
resource
management

Resource management – overview

- To properly manage your Kubernetes clusters, it's important to carefully manage resource usage.
- There are two resource types in Kubernetes: CPU and memory
  - CPU is expressed in units of a CPU core. Either a decimal or with the xxxm format. For example, 0.1 and 100m mean the same thing, or ten percent. The second format is referred to as "millicores".
  - Memory is express in bytes, kilobytes, megabytes, etc. The notation is xxMi, xxGi, etc. Note that the units aren't exactly megabyte, gigabyte, etc. They're actually mebibyte, gibibyte, etc. But they're so close that it's not worth worrying about.
- Resources are specified on containers, not pods. The resources for a pod is simply the sum of all containers specified for it.

Resource management – requests

- There are two ways Kubernetes controls resources such as CPU and memory:
  - Requests – This is the value the container is guaranteed to get when it's pod is scheduled. If the scheduler can't find a node with this amount, then the pod wont get scheduled.
  - Limits – This it the limit placed on the CPU or memory. The container will never use more than this.
- Requests can never be higher than limits. Kubernetes will throw an error.
- These values are assigned to containers, not pods.

Resource management – requests

Total for pod
Total CPU request: 300 millicore
Total memory request: 96 Mi

Total CPU limit: 900 millicore
Total memory limit: 192 Mi

```
containers:
  - name: container1
    image: myimage:v1
    resources:
      requests:
        memory: "64Mi"
        cpu: "200m"
      limits:
        memory: "128Mi"
        cpu: "600m"
  - name: container2
    image: myotherimage:v1
    resources:
      requests:
        memory: "32Mi"
        cpu: "100m"
      limits:
        memory: "64Mi"
        cpu: "300m"
```

Resource management – resource limits

- What happens when you don’t specify a CPU limit?
  - The container has no upper bound, and could use all of the available CPU available on it’s node.
  - If the namespace has a default limit, then it will inherit that.

Resource management – namespace settings

- In an ideal world, everyone will behave and set their limits appropriately.
- But in reality, people forget to put proper limits, or make limits that are too high. Engineers tend to overprovision because it's easier than managing things. They may set requests and limits that are higher than their fair share and waste node capacity.
- To prevent this, you can set ResourceQuotas and LimitRanges at the namespace level.

Resource management – resourcequota

- ResourceQuotas set limits for all containers **in the namespace**. Note this is not a per node quota.
- You can use this to keep a team in check for example. If you have several teams using the same Kubernetes cluster, then this prevents any one team from consuming too many CPU and memory units.
- Requests.cpu - this is the total requests for CPU for all containers in the namespace.
- Requests.memory - this is the total requests for memory for all containers in the namespace.
- Limits.cpu - the total limits for all containers in the namespace
- Limits.memory - the total limits for all containers in the namespace

```
apiVersion: v1
kind: ResourceQuota
metadata:
  name: myquota
spec:
  hard:
    requests.cpu: 500m
    requests.memory: 256Mi
    limits.cpu: 1000m
    limits.memory: 1024Mi
```

Resource management – limitrange

- Unlike a quota, a LimitRange applies to any container rather than the total for all containers in a namespace.
- This can prevent people from creating super small or super large containers. You can set the minimum and maximum values to ensure that all containers are given reasonable limits.
- `Default` sets default CPU and memory limit settings if they aren't specified for a container in a pod.
- `defaultRequest` sets the default requests for CPU and memory if they aren't specified for a container in a pod.
- `min` and `max` set the limits on an individual container.

```
apiVersion: v1
kind: LimitRange
metadata:
  name: mylimitrange
spec:
  limits:
    - default:
      cpu: 500m
      memory: 256Mi
    - defaultRequest
      cpu: 250m
      memory: 256Mi
    - min:
      cpu: 256m
      memory: 128Mi
    - max:
      cpu: 1000m
      memory: 1Gi
    type: Container
```

Resource management – how pods are scheduled

- At the time a pod is scheduled, the scheduler runs checks on nodes to determine if enough capacity exists. CPU and memory is finite on a node, and that capacity is used up, or “reserved” by the existing pods.
- While actual CPU and memory usage will be lower, the scheduler uses the maximum specified to give room for load later.
- If there is less resource available on a node than the pod requests, then the placement is refused and Kubernetes looks at the next node.
- If Kubernetes can’t find a suitable home for a pod, then the pod goes into a “pending” state.
- When Kubernetes places a pod on a node, these resource specifications are handed off to the container runtime (Docker run command).

Scaling in

kubernetes

Scaling in kubernetes – overview

There are two main considerations with scaling to meet demand in Kubernetes:

1. We need pods to scale either horizontally or vertically to handle increases or decreases in demand.
2. We need our cluster to scale in or out to provide the necessary compute power to hold more pods.

We have tools at our disposal to handle both of these concerns.

Scaling in kubernetes – overview

Let's get some terms straight.

