



Introduction to Kubernetes and Docker Containers



Preface

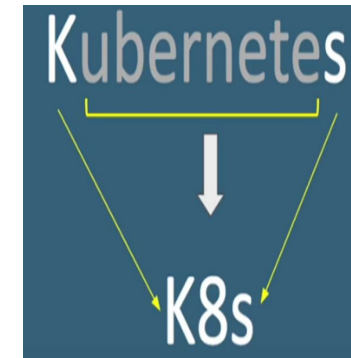
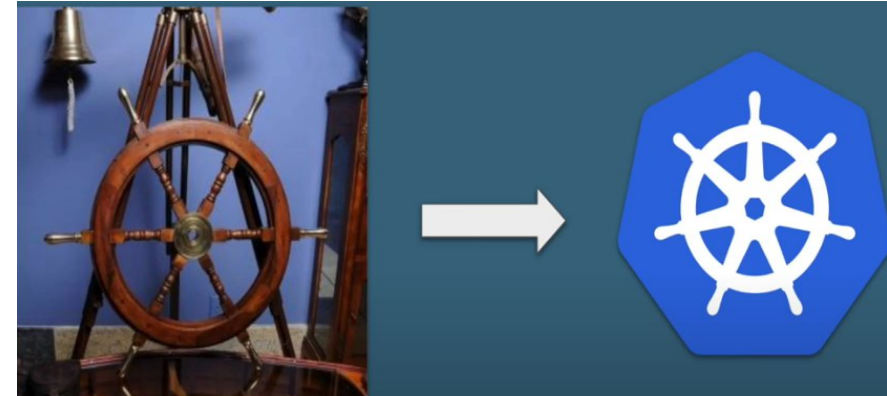
- Introduction to Kubernetes
 - Containers
 - Orchestration
 - Concepts of Dockers
 - Power of Kubernetes to deploy software on edge devices
- 
- 

Introduction to Kubernetes

- Kubernetes is the Greek word for helmsman or captain of a ship.
- Kubernetes also known as k-8 was built by Google based on their experience running containers in production
- It is now an open source project and is one of the best and most popular container orchestration technologies out there.
- As applications grow to span multiple containers deployed across multiple servers, **operating them becomes more complex**. To manage this complexity, Kubernetes provides an open source API that controls how and where those containers will run.

To understand Kubernetes first we need to understand two things:

- Container and
- Orchestration



Introduction to Kubernetes

Containers are isolated environments, have their own processes, services, networking interfaces, mounts similar to virtual machines except the fact that they all share the same operating system kernel.

Orchestration consists of a set of tools and scripts that can help host containers in a production environment. An orchestration consists of multiple container hosts that can host containers, if one fails the application is still accessible through the others.

Introduction to Kubernetes

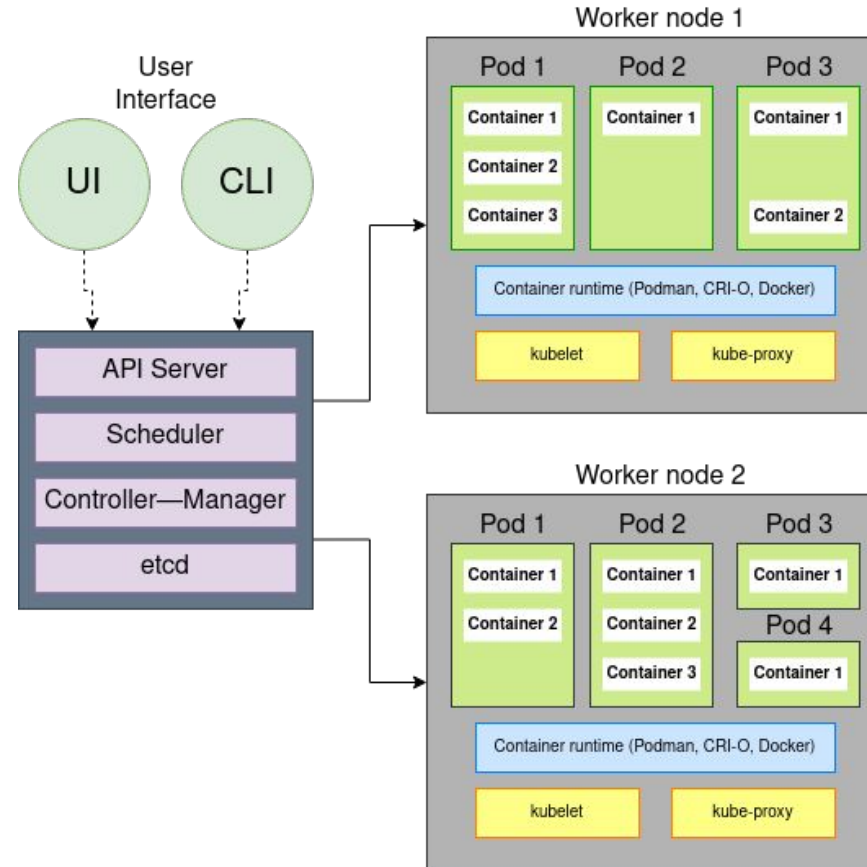
Kubernetes consists of one computer that gets designated as the control plane, and lots of other computers that get designated as worker nodes. Each of these has a complex but robust stack making orchestration possible,

