## Exploring pod autoscaling lead time

The time it takes to create a new Pod on a new Node is determined by four major factors:

1. ****Horizontal Pod Autoscaler reaction time.****
2. ****Cluster Autoscaler reaction time.****
3. ****Node provisioning time.****
4. ****Pod creation time.****

By default, **[pods' CPU and memory usage is scraped by kubelet every 10 seconds.](https://github.com/kubernetes/kubernetes/blob/2da8d1c18fb9406bd8bb9a51da58d5f8108cb8f7/pkg/kubelet/kubelet.go" \l "L1855" \t "https://learnk8s.io/_blank)**

**[Every minute, the Metrics Server will aggregate those metrics](https://github.com/kubernetes-sigs/metrics-server/blob/master/FAQ.md" \l "how-often-metrics-are-scraped" \t "https://learnk8s.io/_blank)** and expose them to the rest of the Kubernetes API.

The Horizontal Pod Autoscaler controller is in charge of checking the metrics and deciding to scale up or down your replicas.

By default, the **[Horizontal Pod Autoscaler checks Pods metrics every 15 seconds.](https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/" \l "how-does-the-horizontal-pod-autoscaler-work" \t "https://learnk8s.io/_blank)**

In the worst-case scenario, the Horizontal Pod Autoscaler can take up to 1 minute and a half to trigger the autoscaling (i.e. 10s + 60s + 15s).



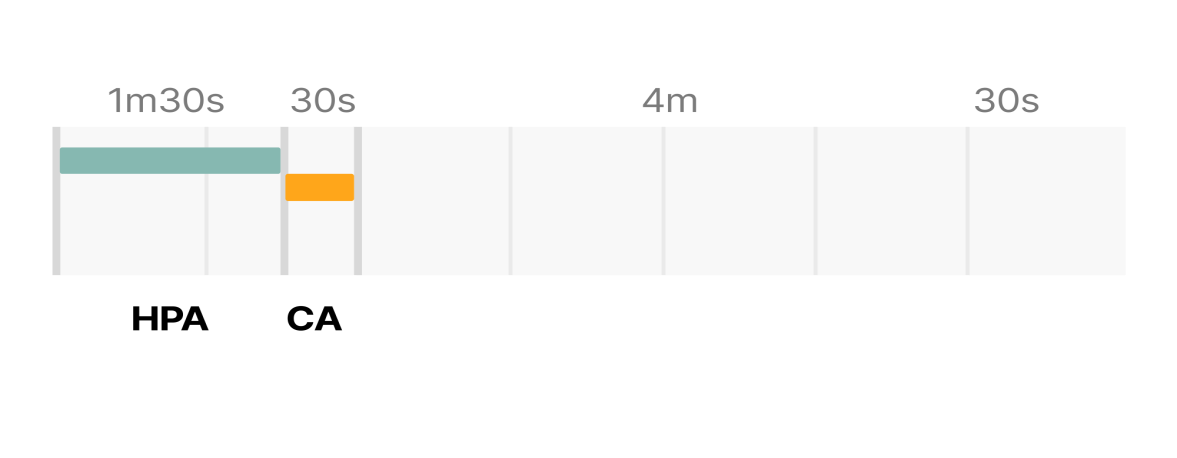
**[The Cluster Autoscaler checks for unschedulable Pods in the cluster every 10 seconds.](https://github.com/kubernetes/autoscaler/blob/master/cluster-autoscaler/FAQ.md" \l "how-does-scale-up-work" \t "https://learnk8s.io/_blank)**

Once one or more Pods are detected, it will run an algorithm to decide:

1. ****How many nodes**** are necessary to deploy all pending Pods.
2. ****What type of node group**** should be created.