Horizontal scaling – adding or removing units of work
When we say scale in or out, we're talking about horizontal scaling, or adding/removing copies of a resource.

Vertical scaling – making units of work bigger or smaller
When we say scale up or down, we're talking about vertical scaling.

How aggressive should scaling events be? In general, we want to scale out quickly to meet increased demand and then scale back in slowly.

Metrics – overview

- Kubernetes collects and manages information about resources in your cluster, such as containers, pods, services, and your overall cluster.
- Kubernetes makes this detailed information available to you so that you can evaluate the performance of your applications and make decisions.
- Metrics are collected by the metrics-server and exposed via an API.

Metrics – overview

- In Kubernetes, monitoring does not depend on a single monitoring solution. Instead, you can use two separate pipelines:
  - Resource metrics pipeline – provides basic metrics about the resources in your cluster. Metrics-server discovers all nodes in your cluster and then queries each node's kubelet for CPU and memory usage. It does this on a preset interval.
  - A full metrics pipeline – this is third party plugins such as Prometheus that provide rich metrics information and then feeds the information back to Kubernetes via an adapter which is then made available via the API.
- Scalers operate based on metrics provided by the Kubernetes Metrics API.
- Let's set up metrics collection so that scaling can work.

Metrics – metrics-server setup

- NOTE: Metrics-server replaced Heapster in v1.9+. You'll likely want to upgrade Kubernetes to the latest version and ALSO update kubectl on your machine, or you'll get strange results.
- Let's install metrics-server, available on Github.
- Clone the repo: https://github.com/kubernetes-incubator/metrics-server
- This contains resources in the deploy folder. We need to choose the right one based on our Kubernetes cluster version.
- Let's make a change to the deployment to ignore certificate issues for now. In the deployment file, add the following snippet to the container definition:

```
command:
  - /metrics-server
  - --kubelet-insecure-tls
```

- Then apply all of the files. Ie kubectl apply –f deploy/1.8+
- Resources will be placed in the kube-system namespace.
- You can view the pods and deployments, and look at the log output for the pod to verify that it is running.

Metrics – metrics-server

- Once installed, let's check metrics:
  - Kubectl top nodes
  - Kubectl top pods
- If you have watch installed, you can watch this change with watch kubectl top pods

Horizontal pod

autoscaler

HPA – overview

- A HorizontalPodAutoscaler is a separate resource type in Kubernetes.
- You point it at, or target a deployment by name.
- You give the HPA a target CPU or memory to maintain.
- It periodically checks metrics, and when the average CPU or memory goes too high, it tells Kubernetes to increase the replicas of the target deployment.

HPA – overview

- CPU and memory value is determined as an average of your pods.
- For HPA to work, you must provide resource limits to your deployments.
- HPA can scale using more detailed custom metrics. This requires a lot more setup to get more metrics available to the metrics API.
- You provide a minimum and maximum pod count to your HPA configuration. These override what's set up in the ReplicaSet or Deployment.

