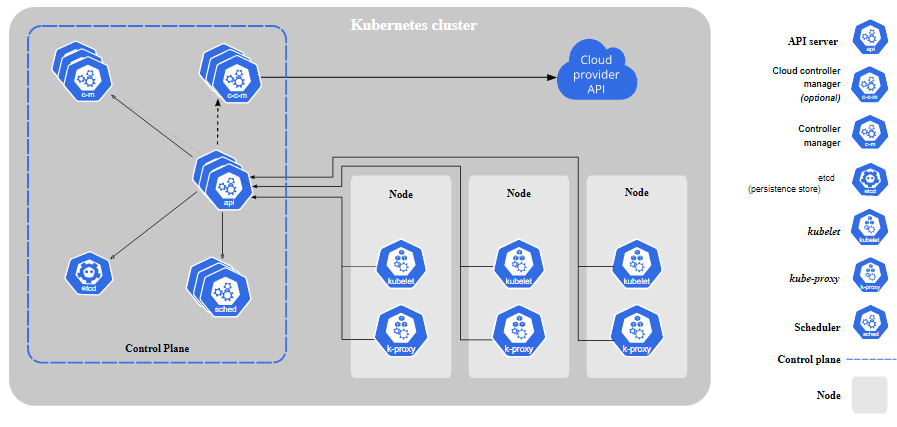
**Kubernetes Cluster Architecture**

# Kubernetes Components

When you deploy Kubernetes, you get a cluster.

A Kubernetes cluster consists of a set of worker machines, called [nodes](https://kubernetes.io/docs/concepts/architecture/nodes/), that run containerized applications. Every cluster has at least one worker node.

The worker node(s) host the [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) that are the components of the application workload. The [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-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.



**Cluster Architecture**

## Control Plane Components:

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events (for example, starting up a new [pod](https://kubernetes.io/docs/concepts/workloads/pods/) when a deployment's replicas field is unsatisfied).

Control plane components can be run on any machine in the cluster.

### kube-apiserver

The API server is a component of the Kubernetes [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) that exposes the Kubernetes API. The API server is the front end for the Kubernetes control plane and the core of Kubernetes' [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) . The API server exposes an HTTP API 443 that lets end users, different parts of your cluster, and external components communicate with one another.

The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes (for example: Pods, Namespaces, ConfigMaps, and Events).

**etcd**

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster data

etcd stores data in a multiversion [persistent](https://en.wikipedia.org/wiki/Persistent_data_structure) key-value store. The persistent key-value store preserves the previous version of a key-value pair when its value is superseded with new data. The key-value store is effectively immutable which means all the past versions of keys are still accessible and watchable after modification.

### kube-scheduler

In Kubernetes, scheduling refers to making sure that [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) are matched to [Nodes](https://kubernetes.io/docs/concepts/architecture/nodes/) so that [Kubelet](https://kubernetes.io/docs/reference/generated/kubelet" \t "_blank) can run them.

For every newly created pod or other unscheduled pods, kube-scheduler selects an optimal node for them to run on. However, every container in pods has different requirements for resources and every pod also has different requirements. Therefore, existing nodes need to be filtered according to the specific scheduling requirements.

kube-scheduler selects a node for the pod in a 2-step operation:

1. Filtering
2. Scoring

The filtering step finds the set of Nodes where it's feasible to schedule the Pod.  If none of the nodes are suitable, the pod remains unscheduled until the scheduler is able to place it.

The scheduler finds feasible Nodes for a Pod and then runs a set of functions to score the feasible Nodes and picks a Node with the highest score among the feasible ones to run the Pod. The scheduler then notifies the API server about this decision in a process called *binding*.

### kube-controller-manager

Control Plane component that runs [controller](https://kubernetes.io/docs/concepts/architecture/controller/) processes.

In Kubernetes, controllers are like the control loops that watch the state of your [cluster](https://kubernetes.io/docs/reference/glossary/?all=true#term-cluster), then make or request changes where needed. Each controller tries to move the current cluster state closer to the desired state based on the object spec field

## Controller pattern

Some types of these controllers are:

* Node controller: Responsible for noticing and responding when nodes go down.
* Job controller: Watches for Job objects that represent one-off tasks, then creates Pods to run those tasks to completion.
* Endpoints controller: Populates the Endpoints object (that is, joins Services & Pods).
* Service Account & Token controllers: Create default accounts and API access tokens for new namespaces.