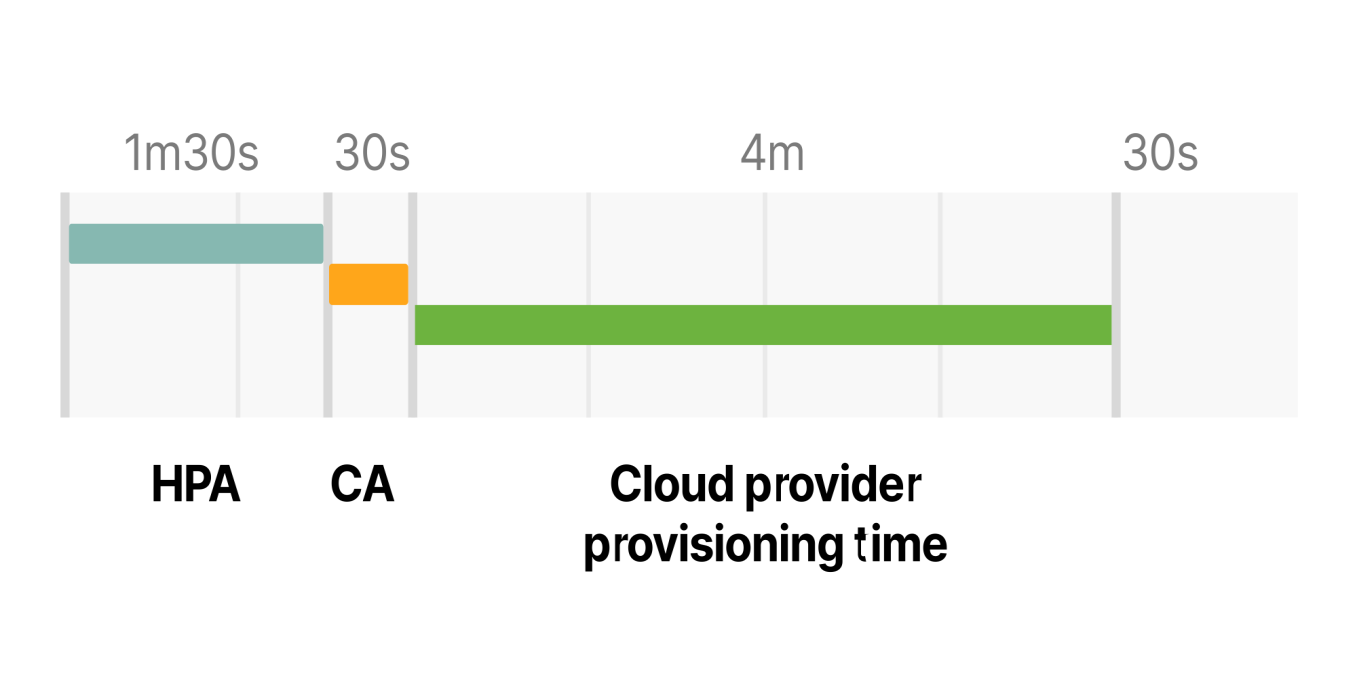
The entire process should take:

* ****No more than 30 seconds**** on clusters with ****less than 100 nodes**** with up to 30 pods each. The average latency should be about 5 seconds.
* ****No more than 60 seconds**** on cluster with ****100 to 1000 nodes****. The average latency should be about 15 seconds.



Then, there's the Node provisioning time, which depends mainly on the cloud provider.

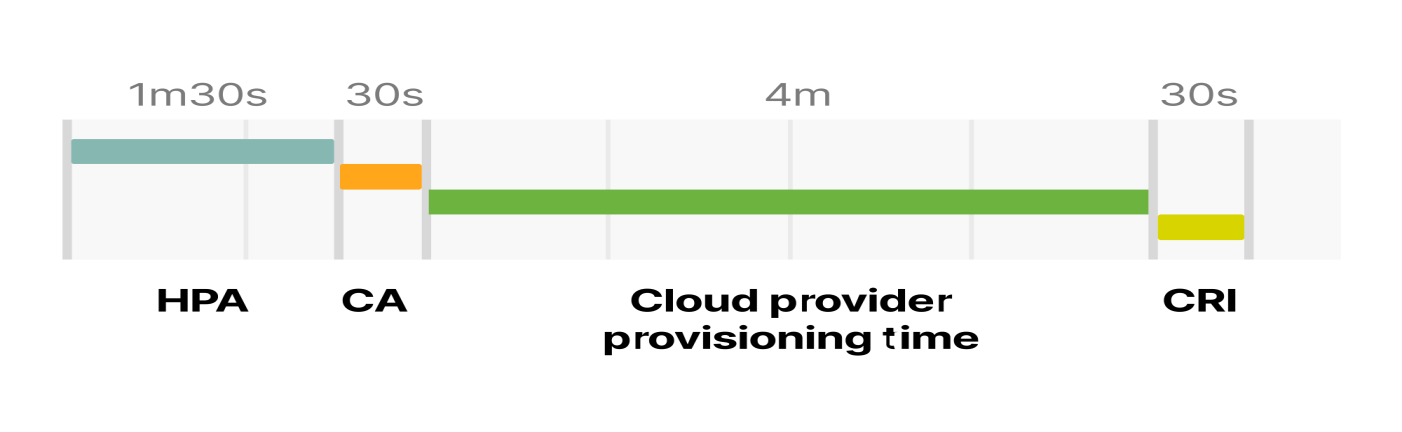
****It's pretty standard for a new compute resource to be provisioned in 3 to 5 minutes.****



Lastly, the Pod has to be created by the container runtime.