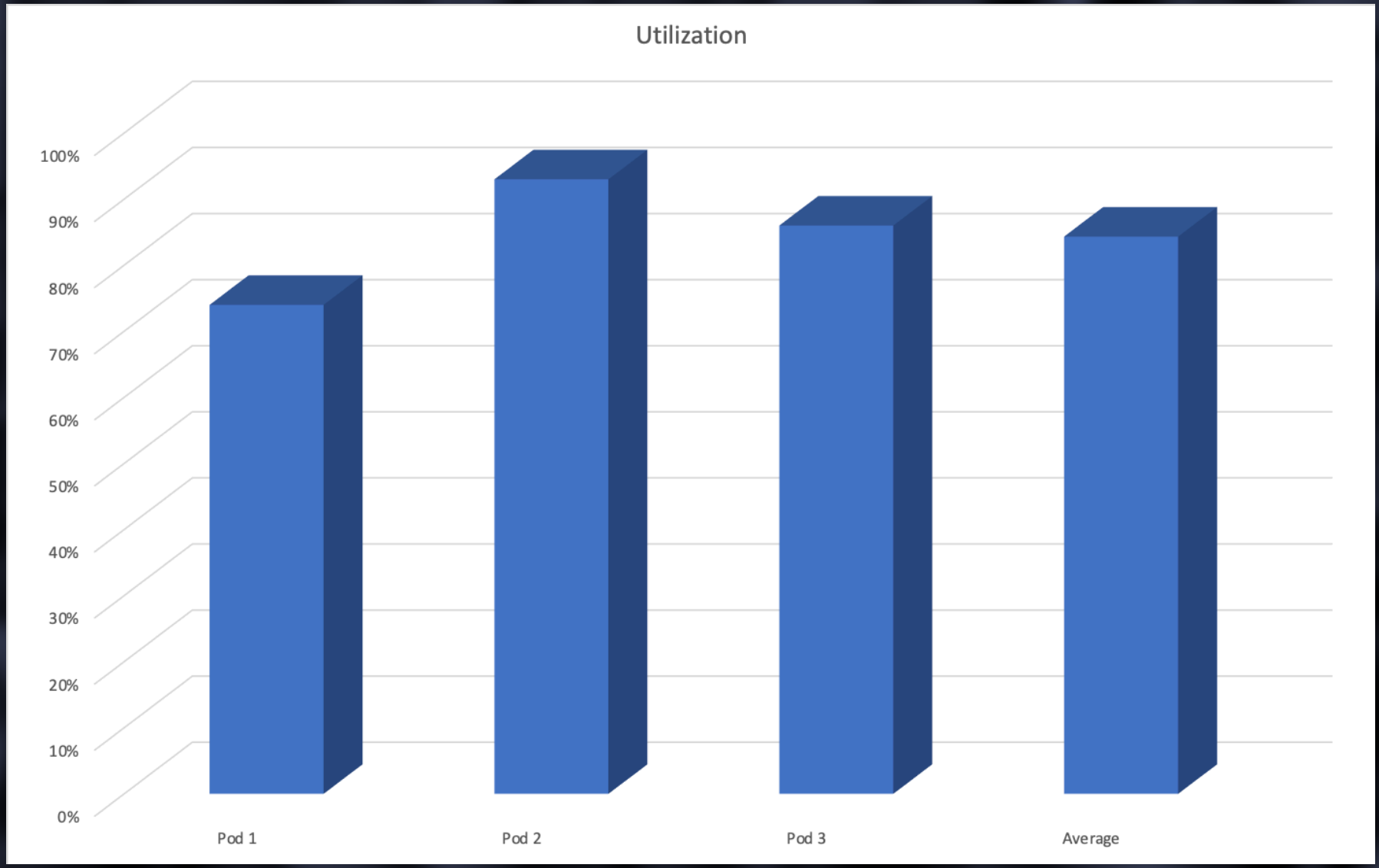
HPA – scaling decisions

- Kubernetes checks on our metrics periodically (default 30 seconds)
- As long as the cooldown delay has passed, it can make a scaling operation.
- But it must first decide how many pods to add or remove to achieve your target values.
- Let’s look at a couple examples.

HPA – scaling decisions

Target CPU utilization: 50%

Current utilization: 84%



HPA – scaling decisions

Target CPU utilization: 50%

Current utilization: 84%

How many pods need to be added to our cluster to bring the average CPU utilization down to 50%?

Desired pods = ceil(Current pods * (Current value / target value))

