# **Kubernetes**

### **Kubernetes Architecture and Components:**

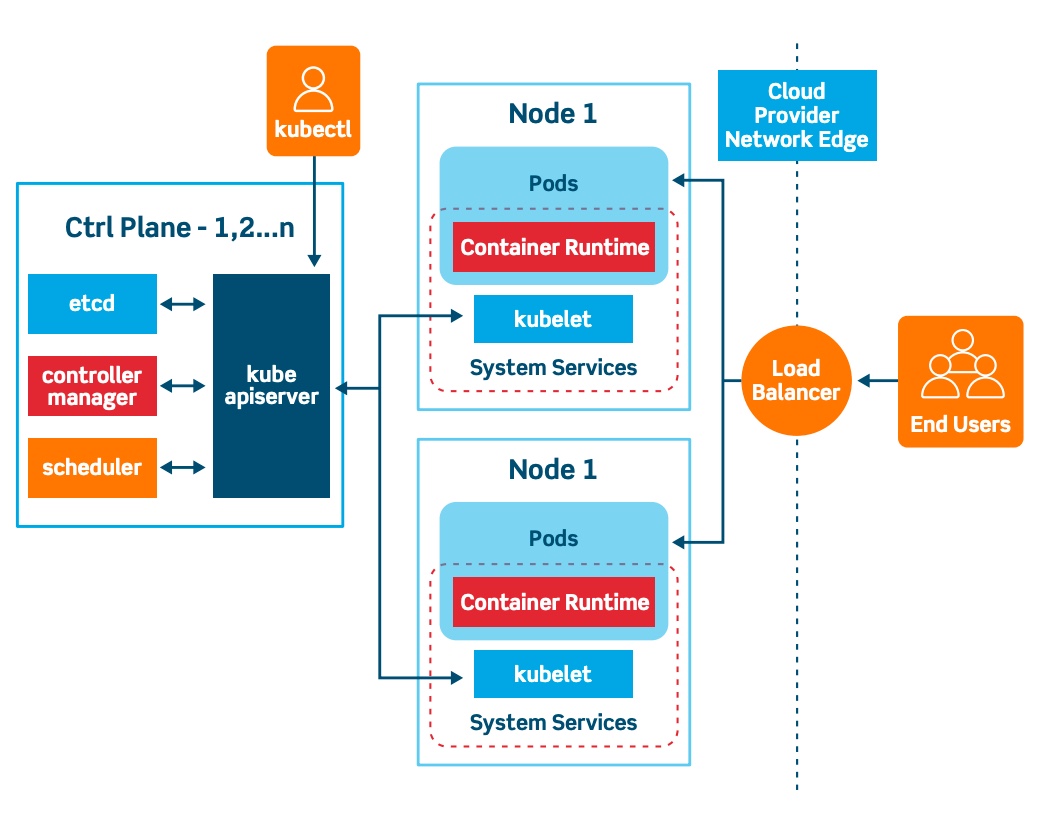
 [Kubernetes Control Plane](https://spot.io/resources/kubernetes-architecture-11-core-components-explained/?hsa_ver=3&hsa_kw=&hsa_cam=16712870764&hsa_tgt=dsa-844457319037&hsa_acc=8916801654&hsa_mt=&hsa_net=adwords&hsa_ad=590255918540&hsa_src=g&hsa_grp=133419158565#a2) Components on the Master Node

* kube-apiserver
* kube-scheduler
* kube-controller-manager
* etcd
* cloud-controller-manager

 [Kubernetes Worker Nodes](https://spot.io/resources/kubernetes-architecture-11-core-components-explained/?hsa_ver=3&hsa_kw=&hsa_cam=16712870764&hsa_tgt=dsa-844457319037&hsa_acc=8916801654&hsa_mt=&hsa_net=adwords&hsa_ad=590255918540&hsa_src=g&hsa_grp=133419158565#a3)

* Nodes
* Pods
* Container Runtime Engine
* kubelet
* kube-proxy
* Container Networking

1. An administrator creates and places the desired state of an application into a manifest file.
2. The file is provided to the Kubernetes API Server using a CLI or UI. Kubernetes’ default command-line tool is called ***kubectl***. In case you need a comprehensive list of kubectl commands, check out our [Kubectl Cheat Sheet](https://phoenixnap.com/kb/kubectl-commands-cheat-sheet).
3. Kubernetes stores the file (an application’s desired state) in a database called the **Key-Value Store (etcd)**.
4. Kubernetes then implements the desired state on all the relevant applications within the cluster.
5. [Kubernetes continuously monitors the elements of the cluster](https://phoenixnap.com/kb/prometheus-kubernetes-monitoring) to make sure the current state of the application does not vary from the desired state.



