Scale up the Cluster

This lesson focuses on how to scale up the cluster and the rules which govern it.

WE'LL COVER THE FOLLOWING

- Scale up the nodes
    - Not enough resources to host all pods
- `Cluster Autoscaler` scaled the nodes
- The rules governing nodes scale up

Scale up the nodes

The objective is to scale the nodes of our cluster to meet the demand of our Pods. We want not only to increase the number of worker nodes when we need additional capacity, but also to remove them when they are underused. For now, we'll focus on the former, and explore the latter afterward.

Let's start by taking a look at how many nodes we have in the cluster.

```
kubectl get nodes
```

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

```
NAME              STATUS ROLES  AGE   VERSION
gke-devops25-... Ready  <none> 5m27s v1.9.7-gke.6
gke-devops25-... Ready  <none> 5m28s v1.9.7-gke.6
gke-devops25-... Ready  <none> 5m24s v1.9.7-gke.6
```

In your case, the number of nodes might differ. That's not important. What matters is to remember how many you have right now since that number will change soon.

Let's take a look at the definition of the `go-demo-5` application before we roll it out.

out.

```
cat scaling/go-demo-5-many.yml
```

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

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: api
  namespace: go-demo-5
spec:
  ...
  template:
    ...
    spec:
      containers:
      - name: api
        ...
        resources:
          limits:
            memory: 1Gi
            cpu: 0.1
          requests:
            memory: 500Mi
            cpu: 0.01
...
apiVersion: autoscaling/v2beta1
kind: HorizontalPodAutoscaler
metadata:
  name: api
  namespace: go-demo-5
spec:
  scaleTargetRef:
    apiVersion: apps/v1
    kind: Deployment
    name: api
  minReplicas: 15
  maxReplicas: 30
  ...
```

In this context, the only important part of the definition we are about to apply is the `HPA` connected to the `api` Deployment. Its minimum number of replicas is `15`. Given that each `api` container requests 500MB RAM, fifteen replicas

(7.5GB RAM) should be more than our cluster can sustain, assuming that it

was created using one of the Gists. Otherwise, you might need to increase the minimum number of replicas.

Let's apply the definition and take a look at the `HPAs`.