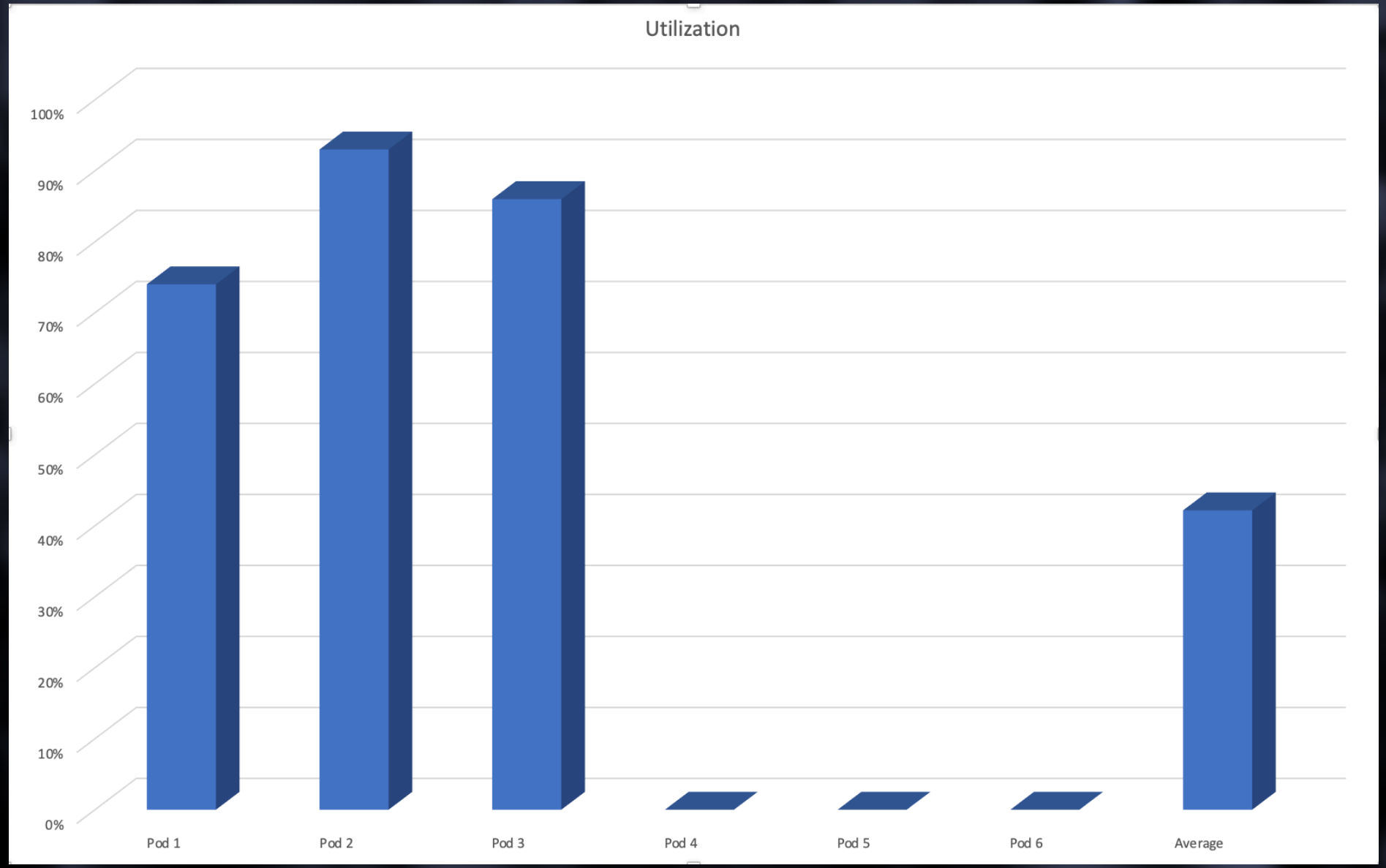
Desired pods = ceil(3 * (.84 / .5)) = 6

HPA will add 3 pods to our cluster

HPA – scaling decisions

Target CPU utilization: 50%

Current utilization: 42%



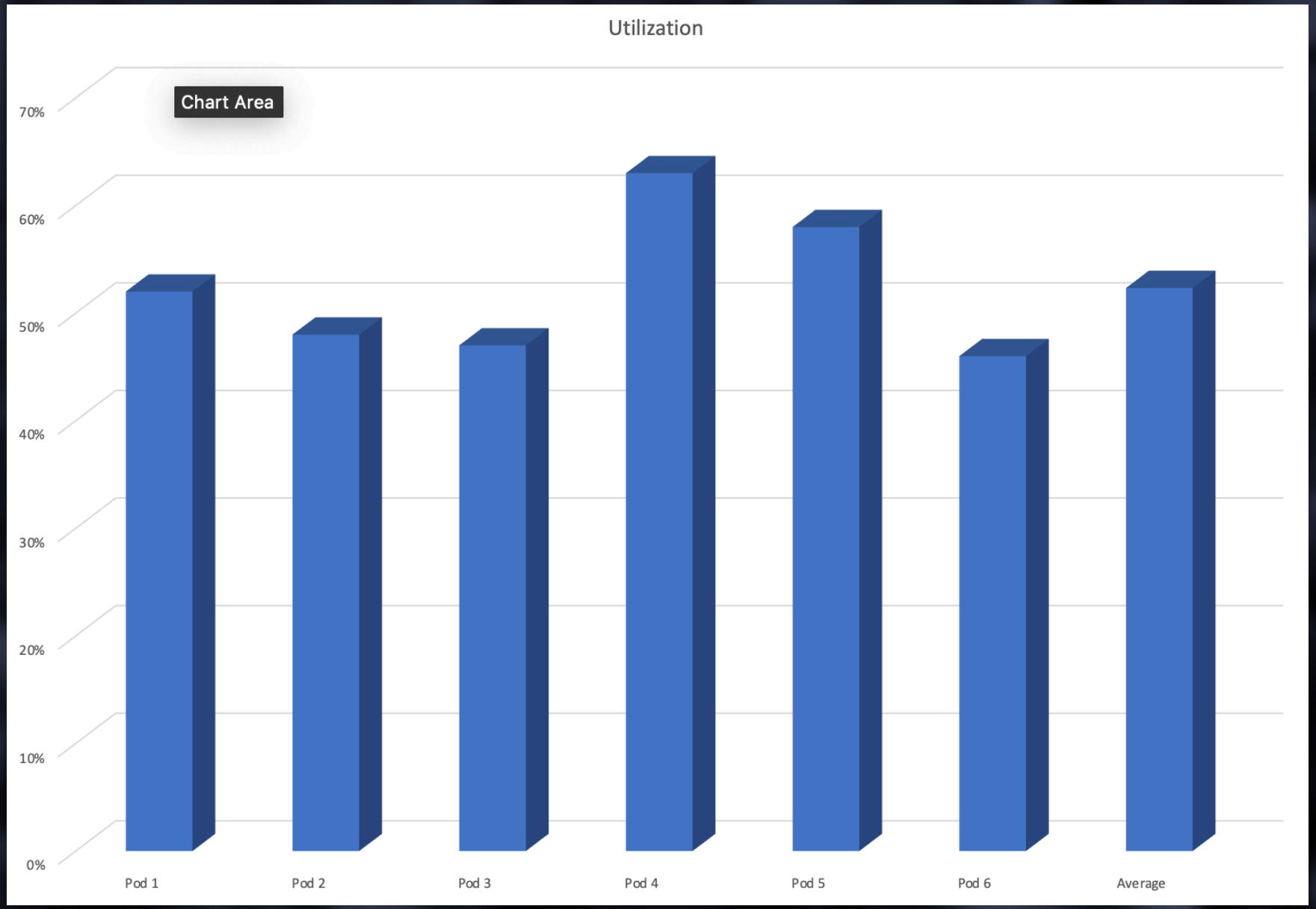
HPA – scaling decisions

- Now we've gone from 3 pods in our deployment to 6 pods.
- Things are looking good. No alarm bells going off!
- Traffic remains stable, and our pods start to even out. Our pods aren't over taxed and are right around the target values. Excellent!

