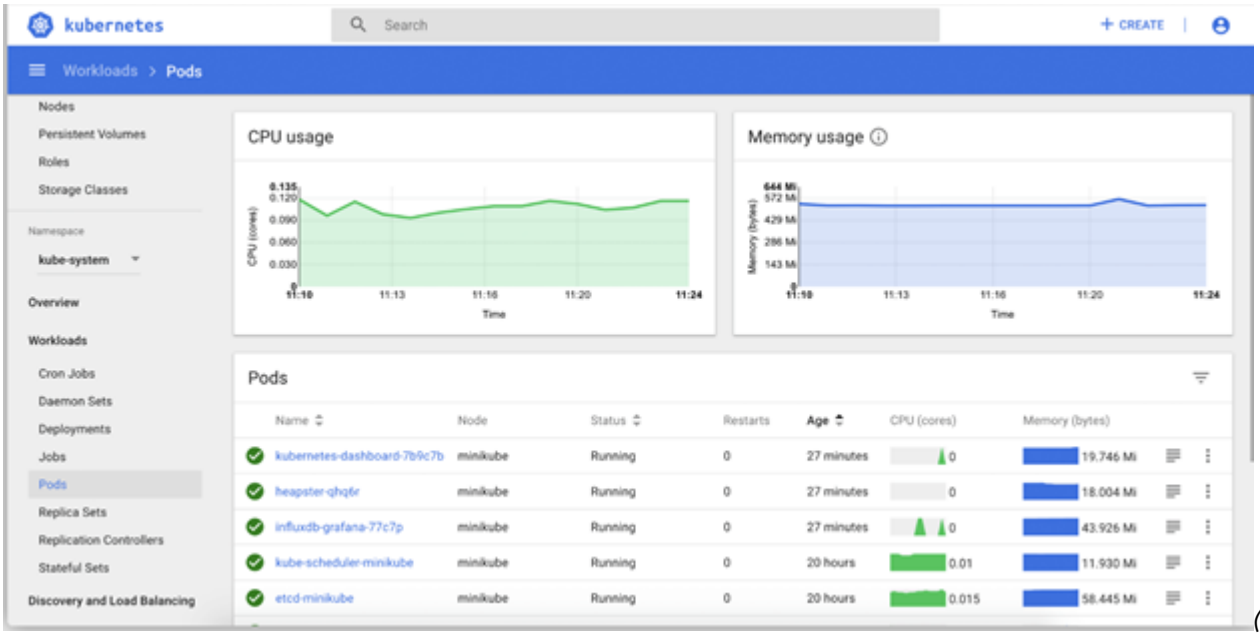
KUBERNETES COMPUTE RESOURCES EXPLAINED

In the Kubernetes world, there are two types of resources:

- **Compute resources**, such as CPU (units) and memory (bytes), which can be measured.
- **API resources**, such as pods, services, etc., which are simply K8s objects.

In this article, I'll explain compute resources.



(This article is part of our Kubernetes Guide. Use the right-hand menu to navigate.)

Quality of Service (QoS)

By default, when you deploy a pod, the pod containers essentially consume whatever compute resources are available on that worker node instance. That's great—we don't have to worry about anything, then, right?

Well, not exactly.

What happens when one of the containers consumes more CPU and memory resources than others? What happens to the other pod containers running on the node? There is a high chance that some of the containers may fail due to lack of resources; if we are running an application with a poor codebase, that can be impactful to the business.

To solve this issue, Kubernetes has resource requests and limits which help control the amount of resources a pod container consumes, ensuring the highest level of quality of service (QoS). It is important to point out that QoS (requests and limits) can also apply to namespaces, not just pods. That way if you have multiple dev teams deploying to a cluster, you can control resource consumption depending on the priority of the service.

How resource request and limit works

Fundamentally, a worker node or instance only allocate a certain portion of its overall compute resource for pods to use so, that portion is what is shared between all pods scheduled on that node. Kubernetes scheduler makes sure that the sum of all the compute resource of containers running on the instance does not exceed the overall allocated resource for pods. It is important to know the difference between request and limit.

- **Request** is how much resource Kubernetes will guarantee a pod.
- **Limit** is the maximum amount of resource Kubernetes will allow a pod to use.

Keep in mind that pod scheduling is based on request, not limit, but a pod and its containers are not allowed to exceed the limit specified. When you deploy a pod, a capacity check determines whether the total amount of compute resource requested is not more than what the node can allocate.

How request and limits are enforced

How the request and limit are enforced is based on whether the resource is compressible or incompressible. A compressible resource can be throttled, but an impressible resource—not so much. For example, CPU is considered compressible and memory is incompressible.

Pods are guaranteed the amount of CPU requested; they may or may not get more CPU depending on other pods running and anything past the limit is throttled. Pod memory request are also guaranteed but when pods exceed memory limit, the process inside the container that is using the most memory will be killed. The pod will not be terminated but the process inside the container using the memory will be terminated.

QoS Classes

There are times when a node is completely out of compute resources; the sum of all the running containers compute resources is greater than the compute resource on the node itself. In extreme cases like this, we have to start killing running containers on the node. Ideally, we first remove less critical containers to free up resources. So, how do we tell Kubernetes which pod needs to go first in the case when all resources are exhausted on the node? By using QoS classes.

We can designate three types of classes—best effort, guaranteed, and burstable—to tell the system which pods to terminate first.

- **Best effort.** Pods with no request or limits, which are terminated first in case of resource exhaustion.

```
Containers:
  name: foo
    Resources:
  name: bar
    resources:
```

- **Guaranteed.** Pods where both limit and (optionally) request are set for all resources (CPU and memory) and their values are the same. These pods are high priority, therefore they are terminated only if they are over the limit and there are no lower priority pods to terminate.

```
containers:
  name: pong
    resources:
      limits:
        cpu: 100m
        memory: 100Mi
      requests:
        cpu: 100m
        memory: 100Mi
```

- **Burstable.** Pods where both request and (optionally) limits are set for one or more resources (CPU and/or memory) and their values are NOT the same. These pods are terminated when there are no best effort pods to terminate.

```
containers:
  name: foo
    resources:
      limits:
        memory: 1Gi

  name: bar
    resources:
      limits:
        cpu: 100m
```

Because application uptime is part of business SLAs, it is important to apply adequate resources to

these applications. Understanding how to properly plan these resources is crucial to business. Using requests and limits will help team properly allocate resources to critical applications and QoS will help control the lifecycle of the application in case of resource exhaustion on the worker node. Teams can also define these requests and limits at the namespace level, that way resources can be carved based on dev teams.