### cloud-controller-manager

A Kubernetes [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) component that embeds cloud-specific control logic. The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that just interact with your cluster.

## Node Components:

Node components run on every node, maintaining running pods and providing the Kubernetes runtime environment.

### Kubelet:

An agent that runs on each [node](https://kubernetes.io/docs/concepts/architecture/nodes/) in the cluster. It makes sure that [containers](https://kubernetes.io/docs/concepts/containers/) are running in a [Pod](https://kubernetes.io/docs/concepts/workloads/pods/).

The kubelet takes a set of PodSpecs that are provided through various mechanisms and ensures that the containers described in those PodSpecs are running and healthy. The kubelet doesn't manage containers which were not created by Kubernetes.

### kube-proxy:

kube-proxy is a network proxy that runs on each [node](https://kubernetes.io/docs/concepts/architecture/nodes/) in your cluster, implementing part of the Kubernetes [Service](https://kubernetes.io/docs/concepts/services-networking/service/) concept.

[kube-proxy](https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/) maintains network rules on nodes. These network rules allow network communication to your Pods from network sessions inside or outside of your 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.

### Container runtime:

The container runtime is the software that is responsible for running containers.

Kubernetes supports several container runtimes: [Docker](https://docs.docker.com/engine/" \o "" \t "_blank), [containerd](https://containerd.io/docs/" \o "" \t "_blank), [CRI-O](https://cri-o.io/#what-is-cri-o), and any implementation of the [Kubernetes CRI (Container Runtime Interface](https://github.com/kubernetes/community/blob/master/contributors/devel/sig-node/container-runtime-interface.md)).

## Management

There are two main ways to have Nodes added to the [API server](https://kubernetes.io/docs/concepts/overview/components/#kube-apiserver):

1. The kubelet on a node self-registers to the control plane
2. You, or another human user, manually add a Node object

**Other Components:**

### DNS

All Kubernetes clusters should have [cluster DNS](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/), Cluster DNS is a DNS server, in addition to the other DNS server(s) in your environment, which serves DNS records for Kubernetes services.

Containers started by Kubernetes automatically include this DNS server in their DNS searches)

### Web UI (Dashboard)

[Dashboard](https://kubernetes.io/docs/tasks/access-application-cluster/web-ui-dashboard/) is a general purpose, web-based UI for Kubernetes clusters. It allows users to manage and troubleshoot applications running in the cluster, as well as the cluster itself.

### Container Resource Monitoring

[Container Resource Monitoring](https://kubernetes.io/docs/tasks/debug-application-cluster/resource-usage-monitoring/) records generic time-series metrics about containers in a central database, and provides a UI for browsing that data.

### Cluster-level Logging

A [cluster-level logging](https://kubernetes.io/docs/concepts/cluster-administration/logging/) mechanism is responsible for saving container logs to a central log store with search/browsing interfac

The Control Plane is responsible for managing the cluster i.e. the Cluster and Nodes hosting application containers.

Communications Channel:

CP to Node: Kuberenetes API

# The Kubernetes API

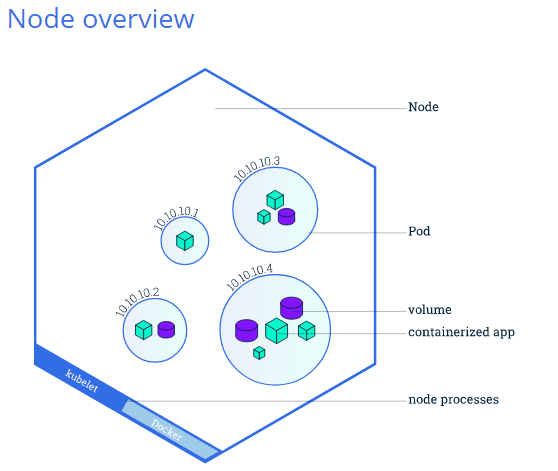
The core of Kubernetes' [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) is the [API server](https://kubernetes.io/docs/concepts/overview/components/#kube-apiserver). The API server exposes an HTTP API that lets end users, different parts of your cluster, and external components communicate with one another.

The Kubernetes API lets you query and manipulate the state of API objects in Kubernetes (for example: Pods, Namespaces, ConfigMaps, and Events).