HPA – scaling decisions

Target CPU utilization: 50%

Current utilization: 52%



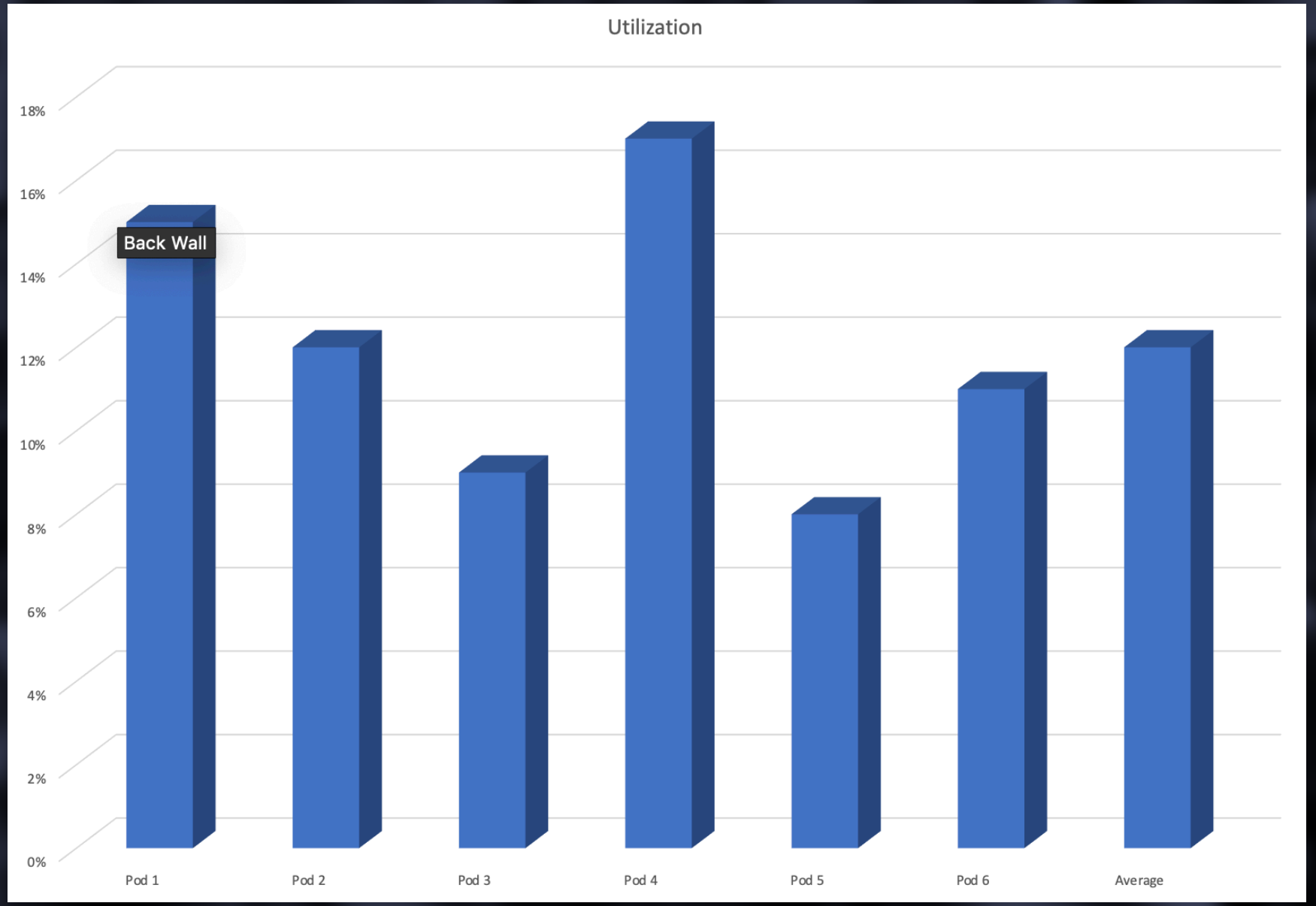
HPA – scaling decisions

- The sun is setting among our customer base and traffic is starting to die down. Our users are auto workers in Detroit and the shift just ended. Traffic suddenly dies off.
- Now we have 6 pods that are hardly doing any work! Hey Kubernetes, time to let go of some pods.