Launching a container shouldn't take more than few milliseconds, but ****downloading the container image could take several seconds.****

If you're not caching your container images, downloading an image from the container registry could take from a couple of seconds up to a minute, depending on the size and number of layers.



So the total timing for trigger the autoscaling when there is no space in the current cluster is:

1. The Horizontal Pod Autoscaler might take up to 1m30s to increase the number of replicas.
2. The Cluster Autoscaler should take less than 30 seconds for a cluster with less than 100 nodes and less than a minute for a cluster with more than 100 nodes.
3. The cloud provider might take 3 to 5 minutes to create the computer resource.
4. The container runtime could take up to 30 seconds to download the container image.

In the worse case, with a small cluster, you have:

HPA delay: 1m30s +

CA delay: 0m30s +

Cloud provider: 4m +

Container runtime: 0m30s +

=========================

Total 6m30s

With a cluster with more than 100 nodes, the total delay could be up to 7 minutes.

*Are you happy to wait for 7 minutes before you have more Pods to handle a sudden surge in traffic?*

*How can you tune the autoscaling to reduce the 7 minutes scaling time if you need a new node?*

You could change:

* The refresh time for the Horizontal Pod Autoscaler (controlled by the --horizontal-pod-autoscaler-sync-period flag, default is 15 seconds).
* The interval for metrics scraping in the Metrics Server (controlled by the metric-resolution flag, default 60 seconds).
* How frequently the cluster autoscaler scans for unscheduled Pods (controlled by the scan-interval flag, default 10 seconds).
* How you cache the image on the local node ([with a tool such as kube-fledged](https://github.com/senthilrch/kube-fledged" \t "https://learnk8s.io/_blank)).

But even if you were to tune those settings to a tiny number, you will still be limited by the cloud provider provisioning time.

*So, how could you fix that?*

Since you can't change the provisioning time, you will need a workaround this time.

You could try two things:

1. ****Avoid creating new nodes,**** if possible.
2. ****Creating nodes proactively**** so that they are already provisioned when you need them.

*Let's have a look at the options one at a time.*

## Choosing the optimal instance size for a Kubernetes node

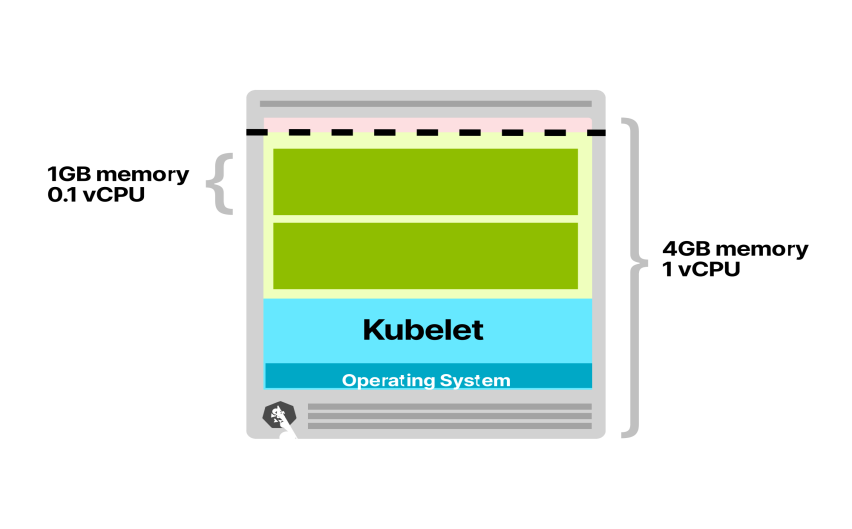
****Choosing the right instance type for your cluster has dramatic consequences on your scaling strategy.****

*Consider the following scenario.*

You have an application that requests 1GB of memory and 0.1 vCPU.