**Nodes:**

Kubernetes runs your workload by placing containers into Pods to run on Nodes. A node may be a virtual or physical machine, depending on the cluster. Each node is managed by the [control plane](https://kubernetes.io/docs/reference/glossary/?all=true#term-control-plane) and contains the services necessary to run [Pods](https://kubernetes.io/docs/concepts/workloads/pods/).

The nodes communicate with the control plane using the Kubernetes API.



Node: A host for the single or multiple Pods.

Pods: Pods are the atomic unit on the Kubernetes platform. When we create a Deployment on Kubernetes, that Deployment creates Pods with containers inside them (as opposed to creating containers directly).

Each Pod is tied to the Node where it is scheduled, and remains there until termination (according to restart policy) or deletion.

In case of a Node failure, identical Pods are scheduled on other available Nodes in the cluster. Each PD has a private IP which enables Pod communication with other Pods in same Node as well as across the other Nodes in the Cluster. Kubernetes master automatically handles scheduling the pods across the Nodes in the cluster

Node Processes:

* Kubelet, a process responsible for communication between the Kubernetes Master and the Node; it manages the Pods and the containers running on a machine.
* A container runtime (like Docker) responsible for pulling the container image from a registry, unpacking the container, and running the application.

Although each Pod has a unique IP address, those IPs are not exposed outside the cluster which means application running inside the Pod Containers needs a method/provision to expose it to the outside world.

**Service:** A concept known as Service enables the application exposure to outside the cluster. Services allow your applications to receive traffic. A Service is defined using YAML [(preferred)](https://kubernetes.io/docs/concepts/configuration/overview/#general-configuration-tips) or JSON, like all Kubernetes objects.

Pods are not constant. One of the best features Kubernetes offers is that non-functioning pods get replaced by new ones automatically.

However, these new pods have a different set of IPs. It can lead to processing issues, and IP churn as the IPs no longer match. If left unattended, this property would make pods highly unreliable.

Services are introduced to provide reliable networking by bringing stable IP addresses and DNS names to the unstable world of pods.

By controlling traffic coming and going to the pod, a Kubernetes service provides a stable networking endpoint – a fixed IP, DNS, and port. Through a service, any pod can be added or removed without the fear that basic network information would change in any way.

#### How Do Kubernetes Services Work?

Pods are associated with services through key-value pairs called **labels** and **selectors**. A service automatically discovers a new pod with labels that match the selector.

This process seamlessly adds new pods to the service, and at the same time, removes terminated pods from the cluster.

For example, if the desired state includes **three replicas of a pod** and a node running **one replica fails**, the current state is reduced to two pods. Kubernetes observers that the desired state is three pods. It then **schedules one new replica** to take the place of the failed pod and assigns it to another node in the cluster.

The same would apply when updating or scaling the application by adding or removing pods. Once we update the desired state, Kubernetes notices the discrepancy and adds or removes pods to match the manifest file. The Kubernetes control panel records, implements, and runs background reconciliation loops that continuously check to see if the environment matches user-defined requirements.

Services can be exposed in different ways by specifying a type in the ServiceSpec:

* *ClusterIP* (default) - Exposes the Service on an internal IP in the cluster. This type makes the Service only reachable from within the cluster.
* *NodePort* - Exposes the Service on the same port of each selected Node in the cluster using NAT. Makes a Service accessible from outside the cluster using <NodeIP>:<NodePort>. Superset of ClusterIP.
* *LoadBalancer* - Creates an external load balancer in the current cloud (if supported) and assigns a fixed, external IP to the Service. Superset of NodePort.
* *ExternalName* - Exposes the Service using an arbitrary name (specified by externalName in the spec) by returning a CNAME record with the name. No proxy is used. This type requires v1.7 or higher of kube-dns.