HPA – scaling decisions

Target CPU utilization: 50%

Current utilization: 12%



HPA – scaling decisions

Target CPU utilization: 50%

Current utilization: 12%

How many pods need to be removed from our cluster to bring the average CPU utilization up to 50%?

Desired pods = ceil(Current pods * (Current value / target value))

Desired pods = ceil(6 * (.12 / .5)) = 2

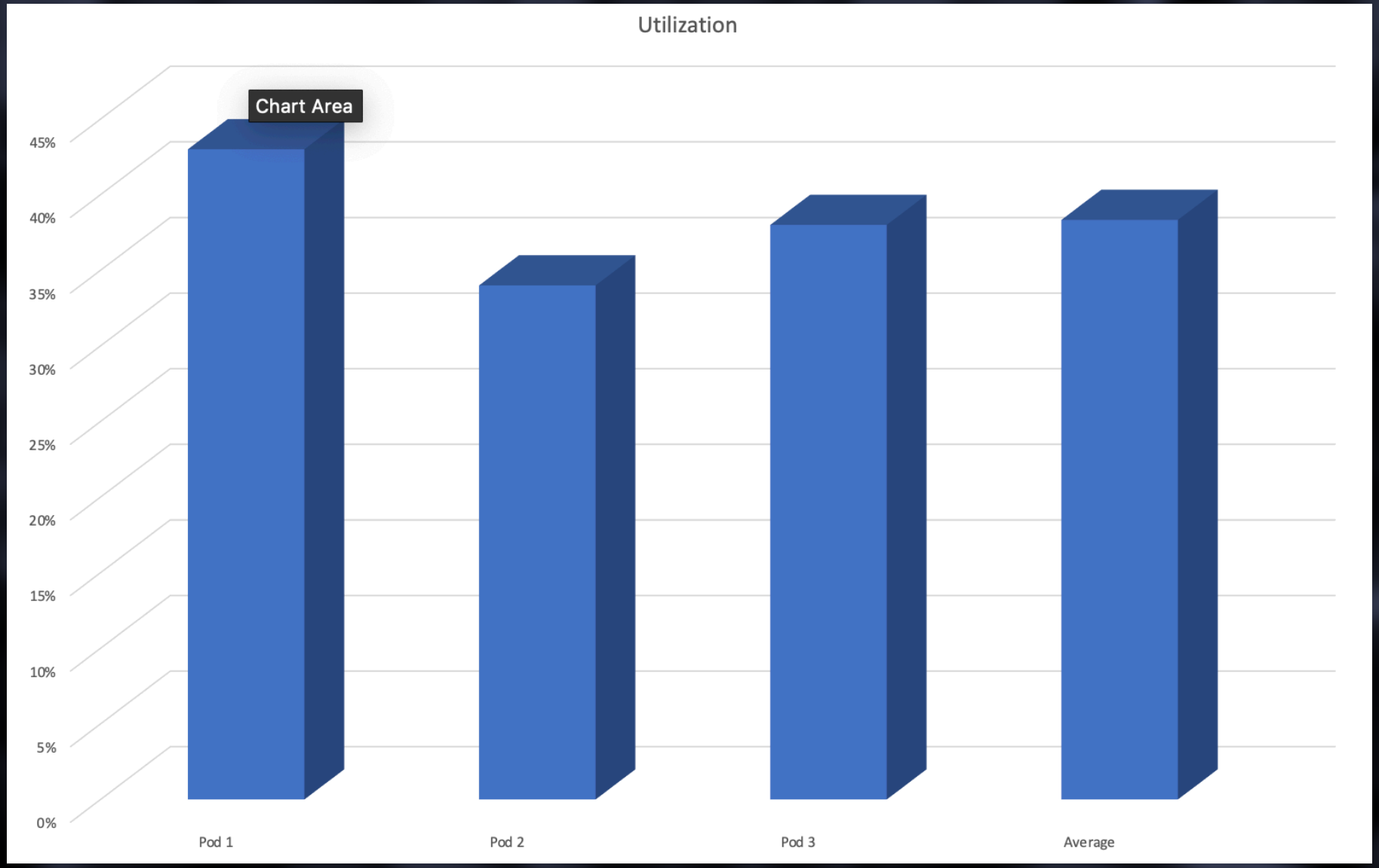
BUT! We've specified a minimum replicas count of 3. So Kubernetes will not remove more than that.