You provision a node that has 4GB of memory and 1 vCPU.

After reserving memory and CPU for the kubelet, operating system and eviction threshold, you are left with ~2.5GB of memory and 0.7 vCPU that can be used for running Pods.



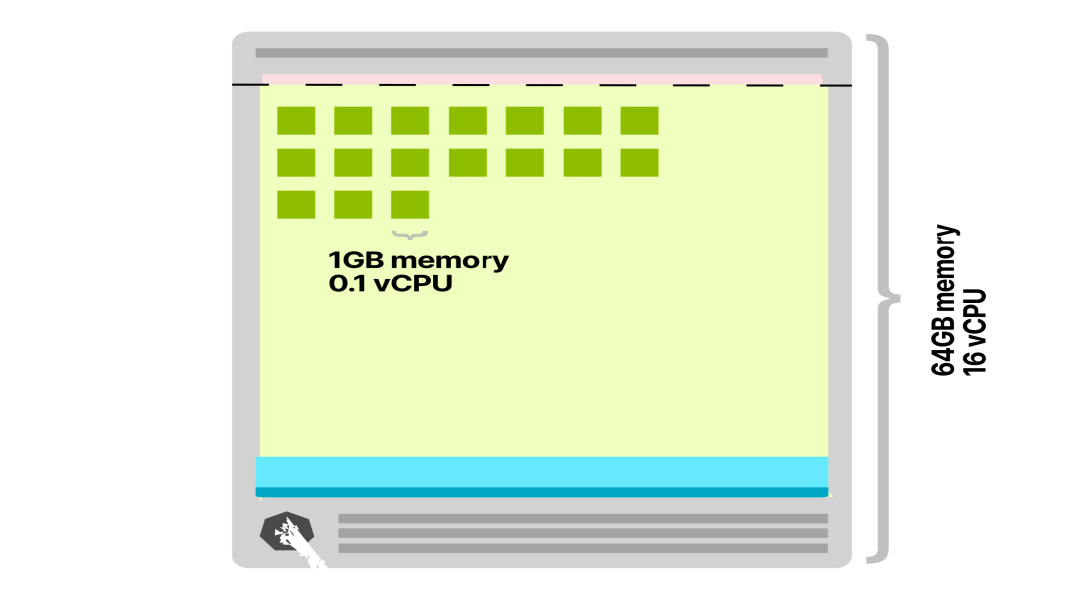
Your node has space for only two Pods.

Every time you scale your replicas, you are likely to incur in up to 7 minutes delay (the lead time to trigger the Horizontal Pod Autoscaler, Cluster Autoscaler and provisioning the compute resource on the cloud provider).

*Let's have a look at what happens if you decide to use a 64GB memory and 16 vCPU node instead.*

After reserving memory and CPU for the kubelet, operating system and eviction threshold, you are left with ~58.32GB of memory and 15.8 vCPU that can be used for running Pods.

****The available space can host 58 Pods, and you are likely to need a new node only when you have more than 58 replicas.****



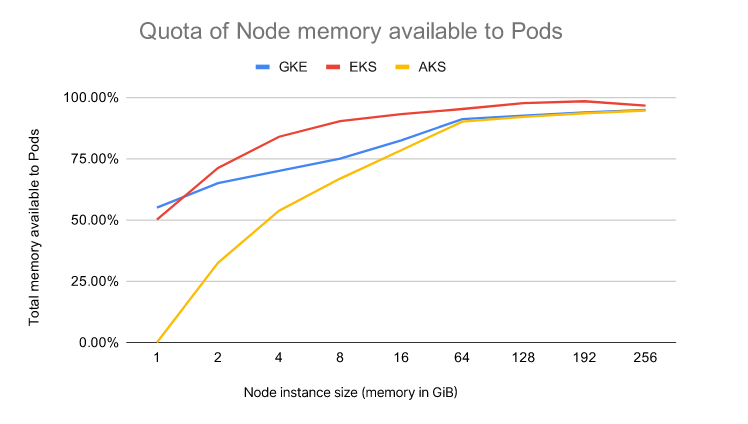
Also, every time a node is added to the cluster, several pods can be deployed.

There is less chance to trigger *again* the Cluster Autoscaler (and provisioning new compute units on the cloud provider).

Choosing large instance types also has another benefit.