Choosing the right deployment procedure depends on the needs, we listed below some of the possible strategies to adopt:

* [recreate](https://blog.container-solutions.com/kubernetes-deployment-strategies#kubernetes-recreate): terminate the old version and release the new one
* [ramped](https://blog.container-solutions.com/kubernetes-deployment-strategies#kubernetes-ramped): release a new version on a rolling update fashion, one after the other
* [blue/green](https://blog.container-solutions.com/kubernetes-deployment-strategies#kubernetes-blue-green): release a new version alongside the old version then switch traffic
* [canary](https://blog.container-solutions.com/kubernetes-deployment-strategies#kubernetes-canary): release a new version to a subset of users, then proceed to a full rollout
* [a/b testing](https://blog.container-solutions.com/kubernetes-deployment-strategies#kubernetes-a-b-testing): release a new version to a subset of users in a precise way (HTTP headers, cookie, weight, etc.). A/B testing is really a technique for making business decisions based on statistics but we will briefly describe the process. This doesn’t come out of the box with Kubernetes, it implies extra work to setup a more advanced infrastructure ([Istio](https://www.istio.io/), [Linkerd](https://linkerd.io/), [Traefik](https://traefik.io/), custom nginx/haproxy, etc).

<https://github.com/ContainerSolutions/k8s-deployment-strategies>

# Cluster Networking

Networking is a central part of Kubernetes, but it can be challenging to understand exactly how it is expected to work. There are 4 distinct networking problems to address:

1. Highly-coupled container-to-container communications: this is solved by [Pods](https://kubernetes.io/docs/concepts/workloads/pods/) and localhost communications.
2. Pod-to-Pod communications: this is the primary focus of this document.
3. Pod-to-Service communications: this is covered by [services](https://kubernetes.io/docs/concepts/services-networking/service/).
4. External-to-Service communications: this is covered by [services](https://kubernetes.io/docs/concepts/services-networking/service/).

Kubernetes is all about sharing machines between applications. Typically, sharing machines requires ensuring that two applications do not try to use the same ports. Coordinating ports across multiple developers is very difficult to do at scale and exposes users to cluster-level issues outside of their control.

Dynamic port allocation brings a lot of complications to the system - every application has to take ports as flags, the API servers have to know how to insert dynamic port numbers into configuration blocks, services have to know how to find each other, etc. Rather than deal with this, Kubernetes takes a different approach.

## The Kubernetes network model:

Every Pod gets its own IP address. This means you do not need to explicitly create links between Pods and you 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 (barring any intentional network segmentation policies):

* pods on a node can communicate with all pods on all nodes without NAT
* agents on a node (e.g. system daemons, kubelet) can communicate with all pods on that node

Note: For those platforms that support Pods running in the host network (e.g. Linux):

* pods in the host network of a node can communicate with all pods on all nodes without NAT

This model is not only less complex overall, but it is principally compatible with the desire for Kubernetes to enable low-friction porting of apps from VMs to containers. If your job previously ran in a VM, your VM had an IP and could talk to other VMs in your project. This is the same basic model.

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.

How this is implemented is a detail of the particular container runtime in use.

It is possible to request ports on the Node itself which forward to your Pod (called host ports), but this is a very niche operation. How that forwarding is implemented is also a detail of the container runtime. The Pod itself is blind to the existence or non-existence of host port

Kubernetes uses some third party tools mentioned below to manage this networking model.

ACI - [Cisco Application Centric Infrastructure](https://www.cisco.com/c/en/us/solutions/data-center-virtualization/application-centric-infrastructure/index.html)

Antrea

AOS from Apstra

### AWS VPC CNI for Kubernetes

### Azure CNI for Kubernetes

### Google Compute Engine (GCE)

### Calico

[Calico](https://docs.projectcalico.org/) is an open source networking and network security solution for containers, virtual machines, and native host-based workloads.

**Flannel**

[Flannel](https://github.com/coreos/flannel#flannel) is a very simple overlay network that satisfies the Kubernetes requirements

### Installing a Pod network add-on

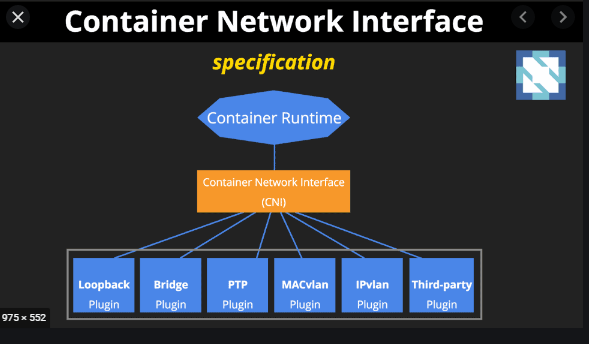
**Caution:**

This section contains important information about networking setup and deployment order. Read all of this advice carefully before proceeding.

