# What is Kubernetes?

Kubernetes (also known as K8s) is a portable, extensible, open source platform for managing containerized workloads and services, that facilitates both declarative configuration and automation. It has a large, rapidly growing ecosystem. Kubernetes services, support, and tools are widely available.

It was designed by Google and now it is maintained by the Cloud Native Computing Foundation. Kubernetes combines over 15 years of Google's experience running production workloads at scale with best-of-breed ideas and practices from the community.

First version was released on July 21, 2015.

# Why we need Kubernetes and what it can do?

Containers are a good way to bundle and run applications. In a production environment, it’s necessary to manage the containers that run the applications and ensure that there is no downtime.

Kubernetes takes care of:

* **Service discovery and load balancing**   
  Kubernetes can expose a container using the DNS name or using their own IP address. Kubernetes is able to load balance and distribute the network traffic so that the deployment is stable.
* **Storage orchestration**   
  Kubernetes allows us to automatically mount a storage system of our choice, such as local storages, public cloud providers, and more.
* **Automated rollouts and rollbacks**Kubernetes can change the actual state to the desired state at a controlled rate. For example, we can automate Kubernetes to create new containers or remove existing for our deployment.
* **Automatic bin packing**   
  We can tell Kubernetes how much CPU and memory each container needs and they would be setup to make the best use of our resources.
* **Self-healing**   
  Kubernetes restarts containers that fail, replaces containers, kills containers that don't respond to user-defined health check, and doesn't advertise them to clients until they are ready to serve.
* **Secret and configuration management**   
  Kubernetes lets us store and manage sensitive information, such as passwords, OAuth tokens, and SSH keys.

## What Kubernetes is not?

Kubernetes is not a traditional, all-inclusive Platform as a Service system. Since Kubernetes operates at the container level rather than at the hardware level, it provides some generally applicable features common to PaaS offerings, such as deployment, scaling, load balancing, and lets users integrate their logging, monitoring, and alerting solutions. However, Kubernetes is not monolithic, and these default solutions are optional and pluggable. Kubernetes provides the building blocks for building developer platforms, but preserves user choice and flexibility where it is important.

Kubernetes:

* **Does not limit the types of applications supported.** Kubernetes aims to support an extremely diverse variety of workloads, including stateless, stateful, and data-processing workloads. If an application can run in a container, it should run great on Kubernetes.
* **Does not deploy source code and does not build our application.** Continuous Integration, Delivery, and Deployment (CI/CD) workflows are determined by organization cultures and preferences as well as technical requirements.
* **Does not provide application-level services, such as middleware** (for example, message buses), **data-processing frameworks** (for example, Spark), **databases** (for example, MySQL), **caches, nor cluster storage systems** (for example, Ceph) **as built-in services.** Such components can run on Kubernetes, and can be accessed by applications running on Kubernetes through portable mechanisms, such as the Open Service Broker.
* **Does not dictate logging, monitoring, or alerting solutions.** It provides some integrations as proof of concept, and mechanisms to collect and export metrics.
* **Does not provide nor mandate a configuration language/system** (for example, Jsonnet). It provides a declarative API that may be targeted by arbitrary forms of declarative specifications.
* **Does not provide nor adopt any comprehensive machine configuration, maintenance, management, or self-healing systems.**
* Additionally, Kubernetes is not a mere orchestration system. In fact, it eliminates the need for orchestration. The technical definition of orchestration is execution of a defined workflow: first do A, then B, then C. In contrast, Kubernetes comprises a set of independent, composable control processes that continuously drive the current state towards the provided desired state. It shouldn't matter how you get from A to C. Centralized control is also not required. This results in a system that is easier to use and more powerful, robust, resilient, and extensible.

# Kubernetes components

Kubernetes cluster consists of a set of worker machines, called nodes, that run containerized applications. Every cluster has at least one worker node.

The worker nodes host the Pods that are the components of the application workload. The control plane manages the worker nodes and the Pods in the cluster. In production environments, the control plane usually runs across multiple computers and a cluster usually runs multiple nodes, providing fault-tolerance and high availability.

Let's take a closer look at the components you can see on the right side of the picture.

* **kube-apiserver** is a component of the Kubernetes control plane that exposes the Kubernetes API that controls cluster. The API server is the front end for the Kubernetes control plane. This component is designed to scale horizontally to balance traffic between instances.
* **etcd** is highly-available key value store used as Kubernetes' backing store for all cluster data. But we should not forget to back it up.
* **kube-scheduler** is component that watches for newly created Pods with no assigned [node](https://kubernetes.io/docs/concepts/architecture/nodes/), and selects a node for them to run on.
* **kube-controller-manager** is component that runs controller processes. Each controller is a separate process, but to reduce complexity, they are all compiled into a single binary and run in a single process.
* **cloud-controller-manager** is component that embeds cloud-specific control logic. The cloud controller manager lets us link cluster into cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with our cluster. The cloud-controller-manager only runs controllers that are specific to cloud provider.   
  As with the kube-controller-manager, the cloud-controller-manager combines several logically independent control loops into a single binary that we run as a single process. It’s possible to scale horizontally to improve performance or to help tolerate failures.
* **kubelet** is an agent that runs on each node in the cluster. It makes sure that containers are running in a Pod. The kubelet takes a set of pod’s specifications that are provided through various mechanisms and ensures that the containers described in those specifications are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.
* **kube-proxy** is a network proxy that runs on each node in our cluster. This component maintains network rules on nodes. These network rules allow network communication to our Pods from network sessions inside or outside of the cluster. kube-proxy uses the operating system packet filtering layer if there is one and it's available. Otherwise, kube-proxy forwards the traffic itself.