HPA will remove 3 pods from our cluster

HPA – scaling decisions

Target CPU utilization: 50%

Current utilization: 38%



HPA – thrashing

- If HPA monitored your deployments and made immediate changes, this would lead to “thrashing”, or instability by adding and removing pods quickly.
- Instead, we need to find a happy medium where the cluster is responsive, but responsive to a trend in metrics, not immediate.
- Again, we want to scale out fairly quickly to handle spikes in load, and scale in a bit slower.

HPA – thrashing

- This is accomplished with ”cool down” periods, or delays between two scale out or scale in operations. It gives your cluster a chance to stabilize and wait for the trend to either stabilize or change before making another scaling operation.
- By default, HPA:
  - Has an interval of 30 seconds between checking metrics.
  - Will wait 5 minutes between any two scaling in operations.
  - Will wait 3 minutes between any two scaling out operations.
- These delays are configurable at the cluster level with some flags.

HPA – walkthrough

Let's see the HPA in action.

1. We'll deploy Nginx and a Service. Let's first deploy one with no limits set.
2. We'll deploy a HPA. Let's take a look at this file first.
3. Let's take a look at the HPA running. Note the TARGETS column. The first column is the current resource usage. The second is the desired.
4. Using Apache Bench, let's put some load on the cluster and watch it scale. We'll put a watch on kubectl get all in a separate window.
   ```
   1.  ab -n 1000000 -c 10 http://localhost:30080/
   ```
5. Notice that the current usage metric on the HPA shows <unknown>. This is because we have not enabled any limits on our pods. The Nginx deployment also doesn't scale, even though we're hitting it with 1,000,000 requests with 10 concurrent requests. If this were production, we'd be effectively down.
6. Now let's delete the Nginx deployment and service.
7. We will choose a Nginx workload with limits set. Let's look at the file first.
8. We repeat the Apache Bench test, and should now see the HPA not only have metrics, but also scale. Note the extra pods that are created. Also note the numbers on the HPA.
9. Now let's kill the Bench test, and watch the HPA scale our deployment back in. Note that it will take several minutes, since HPA scales in slowly.

HPA – best practices

- Make sure you declare resource limits on your pods. Without them, HPA won't work.
- Make sure you have a reasonable minimum replica count. For production, one or two pods as a minimum isn't enough! Put some thought into this number.
- CPU utilization as a metric is great, but will not always make sense for every application. If other metrics make sense, then it's worth diving in deep to implement custom metrics. You're going to use a monitoring solution like Prometheus anyway, right?

HPA – best practices

- Give plenty of buffer for your utilization.
  - If you don’t have a buffer, then your application can’t handle sudden spikes in traffic.
  - Auto scaling is not immediate. It can take several minutes to scale out, especially if there’s already been a scaling operation. Consider that your application takes time to start up.
  - As you’ll learn next, if your cluster must also scale out before pods can be added, this adds extra time, so we must have a buffer.
  - Shoot for a 30% or so buffer, or a 70% target utilization.

HPA – best practices

- Make sure your application is well designed to handle auto scaling.
  - It should be stateless if at all possible. There should be no coupling between requests.
  - Requests should be short.

Init containers

Init containers – overview

- Init containers are specialized containers that run before regular containers to provide setup or initialization logic for your main application containers.
- A pod can have multiple containers to run your application. But it can also have one or more init containers, which are run before the application containers are started.
- They must run to completion successfully and run in the order specified in the PodSpec.
- If an init container fails, then Kubernetes restarts the pod unless the restartPolicy is never.

Init containers – uses

- We don’t want to litter our application containers with setup logic.
- We may want to use languages or utilities that we don’t want in our application container. For example, we may want to use curl for some setup work on a pod, but we don’t want that running in the application container for security reasons. An init container will separate this out.
- We may want to prevent our application container from starting until a dependent service is up. Our init container could loop, checking for the service before exiting successfully, thus starting the application container.
- Perhaps we want to register our pod with a remote system.
- We may want to clone a git repository into a volume attached to our pod.
- We may want to write out some configuration information into the filesystem for the main app to use.

Init containers – the spec

```
apiVersion: v1
kind: Pod
metadata:
  name: myapp-pod
  labels:
    app: myapp
spec:
  containers:
  - name: myapp-container
    image: busybox:1.28
    command: ['sh', '-c', 'echo The app is running! && sleep 3600']
  initContainers:
  - name: init-myservice
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup myservice; do echo waiting for myservice; sleep 2; done;']
  - name: init-mydb
    image: busybox:1.28
    command: ['sh', '-c', 'until nslookup mydb; do echo waiting for mydb; sleep 2; done;']
```

Init containers – walkthrough

- We'll create a pod that has one app container and two init containers.
- The job of the init containers is to check to see if other services are up and running. They will run one after the other.
- Let's create our pod, but leave the two dependent services down, and see what happens.
- Note that a kubectl logs on the pod will not show the output of the init containers. For that, we must specify which container with :
  - `kubectl logs myapp-pod -c init-myservice`
  - `kubectl logs myapp-pod -c init-mydb`
- Note that if we look at the logs for init-mydb, it will say the container isn't started yet. That's because myservice hasn't finished. Look at the logs for init-myservice.
- We can also inspect the pod before we've started the services to see it's state. `kubectl describe pod myapp-pod`
- If we `watch kubectl get all`, as soon as we apply the service.yaml file, our pod starts running because the init containers finish.