**You must deploy a**[Container Network Interface](https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/#cni)**(CNI) based Pod network add-on so that your Pods can communicate with each other. Cluster DNS (CoreDNS) will not start up before a network is installed.**

* Take care that your Pod network must not overlap with any of the host networks: you are likely to see problems if there is any overlap. (If you find a collision between your network plugin's preferred Pod network and some of your host networks, you should think of a suitable CIDR block to use instead, then use that during kubeadm init with --pod-network-cidr and as a replacement in your network plugin's YAML).
* By default, kubeadm sets up your cluster to use and enforce use of [RBAC](https://kubernetes.io/docs/reference/access-authn-authz/rbac/) (role based access control). Make sure that your Pod network plugin supports RBAC, and so do any manifests that you use to deploy it.

CNI:



**Note:** Currently Calico is the only CNI plugin that the kubeadm project performs e2e tests agains

install a Pod network add-on with the following command on the control-plane node or a node that has the kubeconfig credentials:

kubectl apply -f <add-on.yaml>

You can install only one Pod network per cluster.

Once a Pod network has been installed, you can confirm that it is working by checking that the CoreDNS Pod is Running in the output of kubectl get pods --all-namespaces. And once the CoreDNS Pod is up and running, you can continue by joining your nodes.

# 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.

## Service resources

In Kubernetes, a Service is an abstraction which defines a logical set of Pods and a policy by which to access them (sometimes this pattern is called a micro-service). The set of Pods targeted by a Service is usually determined by a [selector](https://kubernetes.io/docs/concepts/overview/working-with-objects/labels/)

## Defining a Service

A Service in Kubernetes is a REST object, similar to a Pod. Like all of the REST objects, you can POST a Service definition to the API server to create a new instance. The name of a Service object must be a valid [DNS label name](https://kubernetes.io/docs/concepts/overview/working-with-objects/names#dns-label-names).

For example, suppose you have a set of Pods where each listens on TCP port 9376 and contains a label app=MyApp:

**apiVersion**: v1

**kind**: Service

**metadata**:

**name**: my-service

**spec**:

**selector**:

**app**: MyApp

**ports**:

- **protocol**: TCP

**port**: 80

**targetPort**: 9376

This specification creates a new Service object named "my-service", which targets TCP port 9376 on any Pod with the app=MyApp label.

Kubernetes assigns this Service an IP address (sometimes called the "cluster IP"), which is used by the Service proxies (see [Virtual IPs and service proxies](https://kubernetes.io/docs/concepts/services-networking/service/#virtual-ips-and-service-proxies) below).

The controller for the Service selector continuously scans for Pods that match its selector, and then POSTs any updates to an Endpoint object also named "my-service".

### DNS

You can (and almost always should) set up a DNS service for your Kubernetes cluster using an [add-on](https://kubernetes.io/docs/concepts/cluster-administration/addons/).

A cluster-aware DNS server, such as CoreDNS, watches the Kubernetes API for new Services and creates a set of DNS records for each one. If DNS has been enabled throughout your cluster then all Pods should automatically be able to resolve Services by their DNS name.

For example, if you have a Service called my-service in a Kubernetes namespace my-ns, the control plane and the DNS Service acting together create a DNS record for my-service.my-ns. Pods in the my-ns namespace should be able to find the service by doing a name lookup for my-service (my-service.my-ns would also work).

Pods in other namespaces must qualify the name as my-service.my-ns. These names will resolve to the cluster IP assigned for the Service.

Kubernetes also supports DNS SRV (Service) records for named ports. If the my-service.my-ns Service has a port named http with the protocol set to TCP, you can do a DNS SRV query for \_http.\_tcp.my-service.my-ns to discover the port number for http, as well as the IP address.