****The ratio between resource reserved for Kubelet, operating system and eviction threshold and available resources to run Pods is greater.****

Have a look at this graph that pictures the memory available to pods.



As the instance size increase, you can notice that (in proportion) the resources available to pods increase.

In other words, you are utilising your resources more efficiently than having two instances of half of the size.

*Should you select the biggest instance all the time?*

****There's a peak in efficiency dictated by how many Pods you can have on the node.****

Some cloud providers limit the number of Pods to 110 (i.e. GKE). Others have limits dictated by the underlying network on a per-instance basis (i.e. AWS).

[You can inspect the limits from most cloud providers here.](https://docs.google.com/spreadsheets/d/1RPpyDOLFmcgxMCpABDzrsBYWpPYCIBuvAoUQLwOGoQw/edit" \l "gid=907731238" \t "https://learnk8s.io/_blank)

****And choosing a larger instance type is not always a good option.****

You should also consider:

1. ****Blast radius**** — if you have only a few nodes, then the impact of a failing node is bigger than if you have many nodes.
2. ****Autoscaling is less cost-effective**** as the next increment is a (very) large node.

Assuming you have selected the right instance type for your cluster, you might still face a delay in provisioning the new compute unit.

*How can you work around that?*

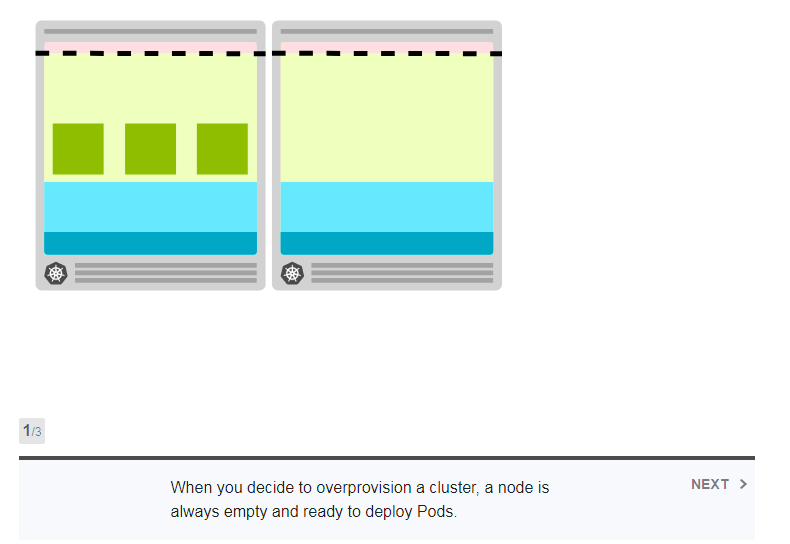
*What if instead of creating a new node when it's time to scale, you create the same node ahead of time?*

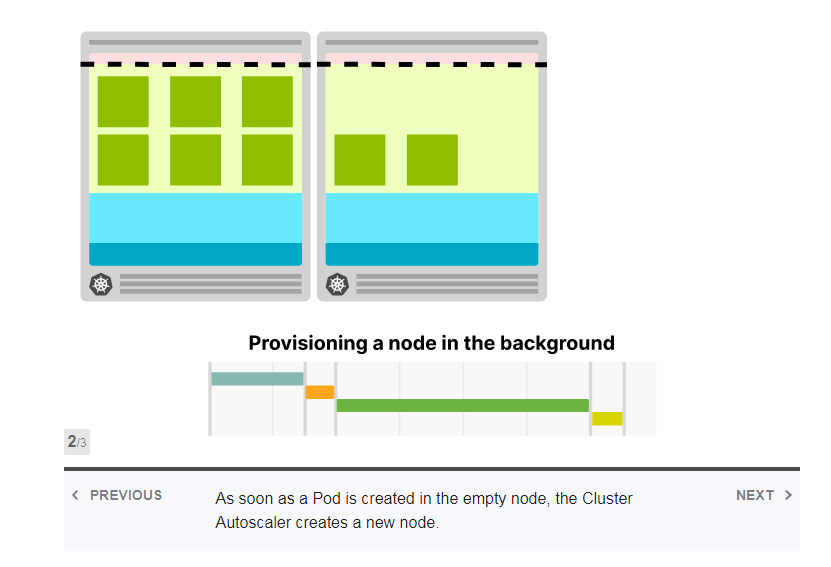
## Overprovisioning nodes in your Kubernetes cluster

