What is Kubernetes?

Get introduced to Kubernetes and understand its architecture.

We'll cover the following

An overview of Kubernetes

Kubernetes is a container orchestration platform that helps better manage containerized applications in distributed environments. It comes with capabilities such as:

- Declarative APIs
- Zero downtime deployment
- Dynamic scheduling with various strategies
- Automated scaling
- Seamless rollout
- Container-centric infrastructure

It dramatically reduces the effort of maintaining applications running in a distributed system, so that a team can concentrate on making the core part, i.e., the application.

- Kubernetes has dramatically changed how applications are built and deployed. It has made essential abstractions that hide the complexity of managing containers, which empowers us to create more sophisticated applications with reliability, scalability, and resiliency. It's the de facto backbone of current cloud-native application developments because it brings in more velocity and efficiency than before.
- Kubernetes is also portable and provider-agnostic, so it can run on various infrastructures such as bare-metal machines, virtual machines, and public or private cloud platforms. It can even run on our Raspberry Pi.

- Kubernetes adoption has truly gone mainstream. It has been globally used across organizations of all types and sizes. The next frontier for Kubernetes is not adoption anymore, but extension and customization.
- Kubernetes is a well-designed and influential project that allows users to easily run scalable and highly available containerized workloads. It provides lots of highly abstracted objects and APIs that can help users efficiently design and develop containerized services.

Design principles

While the architecture, concepts, and components of Kubernetes seem daunting at first glance, it provides unparalleled and robust feature sets. By understanding the building blocks of Kubernetes, we can get to know how to fully leverage its capabilities to run our workloads with scalability and high availability.

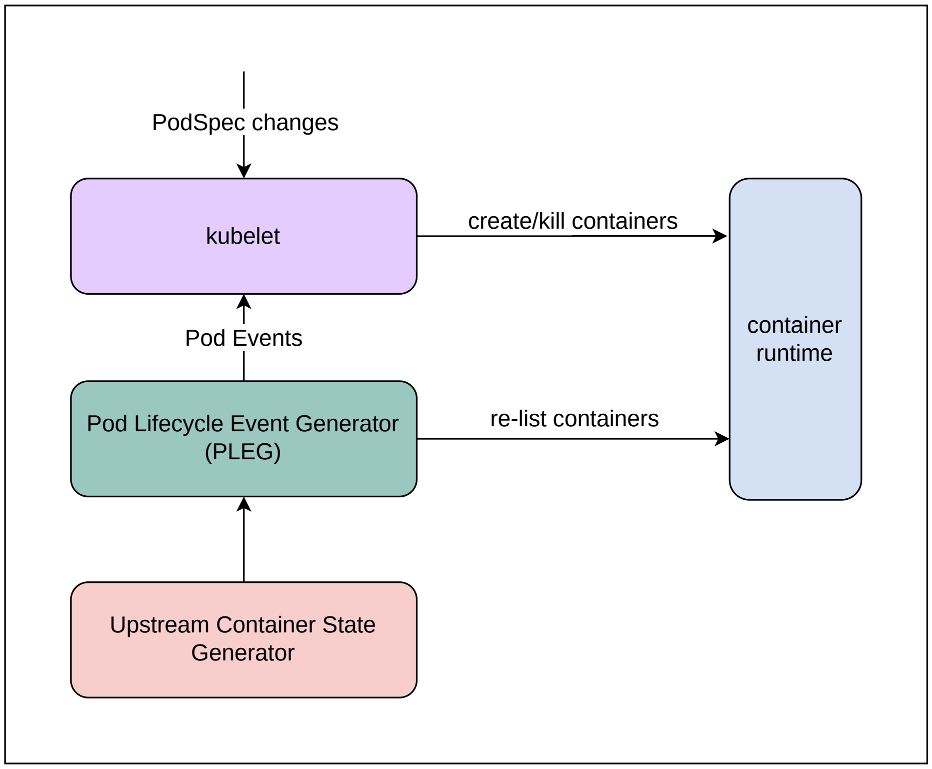
Before diving into Kubernetes architecture, let’s take some time to discuss the design principles of Kubernetes first. These principles will not only help us better understand the architecture of Kubernetes, but also provide us with more hints and guidelines when we try to extend and customize it.

Declarative APIs

Perhaps the most important and distinguished design principle in Kubernetes is the **declarative API**, which can simply be thought of as the desired state of our system. Kubernetes ensures every declarative object reflects and matches its desired state. We only need to define what our system should look like in a desired state and leave the rest to Kubernetes. This powers our systems to be self-healing and available, without the need for human intervention.