The Kubernetes DNS server is the only way to access ExternalName Services. You can find more information about ExternalName resolution in [DNS Pods and Services](https://kubernetes.io/docs/concepts/services-networking/dns-pod-service/)

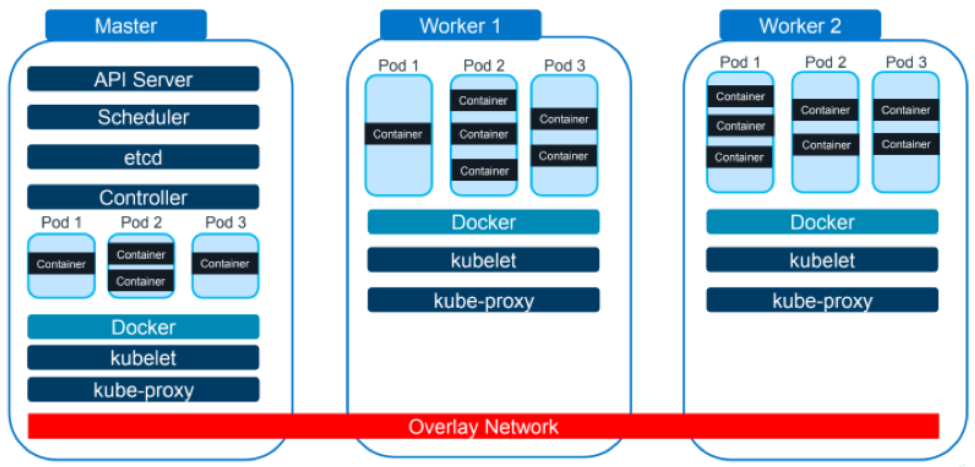
## Publishing Services (ServiceTypes)

Kubernetes ServiceTypes allow you to specify what kind of Service you want. The default is ClusterIP.

Type values and their behaviors are:

* ClusterIP: Exposes the Service on a cluster-internal IP. Choosing this value makes the Service only reachable from within the cluster. This is the default ServiceType.
* [NodePort](https://kubernetes.io/docs/concepts/services-networking/service/#nodeport): Exposes the Service on each Node's IP at a static port (the NodePort). A ClusterIP Service, to which the NodePort Service routes, is automatically created. You'll be able to contact the NodePort Service, from outside the cluster, by requesting <NodeIP>:<NodePort>.
* [LoadBalancer](https://kubernetes.io/docs/concepts/services-networking/service/#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.
* [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.

**KS8 Challenges:**



Docker is a container platform that’s ideal for running containers on a single piece of hardware or virtual machine (VM), however, as we see “Docker” appears in each section so are so many containers across the cluster in different nodes making the communication between containers a real challenge.

Overlay network spread across entire cluster helps a container in master to simply talk to a container in any other node.

## Challenges in Kubernetes Environment:

* Internal and external networks are isolated –

KS8 networking is unconventional despite the Overlay Network because of the internal and external networks are distinct from one another.

* IP addresses of pods and containers can change

Kubernetes intentionally isolates malfunctioning or failing nodes or pods in order to keep them from bringing down the entire application. This can result in frequently changing IP addresses between nodes. Services that rely on knowing a pod or container’s IP address then have to figure out what the new IP addresses are.

* No access control between microservices

When it comes to access control between microservices, it’s important for companies to realize that traffic flowing between Kubernetes nodes are also capable of flowing to an external physical box or VM. This can both eat up resources and weaken security. This traffic flow between nodes and pods across nodes is called East-West traffic.

* No application layer visibility

Inability to examine information at the application layer is a big problem. Without that visibility, enterprises can miss key opportunities to gather detailed analytics and actionable insights.

## Kubernetes and Cloud Security Requirements:

1. Security for north-south traffic

North-south traffic, traffic is flowing in and out of the Kubernetes cloud.

As mentioned before, companies need something placed above the Kubernetes cloud to watch the traffic. For example, a firewall, a DDoS mitigation system or anything else that can catch malicious traffic.

These things are also useful in terms of traffic management. So, if there’s traffic that needs to go to specific places, this is the ideal place to do that. The Ingress controller is also helpful in this area.

1. Access control between Microservices:

Traffic flow between Kubernetes nodes is also called east-west traffic.

When traffic flows between Kubernetes nodes or between Pods, this traffic can be sent over physical networks, virtual or overlay networks, or both. Managing this east-west traffic flow from one pod or container to another can become quite complex without some way of monitoring it.

Second, it can also present a serious security risk: attackers who gain access to one container can gain access to the entire internal network.

One Solution: A Secure Service Mesh can secure east-west traffic by acting as a proxy between containers to implement security rules, and is also able to help with scaling, load balancing, service monitoring and more.

Additionally, a service mesh can function inside the Kubernetes cloud, without sending traffic to a physical box or VM.