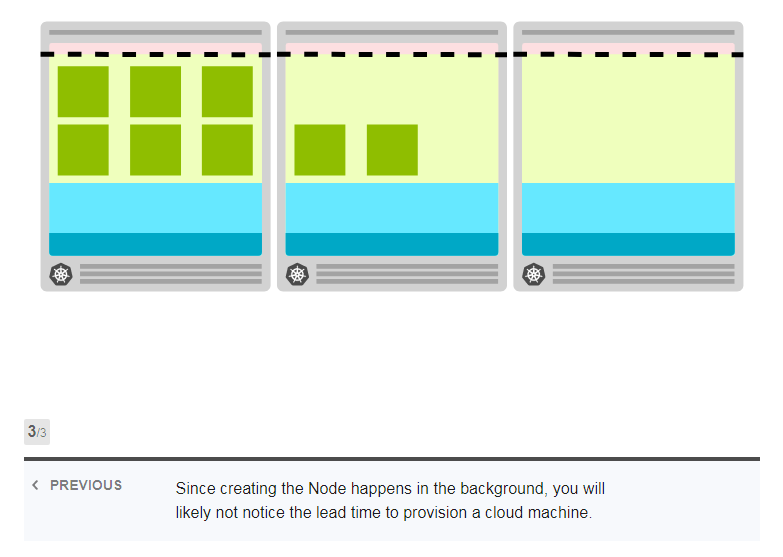
If you can afford to have a spare node available at all times, you could:

1. Create a node and leave it empty.
2. As soon as there's a Pod in the empty node, you create another empty node.

****In other words, you teach the autoscaler always to have a spare empty node if you need to scale.****







****It's a trade-off: you incur an extra cost (one empty compute unit available at all times), but you gain in speed.****

With this strategy, you can scale your fleet much quicker.

*But there's bad and good news.*

The bad news is that the Cluster Autoscaler doesn't have this functionality built-in.

****It cannot be configured to be proactive, and there is no flag to "always provision an empty node".****

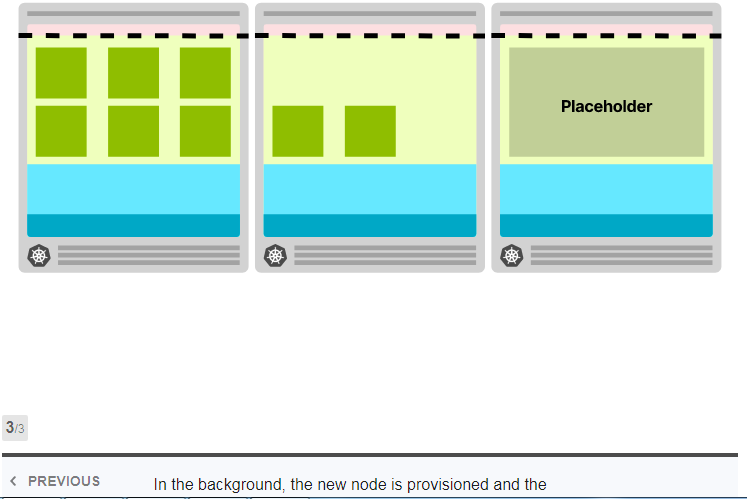
The good news is that you can still fake it.

*Let me explain.*

****You could run a Deployment with enough requests to reserve an entire node.****

You could think about this pod as a placeholder — it is meant to reserve space, not use any resource.

As soon as a real Pod is created, you could evict the placeholder and deploy the Pod.



Notice how this time, you still have to wait 5 minutes for the node to be added to the cluster, but you can keep using the current node.

In the meantime, a new node is provisioned in the background.

How to achieve that?

****Overprovisioning can be configured using deployment running a pod that sleeps forever.****

overprovision.yaml

apiVersion: apps/v1kind: Deploymentmetadata:

name: overprovisioningspec:

replicas: 1

selector:

matchLabels:

run: overprovisioning

template:

metadata:

labels:

run: overprovisioning

spec:

containers:

- name: pause

image: k8s.gcr.io/pause

resources:

requests:

cpu: '1739m'

memory: '5.9G'