All the controllers in Kubernetes will reconcile declarative objects, detect the differences between the current and desired states, and proceed with the necessary operations that can drive those objects back to the desired state. This is a rule that must be followed when we implement our controllers, and will also be discussed in depth in later lessons.

Taking the `kubelet` as an example, the PLEG (Pod Lifecycle Event Generator) controller in the `kubelet` periodically re-lists all containers to discover changes and generate pod events on mismatched configurations. Then, the `kubelet` will take the necessary actions to bring those containers back to desired states.



PLEG in the kubelet

Self-healing

Kubernetes is a self-healing system. Though this could be included in the part above about declarative APIs, self-healing is worthy of being regarded as another essential design principle. It's also a rule that must be followed for any external controllers or operators.

On the one hand, Kubernetes itself is self-healing. With the help of informers, all components inside Kubernetes keep some states or caches that will be periodically refreshed. Those caches also help relieve the burden for `kube-apiserver`. Even if an event is missed or the network suffers a temporary loss, it will soon come back to life again.

On the other hand, Kubernetes will continuously take actions to ensure those objects are in the desired states, including guarding against any failures and exceptions, automated horizontal scaling, etc. Additionally, this helps improve the stability, availability, and reliability of our system. Let's suppose we have declared an application with three replicas to Kubernetes; it will continuously make sure there are precisely three healthy replicas running out there, no more and no less. If someone manually destroys a replica, Kubernetes will create one and bring it to the desired state. Likewise, Kubernetes will remove redundant replicas.

Abstractions and decoupling

Kubernetes is designed as a loosely coupled distributed system whose components are running separately. It has provided many abstractions as well so that we can use different implementations. For example, if we run Kubernetes on AWS, ELB (Elastic Load Balancing) can be used as an external service load balancer. Because of these abstractions, we can easily customize and extend Kubernetes to what we need. This is how Kubernetes can be so powerful.

At the same time, Kubernetes naturally supports running decoupled services by making abstractions. Kubernetes uses `Pod` objects to describe a microservice instance. An abstract object `Service` is used to expose an application running on a set of `Pod` objects; a `Pod` is ephemeral, so it would be terminated and replaced at any time. Such a decoupling design helps keep track of healthy `Pod` objects and bind their IP addresses automatically. As a result, those microservices could be scaled separately because they're logically independent. Developers can apply new changes into production with a higher velocity.

Immutable infrastructure

Last but not least, Kubernetes adheres to the principles of immutable infrastructure.

Everything is prone to failure, and things do go wrong over time. Time is often wasted on chasing down the root causes and debugging the runtime. In most cases, the real causes are misconfigurations.

Kubernetes makes `Pod` objects the immutable infrastructure. A `Pod` is the fundamental building block in Kubernetes, consisting of one or more closely related containers. The benefits of using immutable infrastructure are tremendous, such as simplifying operations by treating `Pod` objects as cattle, reducing configuration drifts, being less error-prone, having a consistent environment, etc. By replacing, instead of maintaining, failed/erroneous instances, it becomes much easier to keep high availability and ensure the service-level agreements. Also, it's feasible to roll back to a previous state if an error occurs.