1. Encryption for east-west traffic
2. Application traffic analytics
3. Central controller for large deployments

# Options for Highly Available topology

* With stacked control plane nodes, where etcd nodes are colocated with control plane nodes
* With external etcd nodes, where etcd runs on separate nodes from the control plane

## Stacked etcd topology

A stacked HA cluster is a [topology](https://en.wikipedia.org/wiki/Network_topology) where the distributed data storage cluster provided by etcd is stacked on top of the cluster formed by the nodes managed by kubeadm that run control plane components.

Each control plane node runs an instance of the kube-apiserver, kube-scheduler, and kube-controller-manager. The kube-apiserver is exposed to worker nodes using a load balancer.

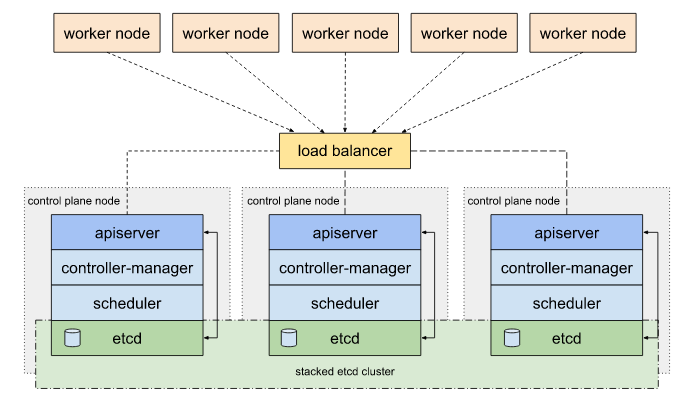
Each control plane node creates a local etcd member and this etcd member communicates only with the kube-apiserver of this node. The same applies to the local kube-controller-manager and kube-scheduler instances.

This topology couples the control planes and etcd members on the same nodes. It is simpler to set up than a cluster with external etcd nodes, and simpler to manage for replication.

However, a stacked cluster runs the risk of failed coupling. If one node goes down, both an etcd member and a control plane instance are lost, and redundancy is compromised. You can mitigate this risk by adding more control plane nodes.

You should therefore run a minimum of three stacked control plane nodes for an HA cluster.

This is the default topology in kubeadm. A local etcd member is created automatically on control plane nodes when using kubeadm init and kubeadm join --control-plane.



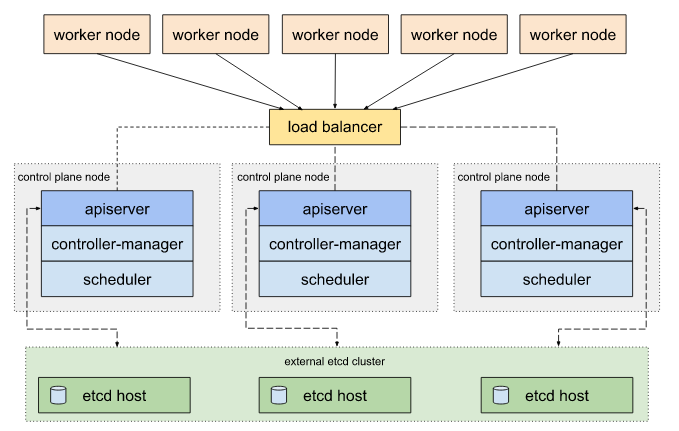
## External etcd topology

An HA cluster with external etcd is a [topology](https://en.wikipedia.org/wiki/Network_topology) where the distributed data storage cluster provided by etcd is external to the cluster formed by the nodes that run control plane components.

Like the stacked etcd topology, each control plane node in an external etcd topology runs an instance of the kube-apiserver, kube-scheduler, and kube-controller-manager. And the kube-apiserver is exposed to worker nodes using a load balancer. However, etcd members run on separate hosts, and each etcd host communicates with the kube-apiserver of each control plane node.

This topology decouples the control plane and etcd member. It therefore provides an HA setup where losing a control plane instance or an etcd member has less impact and does not affect the cluster redundancy as much as the stacked HA topology.

However, this topology requires twice the number of hosts as the stacked HA topology. A minimum of three hosts for control plane nodes and three hosts for etcd nodes are required for an HA cluster with this topology.



kubectl delete -n default service my-app