Scale down the Cluster

This lesson focuses on how to scale down the Cluster and the rules which govern it.

WE'LL COVER THE FOLLOWING

Scale down the nodes

Scaling up the cluster to meet the demand is essential since it allows us to host all the replicas we need to fulfill (some of) our SLAs. When the demand drops and our nodes become underutilized, we should scale down. That is not essential given that our users will not experience problems caused by having too much hardware in our cluster. Nevertheless, we shouldn't have underutilized nodes if we are to reduce expenses. Unused nodes result in wasted money. That is true in all situations, especially when running in Cloud and paying only for the resources we used. Even on-prem, where we already purchased hardware, it is essential to scale down and release resources so that they can be used by other clusters.

We'll simulate a decrease in demand by applying a new definition that will redefine the `HPAs` threshold to `2` (min) and `5` (max).

```
kubectl apply \
    -f scaling/go-demo-5.yml \
    --record

kubectl -n go-demo-5 get hpa
```

The **output** of the latter command is as follows.

```
```

```
NAME REFERENCE      TARGETS           MINPODS MAXPODS REPLICAS AGE
api  Deployment/api 0%/80%, 0%/80%   2       5       15       2m56s

db   StatefulSet/db 56%/80%, 10%/80% 3       5       3        2m57s
```

We can see that the min and max values of the `api` `HPA` changed to `2` and `5`. The current number of replicas is still `15`, but that will drop to `5` soon. The `HPA` already changed the replicas of the Deployment, so let's wait until it rolls out and take another look at the Pods.

```
kubectl -n go-demo-5 rollout status \
    deployment api

kubectl -n go-demo-5 get pods
```

The **output** of the latter command is as follows.

```
NAME    READY STATUS  RESTARTS AGE
api-... 1/1   Running 0        104s
api-... 1/1   Running 0        104s
api-... 1/1   Running 0        104s
api-... 1/1   Running 0        94s
api-... 1/1   Running 0        104s
db-0    2/2   Running 0        4m37s
db-1    2/2   Running 0        3m57s
db-2    2/2   Running 0        3m18s
```

Let's see what happened to the nodes.

```
kubectl get nodes
```

The **output** shows that we still have four nodes (or whatever was your number before we de-scaled the Deployment). Given that we haven't yet reached the desired state of only three nodes, we might want to take another look at the `cluster-autoscaler-status` ConfigMap.

```
kubectl -n kube-system \
    get configmap \
    cluster-autoscaler-status \
    -o yaml
```

The **output**, limited to the relevant parts, is as follows.

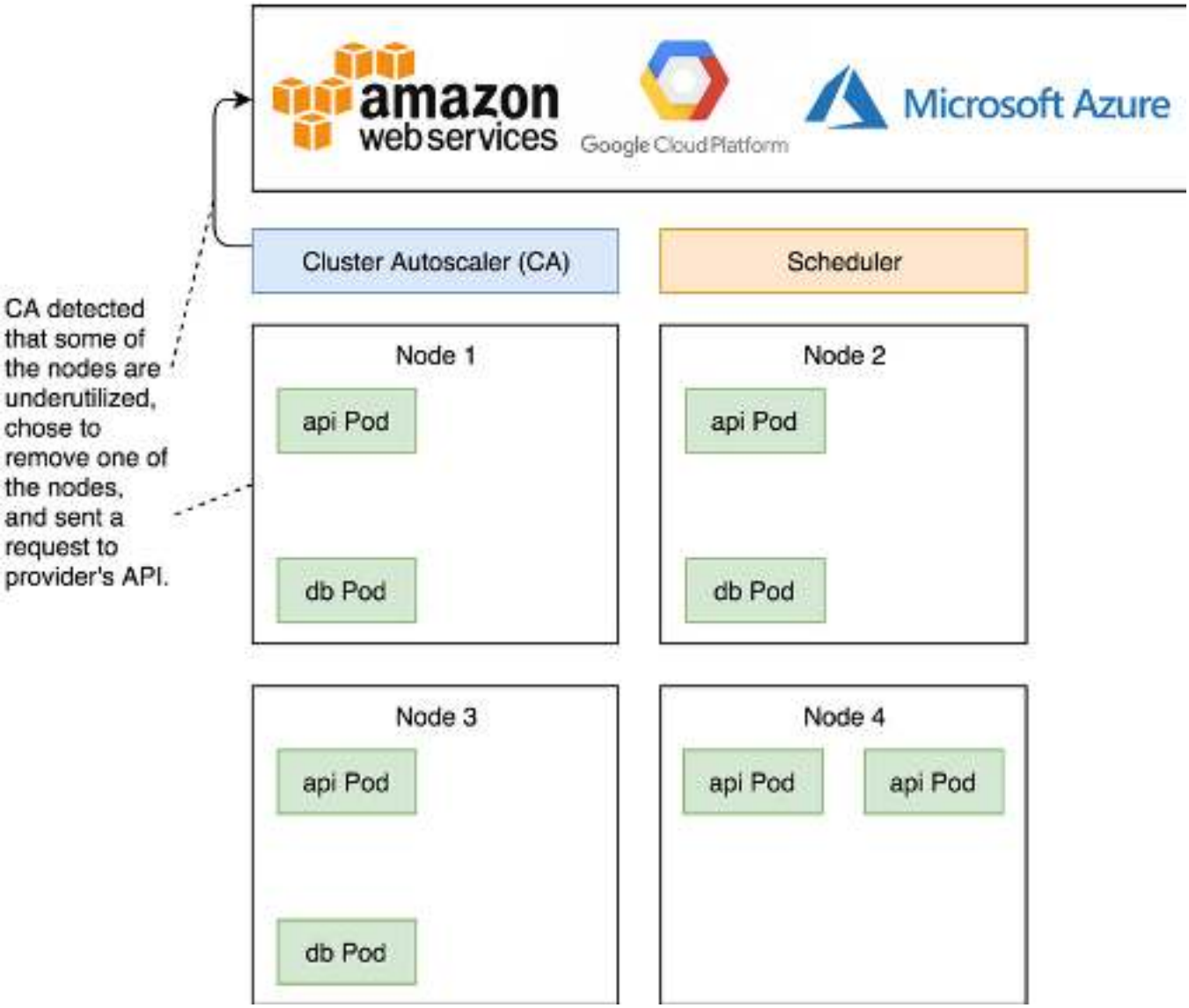
```
apiVersion: v1
data:
  status: |+
    Cluster-autoscaler status at 2018-10-03 ...
    Cluster-wide:
      Health: Healthy (ready=4 ...)
      ...
      ScaleDown: CandidatesPresent (candidates=1)
                 ...
    NodeGroups:
      Name:      ...gke-devops25-default-pool-f4c233dd-grp
      ...
      ScaleDown: CandidatesPresent (candidates=1)
                 LastProbeTime:      2018-10-03 23:06:...
                 LastTransitionTime: 2018-10-03 23:05:...
      ...
```