Init containers – other details

- If a pod is restarted, then init containers must run again.
- Because init containers can be restarted and re-executed, they should have logic to handle multiple reruns. For example, if an init container creates a directory, it should be able to handle the case where the directory already exists due to a previous run.
- Init containers have all the same specs of a regular container. Resource requests and limits for example. However, readinessProbe should not be used.
- The name of all containers in a pod are unique, including init containers. An error will result if you duplicate names.


cluster
autoscaler

Cluster autoscaler – introduction

- Remember, pods are mortal and ever shifting. But they always need a place to run!
- With HPA, you cannot predict how many pods are running in your cluster.
- How many pods you have today won't be the same tomorrow. Or for that matter, this afternoon.
- Nodes are expensive! We should make the most use of our resources and get rid of any not needed. Let's not overprovision.
- On the other hand, our pods are important. Let's not be too stingy.
- Manual intervention in our cluster should be avoided. The last thing an operations person wants to do is get up a 3am to add a new node to the cluster manually. Automation is paramount.

Cluster autoscaler – scheduling

- When a new pod is needed, Kubernetes looks for a node with available capacity and schedules the pod to be placed there.
- This is a complex operation with many decisions, but let's just keep it simple and imagine that the pod can go on any node in our cluster.
- But what happens when there isn't a node available with enough resources? We now have a homeless pod. The pod exists logically, but is in a pending state and isn't actually running anywhere. A new node must be provisioned to give the pod a home.

Cluster autoscaler – scheduling

- What about when load drops? Pods start being removed by HPA. We find ourselves with a bunch of nodes with few pods running on them and all with a lot of excess capacity.
- Cluster autoscaler steps in to shift pods around and condense them onto fewer nodes so that some nodes can be removed.

Cluster autoscaler – overview

- CA runs as a pod, usually on the master node.
- It watches the API for all nodes and pods on your cluster.
- It doesn’t use any metrics like HPA.
- CA works directly with your cloud provider to manage nodes. For example, if you’re running on AWS, it interfaces with Autoscaling Groups. For this reason it isn’t applicable for an on-premise cluster.

Cluster autoscaler – scaling out

- CA watches your pods. When any of them are in a state where they are pending because there isn't the needed capacity in your cluster, then it takes action to expand the cluster to accommodate it.
- CA requests a new node from your underlying cloud group.
- Once the node is created and has joined the cluster, then the scheduler detects that and schedules the pod onto the instance.

Cluster autoscaler – scaling in

- CA watches your cluster to determine if nodes can be removed to save cost.
- CA will initiate a scale in of your cluster if a node has been unneeded for 10 minutes and there haven't been any scale out operations in the last 10 minutes.
- A node is eligible to be removed if:
  - It's CPU and memory utilization are below 50%
  - All pods running on it can be moved elsewhere
  - There are no kube-system pods
  - There are no pods using local storage

Cluster autoscaler – helpful commands

- Kubectl describe configmap cluster-autoscaler-status –n kube-system
  - This contains lots of useful information about the current state of CA
- Kubectl get events

Cluster autoscaler – installation

- Detailed information and instructions can be found on the CA Github: https://github.com/kubernetes/autoscaler
- Because the CA is specific to cloud providers, there's more to it than just running a script. And the steps vary.
- In general, each one includes needing to modify cloud permissions to allow your cluster to manage the instance groups behind your pool of nodes.
  - For example in AWS, you need the IAM policy set to allow your role that your Kubernetes cluster runs as to describe auto scaling groups, set desired capacity, terminate instances, etc.
- From there, you run some scripts against your cluster provided by the CA codebase on Github.

Cluster autoscaler – best practices

- Do not manually modify nodes in your cluster (ie. Adding labels)
- Use Pod Disruption Budgets to manage scale-in without disrupting your services.
- Don’t use local storage for pods. This will prevent scale-in.
- Use homogenous clusters if possible. CA has some kinks to still be worked out when using clusters with a mix of node sizes. Unfortunately this is an impediment to using spot fleets of varying sizes.

Cluster autoscaler – best practices

- Remember, scaling activities at the node level take time. We're dealing with virtual machines rather than just pods. Expect it to take several minutes for a scale out scenario. Therefore you need that buffer as mentioned in the last section. The following times add up:
  - Delay for the HPA to schedule another pod. Then it goes pending.
  - Delay for CA to recognize the need for another node.
  - Delay for the cloud to launch a new instance and join the cluster.
  - Delay for the scheduler to place the pod on the instance.
  - Delay for the pod to start up and start receiving traffic.

review and wrap up