Kubernetes **orchestrates** clusters of virtual machines and schedules containers to run on those virtual machines based on their available compute resources and the resource requirements of each container.

Kubernetes also automatically manages service discovery, incorporates load balancing, tracks resource allocation and scales based on compute utilization. And, it checks the health of individual resources and enables apps to self-heal by automatically restarting or replicating containers.

Now get familiar with each of the Kubernetes components:

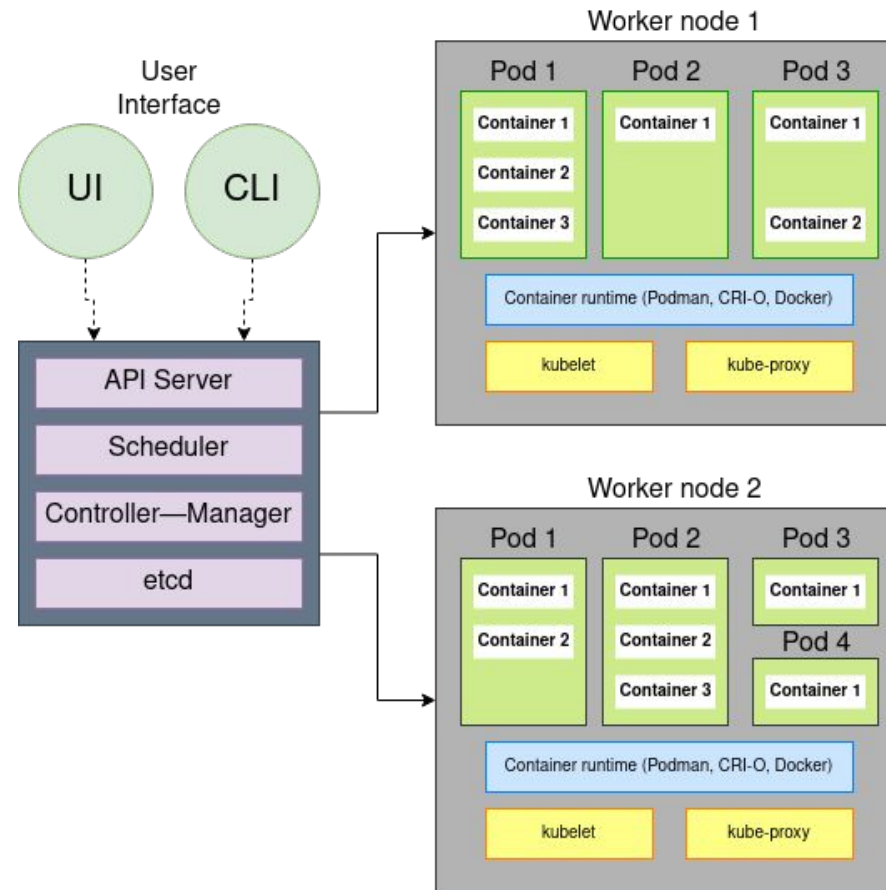
- Control plane component
- Worker node component