```
kubectl apply \
    -f scaling/go-demo-5-many.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   <unknown>/80%, <unknown>/80%   15        30
        1          38s
db     StatefulSet/db   <unknown>/80%, <unknown>/80%   3         5
         1          40s
```

Not enough resources to host all pods

It doesn't matter if the targets are still `unknown`. They will be calculated soon, but we do not care for them right now. What matters is that the `api` `HPA` will scale the Deployment to at least `15` replicas.

Next, we need to wait for a few seconds before we take a look at the Pods in the `go-demo-5` Namespace.

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

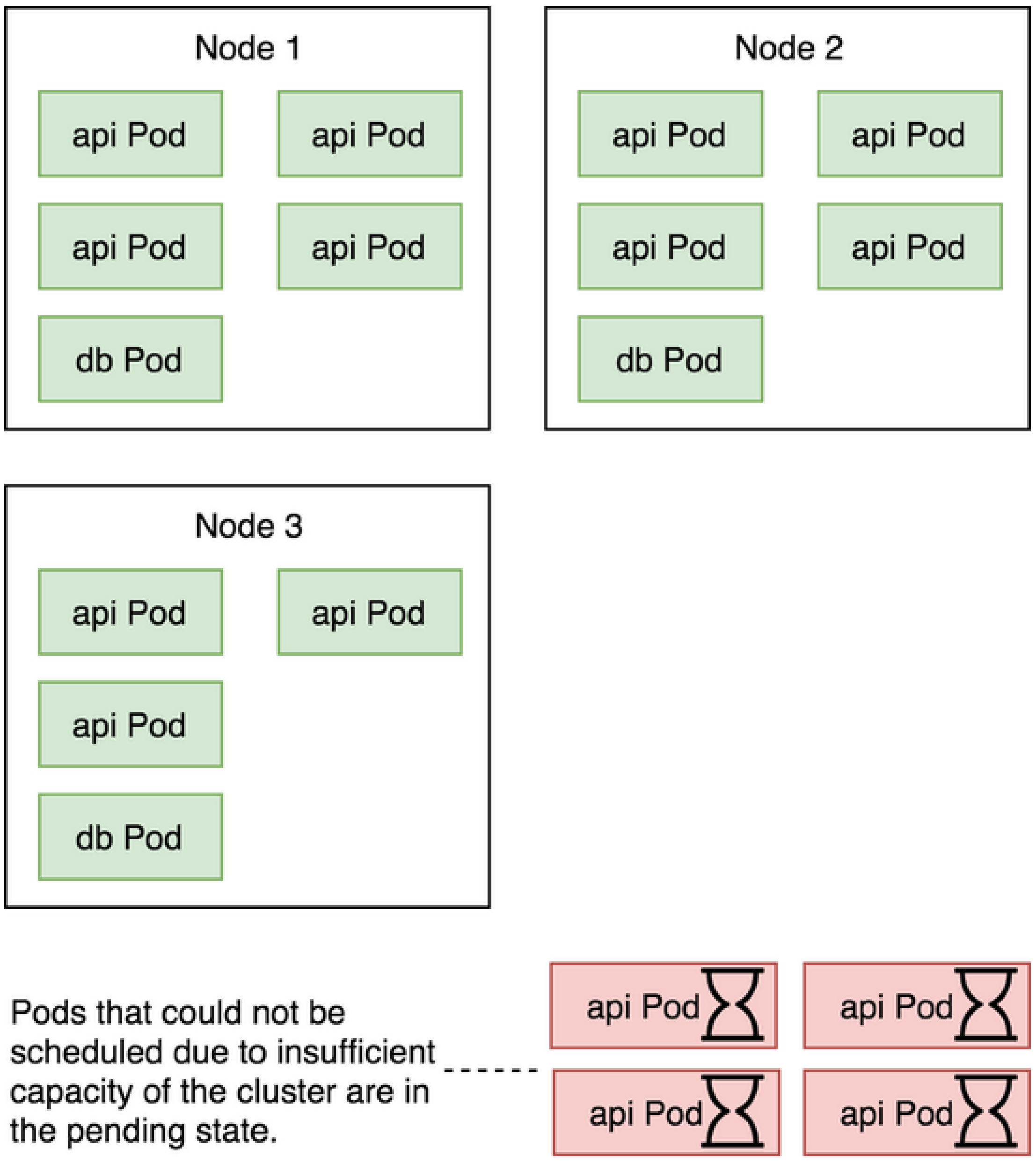
The **output** is as follows.

```
NAME     READY STATUS            RESTARTS AGE
api-... 0/1   ContainerCreating 0        2s
api-... 0/1   Pending           0        2s
api-... 0/1   Pending           0        2s
api-... 0/1   ContainerCreating 0        2s
api-... 0/1   ContainerCreating 0        2s
api-... 0/1   ContainerCreating 1        32s
api-... 0/1   Pending           0        2s
```

```
api-... 0/1    ContainerCreating 0         2s
api-... 0/1    ContainerCreating 0         2s

api-... 0/1    ContainerCreating 0         2s
api-... 0/1    ContainerCreating 0         2s
api-... 0/1    ContainerCreating 0         2s
api-... 0/1    Pending           0         2s
api-... 0/1    ContainerCreating 0         2s
api-... 0/1    ContainerCreating 0         2s
db-0    2/2    Running           0         34s
db-1    0/2    ContainerCreating 0         34s
```

We can see that some of the `api` Pods are being created, while others are pending. There can be quite a few reasons why a Pod would enter into the pending state. In our case, there are not enough available resources to host all the Pods.



Pods waiting for the cluster capacity to increase.

`Cluster Autoscaler` scaled the nodes

Let's see whether `Cluster Autoscaler` did anything to help with our lack of capacity. We'll explore the *ConfigMap* that contains `Cluster Autoscaler` status.

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

The **output** is too big to be presented in its entirety, so we'll focus on the parts that matter.

```
apiVersion: v1
data:
  status: |+
    Cluster-autoscaler status at 2018-10-03 ...
    Cluster-wide:
      ...
      ScaleUp: InProgress (ready=3 registered=3)
               ...