****You should pay extra attention to the memory and CPU requests.****

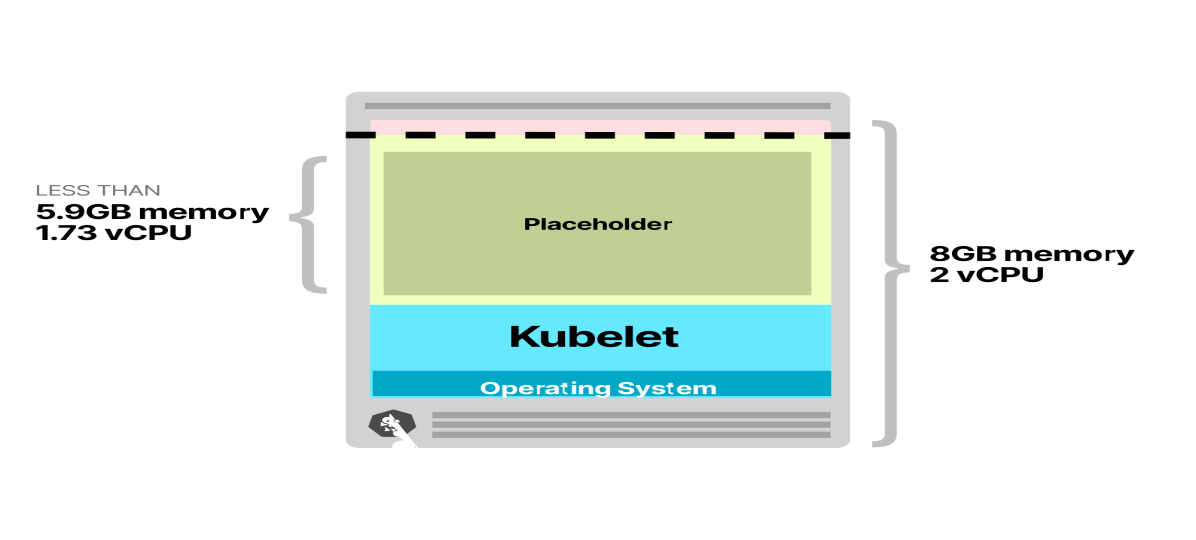
The scheduler uses those values to decide where to deploy a Pod.

In this particular case, they are used to reserve the space.

You could provision a single large pod that has roughly the requests matching the available node resources.

****Please make sure that you account for resources consumed by the kubelet, operating system, kube-proxy, etc.****

If your node instance is 2 vCPU and 8GB of memory and the available space for pods is 1.73 vCPU and ~5.9GB of memory, your pause pod should match the latter.



To make sure that the Pod is evicted as soon as a real Pod is created, you can use [Priorities and Preemptions.](https://kubernetes.io/docs/concepts/scheduling-eviction/pod-priority-preemption/" \t "https://learnk8s.io/_blank)

****Pod Priority indicates the importance of a Pod relative to other Pods.****

When a Pod cannot be scheduled, the scheduler tries to preempt (evict) lower priority Pods to schedule the Pending pod.

You can configure Pod Priorities in your cluster with a PodPriorityClass:

priority.yaml

apiVersion: scheduling.k8s.io/v1

kind: PriorityClassmetadata:

name: overprovisioningvalue: -1

globalDefault: **false**

description: 'Priority class used by overprovisioning.'

Since the default priority for a Pod is 0 and the overprovisioning PriorityClass has a value of -1, those Pods are the first to be evicted when the cluster runs out of space.

PriorityClass also has two optional fields: globalDefault and description.

* The description is a human-readable memo of what the PriorityClass is about.
* The globalDefault field indicates that the value of this PriorityClass should be used for Pods without a priorityClassName. Only one PriorityClass with globalDefault set to true can exist in the system.

You can assign the priority to your sleep Pod with:

overprovision.yaml

apiVersion: apps/v1kind: Deploymentmetadata:

name: overprovisioningspec:

replicas: 1

selector:

matchLabels:

run: overprovisioning

template:

metadata:

labels:

run: overprovisioning

spec:

priorityClassName: overprovisioning

containers:

- name: reserve-resources

image: k8s.gcr.io/pause

resources:

requests:

cpu: '1739m'

memory: '5.9G'

The setup is complete!

When there are not enough resources in the cluster, the pause pod is preempted, and new pods take their place.

Since the pause pod become unschedulable, it forces the Cluster Autoscaler to add more nodes to the cluster.

Now that you're ready to overprovision your cluster, it's worth having a look at optimising your applications for scaling.