#### **API Server**

The **API Server** is the front-end of the control plane and the only component in the control plane that we interact with directly. Internal system components, as well as external user components, all communicate via the same API.

#### **Key-Value Store (etcd)**

The Key-Value Store, also called **etcd**, is a database Kubernetes uses to back-up all cluster data. It stores the entire configuration and state of the cluster. The Master node queries **etcd** to retrieve parameters for the state of the nodes, pods, and containers.

#### **Controller**

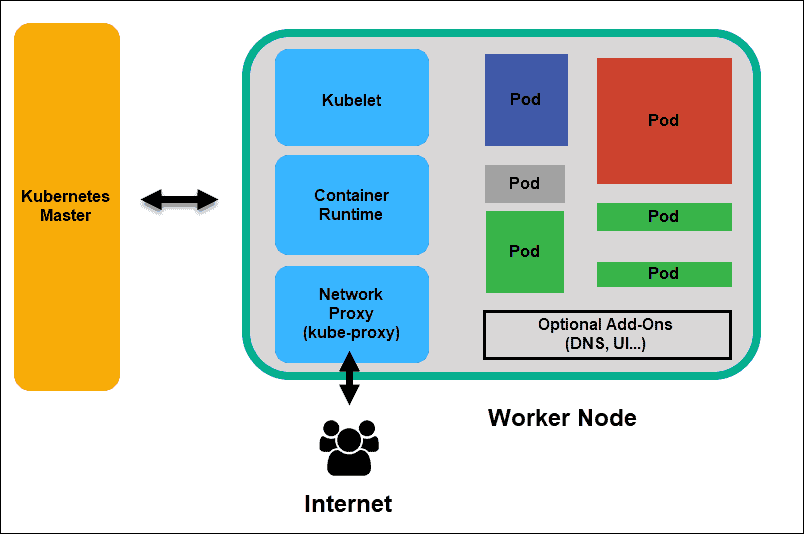
The role of the **Controller** is to obtain the desired state from the API Server. It checks the current state of the nodes it is tasked to control, and determines if there are any differences, and resolves them, if any.

#### **Scheduler**

A **Scheduler** watches for new requests coming from the API Server and assigns them to healthy nodes. It ranks the quality of the nodes and deploys pods to the best-suited node. If there are no suitable nodes, the pods are put in a pending state until such a node appears.

### **What is Worker Node in Kubernetes Architecture?**

Worker nodes listen to the API Server for new work assignments; they execute the work assignments and then report the results back to the Kubernetes Master node.



**Kubernetes Worker Node**

#### **Kubelet**

The **kubelet** runs on every node in the cluster. It is the principal Kubernetes agent. By installing kubelet, the node’s CPU, RAM, and storage become part of the broader cluster. It watches for tasks sent from the API Server, executes the task, and reports back to the Master. It also monitors pods and reports back to the control panel if a pod is not fully functional. Based on that information, the Master can then decide how to allocate tasks and resources to reach the desired state.

#### **Container Runtime**

The **container runtime** pulls images from a **container image registry** and starts and stops containers. A 3rd party software or plugin, such as Docker, usually performs this function.

#### **Kube-proxy**

The **kube-proxy** makes sure that each node gets its IP address, implements local iptables and rules to handle routing and [traffic load-balancing](https://phoenixnap.com/kb/load-balancing).