If your **output** does not contain `ScaleDown: CandidatesPresent`, you might need to wait a bit and repeat the previous command.

If we focus on the `Health` section of the cluster-wide status, all four nodes are still ready.

Candidate for removal

Judging by the cluster-wide section of the status, we can see that there is one candidate to `ScaleDown` (it might be more in your case). If we move to the `NodeGroups`, we can observe that one of them has `CandidatesPresent` set to `1` in the `ScaleDown` section (or whatever was your initial value before scaling up). In other words, one of the nodes is the candidate for removal. If it remains so for ten minutes, the node will be drained first to allow graceful shutdown of the Pods running inside it. After that, it will be physically removed through manipulation of the scaling group.



Cluster Autoscaler processes of scaling-down.

We should wait for ten minutes before we proceed, so this is an excellent opportunity to grab some coffee (or tea).

Now that enough time passed, we'll take another look at the `cluster-autoscaler-status` ConfigMap.

```
kubectl -n kube-system \
    get configmap \
    cluster-autoscaler-status \
    -o yaml
```

The **output**, limited to the relevant parts, is as follows.

```
apiVersion: v1
data:
```

```
  status: |+
    Cluster-autoscaler status at 2018-10-03 23:16:24...
    Cluster-wide:
      Health:    Healthy (ready=3 ... registered=4 ...)
                 ...
      ScaleDown: NoCandidates (candidates=0)
                 ...
    NodeGroups:
      Name:      ...gke-devops25-default-pool-f4c233dd-grp
      Health:    Healthy (ready=1 ... registered=2 ...)
                 ...
      ScaleDown: NoCandidates (candidates=0)
                 ...
```

From the cluster-wide section, we can see that now there are `3` ready nodes, but that there are still `4` (or more) registered. That means that one of the nodes was drained, but it was still not destroyed. Similarly, one of the node groups shows that there is `1` ready node, even though `2` are registered (your numbers might vary).

From the Kubernetes perspective, we are back to three operational worker nodes, even though the fourth is still physically present.

Now we need to wait a bit more before we retrieve the nodes and confirm that only three are available.

```
kubectl get nodes
```

The **output**, from GKE, is as follows.

```
NAME    STATUS ROLES  AGE VERSION
gke-... Ready  <none> 36m v1.9.7-gke.6
gke-... Ready  <none> 36m v1.9.7-gke.6
gke-... Ready  <none> 36m v1.9.7-gke.6
```

We can see that the node was removed and we already know from past experience that Kube Scheduler moved the Pods that were in that node to those that are still operational.

Now that you have experienced scaling down of your nodes, we'll explore the rule that governs the process.

The rules governing nodes scale down

Cluster Autoscaler iterates every 10 seconds (configurable through the `--scan-interval` flag). If the conditions for scaling up are not met, it checks whether there are unneeded nodes. It will consider a node eligible for removal when all of the following conditions are met:

- The sum of CPU and memory requests of all Pods running on a node is less than 50% of the node's allocatable resources (configurable through the `--scale-down-utilization-threshold` flag).
- All Pods running on the node can be moved to other nodes. The exceptions are those that run on all the nodes like those created through **DaemonSets**.

Whether a Pod might not be eligible for rescheduling to a different node when one of the following conditions are met:

- A Pod with affinity or anti-affinity rules that tie it to a specific node
- A Pod that uses local storage
- A Pod created directly instead of through controllers like Deployment, StatefulSet, Job, or ReplicaSet

All those rules boil down to a simple one. If a node contains a Pod that cannot be safely evicted, it is not eligible for removal.

Q

`Cluster Autoscaler` detected that some of the nodes are underutilized, chose to remove one of the nodes and sent a request to _____?

In the next lesson, we will discuss cluster scaling boundaries.