Kubernetes architecture

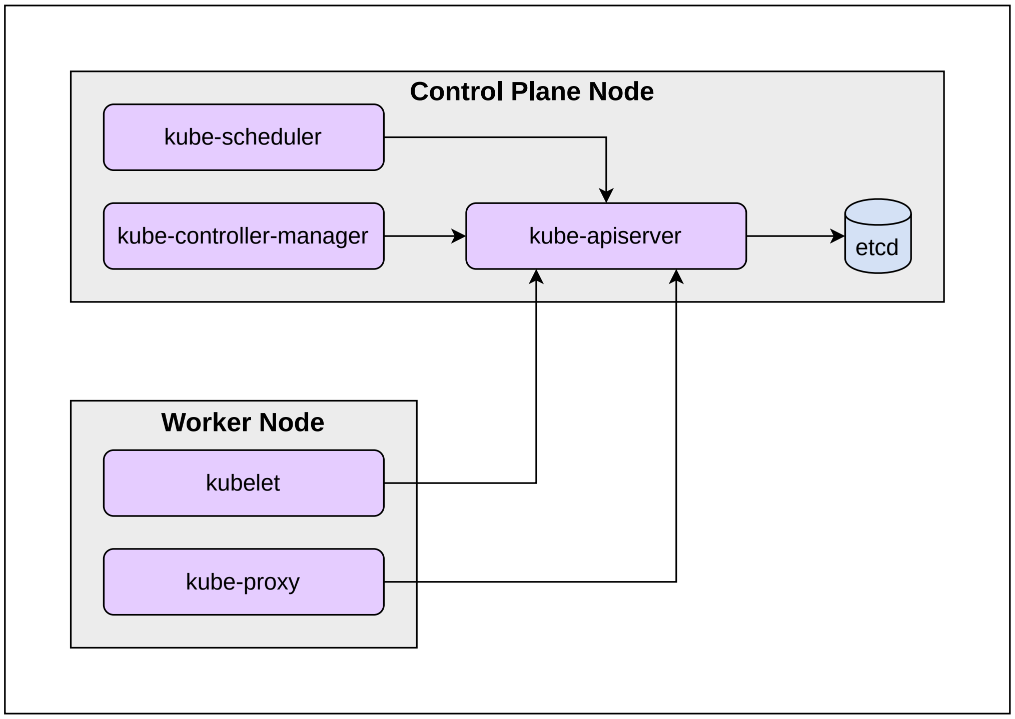
Now, let's take a closer look at Kubernetes architecture. Kubernetes runs on a bunch of nodes (i.e., bare-metal servers, virtual machines, etc.), which can be categorized into two types:

- The **control plane nodes** (previously known as the "master nodes" host the Kubernetes control plane, acting as a gateway. They manage the overall cluster by exposing REST APIs for users and clients, picking candidate nodes (i.e., scheduling), orchestrating communications between other components, etc. Multiple components, including `kube-apiserver`, `kube-scheduler`, `kube-controller-manager`, and `etcd` comprise the control plane.
- The **worker nodes** are where the actual applications are running. On worker nodes, `kubelet` and `kube-proxy` are deployed.

The `etcd` cluster stores the entire cluster's state, including configurations, declarations, and the statuses of all objects. It serves as the brain of a Kubernetes cluster. Thus, a highly available `etcd` cluster is usually deployed with an odd number of nodes, communicated by the Raft algorithm.

The control plane provides most of the core logic in Kubernetes. It's strongly recommended to have multiple master nodes for high availability. If any individual control plane node goes down, or if one of the control plane components faces any temporary failures, the whole cluster will work unaffected.

All the components in Kubernetes are loosely coupled. We'll later elaborate on these five components in the following lessons.



Kubernetes architecture

There's no intrinsic difference between control plane nodes and worker nodes. We could run the `kubelet` on control plane nodes. However, this isn't advisable, because it might bring in security risks and affect control plane stability. Most installers, like `kubeadm`, will set taints to control plane nodes by default. If we do want to schedule `Pod` objects on these nodes, we could run the commands given below to remove such taints:

```
kubectl taint nodes --all node-role.kubernetes.io/control-plane- node-role.kubernetes.io/master-
```

The output will be as follows:

```
node "node-01" untainted
node "node-02" untainted
...
```

The command above will remove the taints `node-role.kubernetes.io/control-plane` and `node-role.kubernetes.io/master` from any nodes, including the control plane nodes. Legacy taint name `node-role.kubernetes.io/master` is officially removed starting from version `1.25` (click this tracking issue to learn more).

Creating a cluster

There are quite a few methods or tools to create a Kubernetes cluster. In this lesson, we'll cover a few of them.

The `Kind` method is not suggested, because it's hard to configure and customize.

The `kubeadm` method is the officially recommended way to bootstrap a minimum viable Kubernetes cluster that conforms to best practices. In this course, we strongly suggest using `kubeadm` to create a Kubernetes cluster due to the following reasons:

- We don't need a very large cluster. A small cluster with two nodes is enough for this course.
- It's super easy and straightforward. It only takes two steps, `kubeadm init` and `kubeadm join`, to create a cluster.
- It helps eliminate complex configurations and lets us focus on the main parts. In most of the cases, default configurations are enough for our use cases. It is easier to demonstrate what we've changed and customize the clusters using simple configurations.
- It's convenient to purge the cluster with `kubeadm reset` and recreate a new one when our Kubernetes cluster gets into trouble.
- It's easy to get a Kubernetes cluster of any version. Every release of `kubeadm` has a range of supported Kubernetes versions. We can bootstrap a Kubernetes cluster with our desired version by choosing a matched `kubeadm`. This can be very useful for performing fast validations to verify our customization.