### **Container Networking**

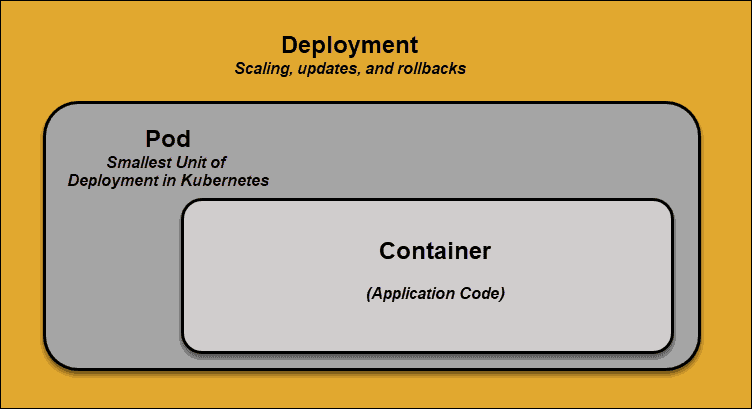
Container networking enables containers to communicate with hosts or other containers. It is often achieved by using the container networking interface (CNI), which is a joint initiative by Kubernetes, Apache Mesos, Cloud Foundry, Red Hat OpenShift, and others.

CNI offers a standardized, minimal specification for network connectivity in containers. You can use the CNI plugin by passing the kubelet --network-plugin=cni command-line option. The kubelet can then read files from --cni-conf-dir and use the CNI configuration when setting up networking for each pod.

#### **Pod**

A **pod** is the smallest element of scheduling in Kubernetes. Without it, a container cannot be part of a cluster. If you need to scale your app, you can only do so by adding or removing pods.

The pod serves as a ‘wrapper’ for a single container with the application code. Based on the availability of resources, the Master schedules the pod on a specific node and coordinates with the container runtime to launch the container.



**Kubernetes Networking:**

<https://kubernetes.io/docs/concepts/cluster-administration/networking/>

**Basics**: <https://kubernetes.io/docs/tutorials/kubernetes-basics/>

**Service:**

An abstract way to expose an application running on a set of [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) as a network service.

With Kubernetes you don't need to modify your 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.

**Types of Service (Networking in Kubernetes)**

Cluster IP: A ClusterIP service is the default type of service in Kubernetes. It creates a service inside the Kubernetes cluster, which can be accessed by other applications in the cluster, without allowing external access.

LoadBalancer: Exposes the Service externally using a cloud provider's load balancer. NodePort and ClusterIP Services, to which the external load balancer routes, are automatically created.

NodePort: A NodePort service opens a specific port from the Ranges 30000-32767on all the Nodes in the cluster, and any traffic sent to that port is forwarded to the service. The service cannot be accessed from the cluster IP.

Ingress: [Ingress](https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.23/#ingress-v1-networking-k8s-io) exposes HTTP and HTTPS routes from outside the cluster to [services](https://kubernetes.io/docs/concepts/services-networking/service/) within the cluster. Traffic routing is controlled by rules defined on the Ingress resource. Ingress is actually not a type of service. It sits in front of multiple services and performs smart routing between them, providing access to your cluster. There are several types of ingress controllers that have different routing capabilities. In GKE, the ingress controller creates an HTTP Load Balancer, which can route traffic to services in the Kubernetes cluster based on path or subdomain.

Headless: Sometimes you don't need load-balancing and a single Service IP. In this case, you can create what are termed "headless" Services, by explicitly specifying "None" for the cluster IP (.spec.clusterIP).

You can use a headless Service to interface with other service discovery mechanisms, without being tied to Kubernetes' implementation.

With selectors[:](https://kubernetes.io/docs/concepts/services-networking/service/#with-selectors) For headless Services that define selectors, the endpoints controller creates Endpoints records in the API, and modifies the DNS configuration to return A records (IP addresses) that point directly to the Pods backing the Service.

Without selectors: For headless Services that do not define selectors, the endpoints controller does not create Endpoints records. However, the DNS system looks for and configures either:

CNAME records for [ExternalName](https://kubernetes.io/docs/concepts/services-networking/service/#externalname)-type Services.

A records for any Endpoints that share a name with the Service, for all other types

[**ExternalName**](https://kubernetes.io/docs/concepts/services-networking/service/#externalname)**:** Maps the Service to the contents of the externalName field (e.g. foo.bar.example.com), by returning a CNAME record with its value. No proxying of any kind is set up.