    NodeGroups:
      Name:    ...gke-devops25-default-pool-ce277413-grp
      ...
      ScaleUp: InProgress (ready=1 cloudProviderTarget=2)
               ...
```

The status is split into two sections; `Cluster-wide` and `NodeGroups`. The `ScaleUp` section of the cluster-wide status shows that scaling is `InProgress`. At the moment, there are `3` ready nodes.

If we move down to the `NodeGroups`, we'll notice that there is one for each group that hosts our nodes. In AWS those groups map to **Autoscaling Groups**, in case of Google to **Instance Groups**, and in Azure to **Autoscale**. One of the `NodeGroups` in the config has the `ScaleUp` section `InProgress`. Inside that group, `1` node is `ready`. The `cloudProviderTarget` value should be set to a number higher than the number of `ready` nodes, and we can conclude that `Cluster Autoscaler` already increased the desired amount of nodes in that group.

Depending on the provider, you might see three groups (GKE) or one

(EKS) node group. That depends on how each provider organizes its node groups internally.

Now that we know that `Cluster Autoscaler` is in progress of scaling up the nodes, we might explore what triggered that action.

Let's describe the `api` Pods and retrieve their events. Since we want only those related to `cluster-autoscaler`, we'll limit the output using `grep`.

```
kubectl -n go-demo-5 \
    describe pods \
    -l app=api \
    | grep cluster-autoscaler
```

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

```
  Normal TriggeredScaleUp 85s cluster-autoscaler pod triggered scale-up:
[{... 1->2 (max: 3)}]
  Normal TriggeredScaleUp 86s cluster-autoscaler pod triggered scale-up:
[{... 1->2 (max: 3)}]
  Normal TriggeredScaleUp 87s cluster-autoscaler pod triggered scale-up:
[{... 1->2 (max: 3)}]
  Normal TriggeredScaleUp 88s cluster-autoscaler pod triggered scale-up:
[{... 1->2 (max: 3)}]
```

We can see that several Pods triggered the `scale-up` event. Those are the Pods that were in the pending state. That does not mean that each trigger created a new node. `Cluster Autoscaler` is intelligent enough to know that it should not create new nodes for each trigger, but that, in this case, one or two nodes (depending on the missing capacity) should be enough. If that proves to be false, it will scale up again a while later.

Let's retrieve the nodes that constitute the cluster and see whether there are any changes.

```
kubectl get nodes
```

The **output** is as follows.

```
NAME                                        STATUS     ROLES     AGE      VERSI
ON
```

```
gke-devops25-default-pool-7d4b99ad-...   Ready      <none>   2m45s   v1.9.
7-gke.6