# The Kubernetes network model

Every Pod in a cluster gets its own unique cluster-wide IP address. This means we do not need to explicitly create links between Pods and we almost never need to deal with mapping container ports to host ports.  
This creates a clean, backwards-compatible model where Pods can be treated much like VMs or physical hosts from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

Kubernetes imposes the following fundamental requirements on any networking implementation:

* pods can communicate with all other pods on any other node without NAT (network address translation)
* agents on any node (system daemons, kubelet) can communicate with all pods on that node.

Kubernetes IP addresses exist at the Pod scope - containers within a Pod share their network namespaces - including their IP address and MAC address. This means that containers within a Pod can all reach each other's ports on localhost. This also means that containers within a Pod must coordinate port usage, but this is no different from processes in a VM. This is called the "IP-per-pod" model.

# Service

In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them. The set of Pods (so called ReplicaSet) targeted by a Service is usually determined by a selector.

With Kubernetes we don't need to modify our application to use an unfamiliar service discovery mechanism. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

# Ingress

To apply load balancing we need to use ingress – an API object that manages external access to services in a cluster, typically HTTP. Ingress exposes HTTP and HTTPS routes from outside the cluster to [services](https://kubernetes.io/docs/concepts/services-networking/service/) within the cluster.

An Ingress may be configured to give Services externally-reachable URLs, different load balancing strategy, and other tasks.

# Deployment

Deployment is a mechanism that provides declarative updates for sets of Pods.

We describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate. We can define Deployments to create new ReplicaSet, or to remove existing Deployments and adopt all their resources with new Deployments.

Typical use cases of this mechanism:

* Create a Deployment to rollout a ReplicaSet. The ReplicaSet creates Pods in the background. Check the status of the rollout to see if it succeeds or not.
* Declare the new state of the Pods by updating the Deployment. A new ReplicaSet is created and the Deployment manages moving the Pods from the old ReplicaSet to the new one at a controlled rate. Each new ReplicaSet updates the revision of the Deployment.
* Rollback to an earlier Deployment revision if the current state of the Deployment is not stable. Each rollback updates the revision of the Deployment.
* Scale up the Deployment to facilitate more load.
* Pause the rollout of a Deployment to apply multiple fixes and then resume it to start a new rollout.
* Use the status of the Deployment as an indicator that a rollout has stuck.
* Clean up older ReplicaSets that you don't need anymore.

# Horizontal Pod Autoscaling

In Kubernetes, a HorizontalPodAutoscaler automatically updates a workload resource (for example Deployment) to be able to serve the current workload that is determined with specified metrics.

Horizontal scaling in Kuberrnetes means to deploy more Pods. This is different from vertical scaling, which for Kubernetes would mean assigning more resources (memory or CPU) to the Pods that are already running for the workload.

If the load decreases, and the number of Pods is above the configured minimum, the HorizontalPodAutoscaler instructs the workload resource to scale back down.

The HorizontalPodAutoscaler is implemented as a Kubernetes API resource and a controller. The resource determines the behavior of the controller. The horizontal pod autoscaling controller, running within the Kubernetes control plane, periodically adjusts the desired scale of its target (for example, a Deployment) to match observed metrics such as average CPU utilization, average memory utilization, or any other custom metric we specify.

# Storage

On-disk files in a container are ephemeral, which presents some problems for non-trivial applications when running in containers. One problem is the loss of files when a container crashes. The kubelet restarts the container but with a clean state. A second problem occurs when sharing files between containers running together in a Pod. The Kubernetes volume abstraction solves both of these problems.

Kubernetes supports many types of volumes. A pod can use any number of volume types simultaneously. Ephemeral volume types have a lifetime of a pod, but persistent volumes exist beyond the lifetime of a pod. When a pod ceases to exist, Kubernetes destroys ephemeral volumes; however, Kubernetes does not destroy persistent volumes. For any kind of volume in a given pod, data is preserved across container restarts.

At its core, a volume is a directory, possibly with some data in it, which is accessible to the containers in a pod.

Kubernetes has PersistentVolume subsystem that provides an API for users and administrators that abstracts details of how storage is provided from how it is consumed. To do this two API resources were introduced by K8s developers: PersistentVolume and PersistentVolumeClaim. This subsystem provides cluster administrators with an ability to use large number of volumes that differ more than just in size and access mode.