Creating a Cluster: Discussing the Specifications

In this lesson, we will look into making feasible choices for our cluster.

We are, finally, ready to create a cluster. But, before we do that, we'll spend a bit of time discussing the requirements we might have. After all, not all clusters are created equal, and the choices we are about to make might severely impact our ability to accomplish the goals we might have.

Making Choice with Master Nodes

The first question we might ask ourselves is whether we want to have high-availability. It would be strange if anyone would answer no. Who doesn't want to have a cluster that is (almost) always available? Instead, we'll ask ourselves what are the things that might bring our cluster down.

When a node is destroyed, Kubernetes will reschedule all the applications that were running inside it into the healthy nodes. All we have to do is to make sure that, later on, a new server is created and joined the cluster, so that its capacity is back to the desired values. We'll discuss later how are new nodes created as a reaction to failures of a server. For now, we'll assume that will happen somehow.

Still, there is a catch. Given that new nodes need to join the cluster, if the failed server was the only master, there is no cluster to join. All is lost. The most important part is where the master servers are. They host the critical components without which Kubernetes cannot operate.

components without which Kubernetes cannot operate.

So, we need more than one master node. How about two? If one fails, we still have the other one. Still, that would not work.

Every piece of information that enters one of the master nodes is propagated to the others, and only after the majority agrees, that information is committed. If we lose majority (50%+1), masters cannot establish a quorum and cease to operate. If one out of two masters is down, we can get only half of the votes, and we would lose the ability to establish the quorum. Therefore, we need three masters or more. Odd numbers greater than one are “magic” numbers. Given that we won’t create a big cluster, three should do.

With three masters, we are safe from a failure of any single one of them. Given that failed servers will be replaced with new ones, as long as only one master fails at the time, we should be fault tolerant and have high availability.

> Always set an odd number greater than one for master nodes.

Making Choice with Data Centers

The whole idea of having multiple masters does not mean much if an entire data center goes down.

Attempts to prevent a data center from failing are commendable. Still, no matter how well a data center is designed, there is always a scenario that might cause its disruption. So, we need more than one data center. Following the logic behind master nodes, we need at least three. But, as with almost anything else, we cannot have any three (or more) data centers. If they are too far apart, the latency between them might be too high. Since every piece of information is propagated to all the masters in a cluster, slow communication between data centers would severely impact the cluster as a whole.

All in all, we need three data centers that are close enough to provide low latency, and yet physically separated, so that failure of one does not impact the others. Since we are about to create the cluster in AWS, we’ll use availability zones (AZs) which are physically separated data centers with low latency.

> Always spread your cluster between at least three data centers which are close enough to warrant low latency.

There's more to high-availability than running multiple masters and spreading a cluster across multiple availability zones. We'll get back to this subject later. For now, we'll continue exploring the other decisions we have to make.

Making Choice with Networking

Which networking shall we use? We can choose any of the following networkings:

- *kubenet*
- *CNI*
- *classic*
- *external*

The **classic** Kubernetes native networking is deprecated in favor of kubenet, so we can discard it right away.

The **external** networking is used in some custom implementations and for particular use cases, so we'll discard that one as well.

That leaves us with kubenet and CNI.

Container Network Interface (CNI) allows us to plug in a third-party networking driver. Kops supports Calico, flannel, Canal (Flannel + Calico), kopeio-vxlan, kube-router, romana, weave, and amazon-vpc-routed-eni networks. Each of those networks comes with pros and cons and differs in its implementation and primary objectives. Choosing between them would require a detailed analysis of each. We'll leave a comparison of all those for some other time and place. Instead, we'll focus on `kubenet`.

Kubenet is kops' default networking solution. It is Kubernetes native networking, and it is considered battle tested and very reliable. However, it comes with a limitation. On AWS, routes for each node are configured in AWS

VPC routing tables. Since those tables cannot have more than fifty entries,

kubenet can be used in clusters with up to fifty nodes. If you're planning to have a cluster bigger than that, you'll have to switch to one of the previously mentioned CNIs.

> Use kubenet networking if your cluster is smaller than fifty nodes.

The good news is that using any of the networking solutions is easy. All we have to do is specify the `--networking` argument followed with the name of the network.

Given that we won't have the time and space to evaluate all the CNIs, we'll use kubenet as the networking solution for the cluster we're about to create.

Making Choice with the Nodes' Size

Finally, we are left with only one more choice we need to make. What will be the size of our nodes? Since we won't run many applications, *t2.small* should be more than enough and will keep AWS costs to a minimum. *t2.micro* is too small, so we elected the second smallest among those AWS offers.

> ℹ You might have noticed that we did not mention persistent volumes. We'll explore them in the next chapter.

In the next lesson, we will run and verify our cluster.