gke-devops25-default-pool-cb207043-...   Ready      <none>   2m45s   v1.9.
7-gke.6
gke-devops25-default-pool-ce277413-...   NotReady   <none>   12s     v1.9.
7-gke.6
gke-devops25-default-pool-ce277413-...   Ready      <none>   2m48s   v1.9.
7-gke.6
```

We can see that a new worker node was added to the cluster. It is not yet ready, so we'll need to wait for a few moments until it becomes fully operational.

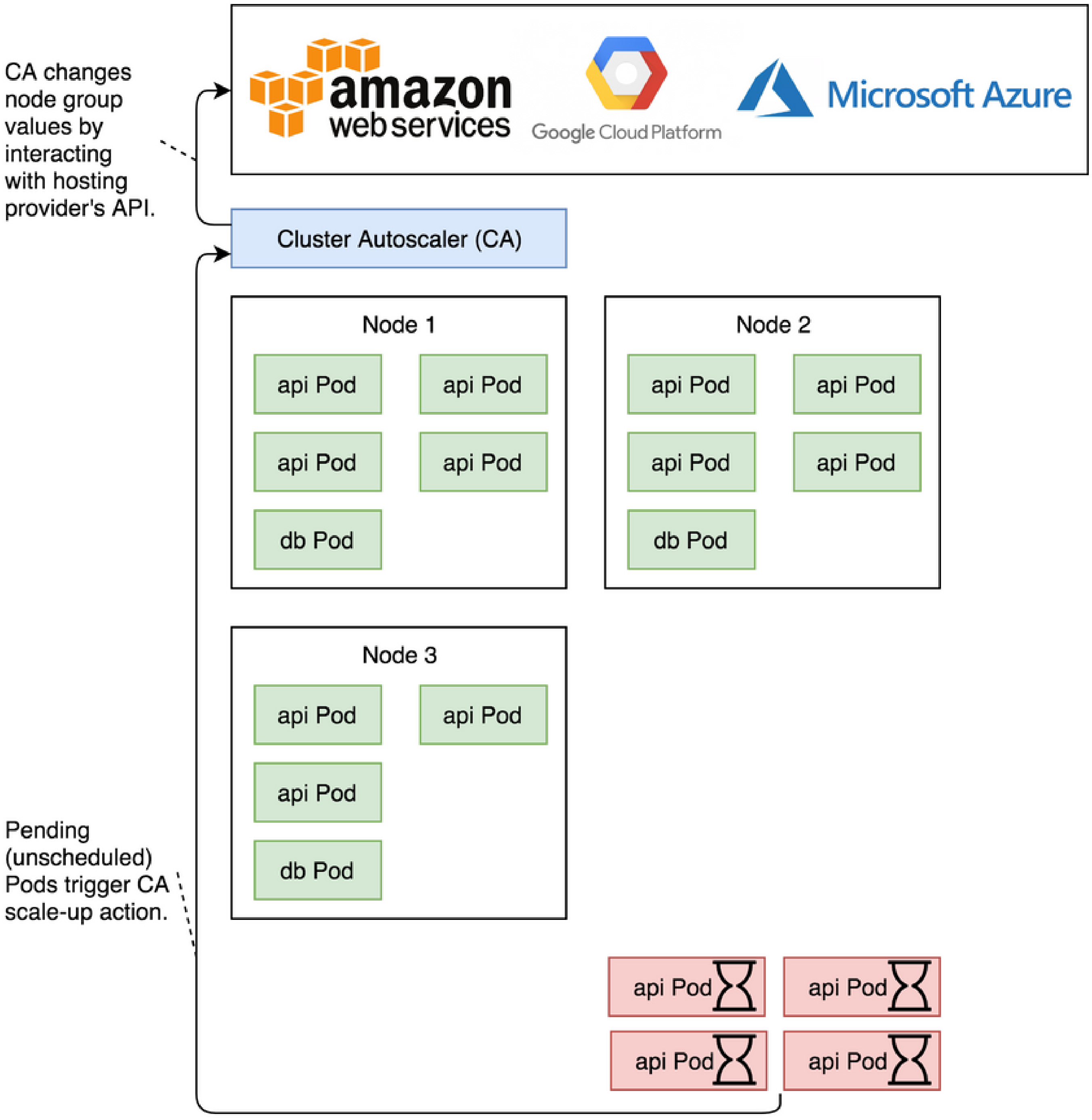
> 🔍 Please note that the number of new nodes depends on the required capacity to host all the Pods. You might see one, two, or more new nodes.

Q

The `ConfigMap` contains `Cluster Autoscaler` status.

COMPLETED 0%

1 of 1



Creation of the new node through node groups and rescheduling of the pending Pods.

The rules governing nodes scale up

`Cluster Autoscaler` monitors Pods through a watch on Kube API. It checks every 10 seconds whether there are any unschedulable Pods (configurable through the `--scan-interval` flag). In that context, a Pod is unschedulable when the Kubernetes Scheduler is unable to find a node that can accommodate it. For example, a Pod can request more memory than what is available on any of the worker nodes.

> 🔍 `Cluster Autoscaler` assumes that the cluster is running on top of some kind of node groups. As an example, in the case of AWS, those groups are Autoscaling Groups (ASGs). When there is a need for additional nodes, `Cluster Autoscaler` creates a new node by increasing the size of a node group.

`Cluster Autoscaler` assumes that requested nodes will appear within 15 minutes (configurable through the `--max-node-provision-time` flag). If that period expires and a new node was not registered, it will attempt to scale up a different group if the Pods are still in a pending state. It will also remove unregistered nodes after 15 minutes (configurable through the `--unregistered-node-removal-time` flag).

Next, we'll explore how to scale down the cluster.