Kubernetes: Control plane Components

Etcd:

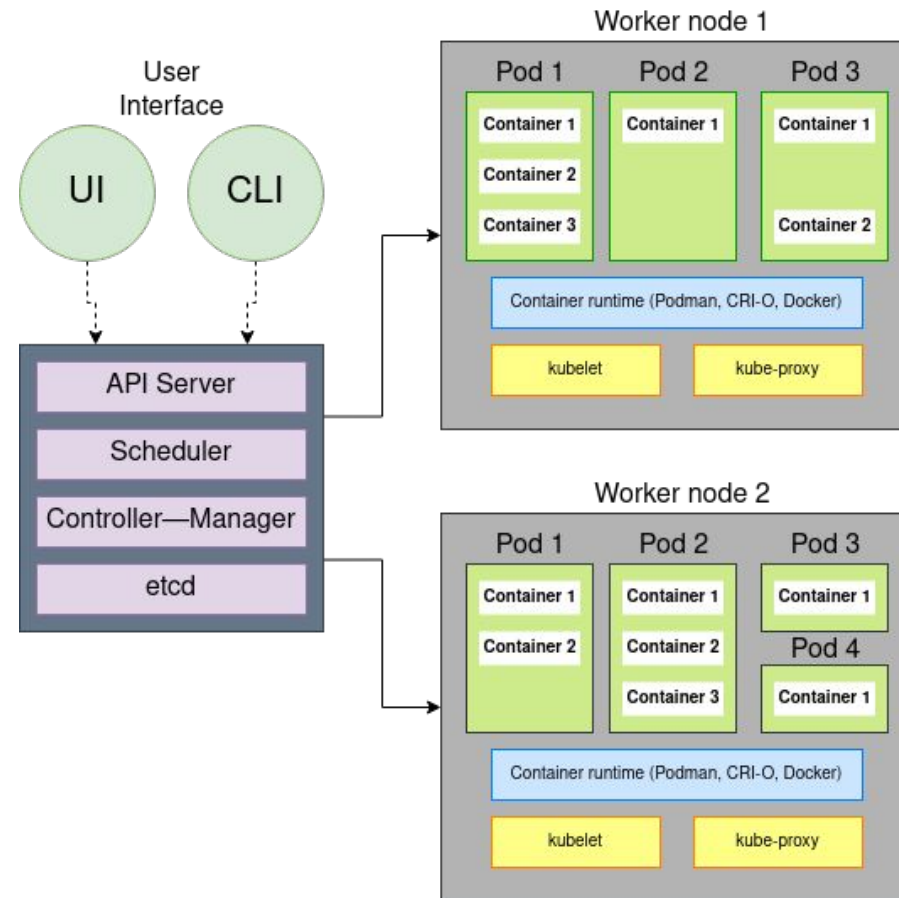
- Etcd is a fast, distributed, and consistent key-value store used as a backing store for persistently storing Kubernetes object data such as pods, replication controllers, secrets, and services.
- Etcd is the only place where Kubernetes stores cluster state and metadata. The only component that talks to etcd directly is the Kubernetes API server. All other components read and write data to etcd indirectly through the API server.
- Etcd also implements a watch feature, which provides an event-based interface for asynchronously monitoring changes to keys. Once you change a key, its watchers get notified. The API server component heavily relies on this to get notified and move the current state of etcd towards the desired state.



Kubernetes: Control plane Components

API Server:

- The API server is the only component in Kubernetes that directly interacts with etcd. All other components in Kubernetes must go through the API server to work with the cluster state, including the clients (kubectl). The API server has the following functions:
- Provides a consistent way of storing objects in etcd.
- Performs validation of those objects so clients can't store improperly configured objects.
- Provides a RESTful API to create, update, modify, or delete a resource.
- Performs authentication and authorization of a request that the client sends.
- Responsible for admission control if the request is trying to create, modify, or delete a resource. For example, AlwaysPullImages, DefaultStorageClass, and ResourceQuota.



Kubernetes: Control plane Components

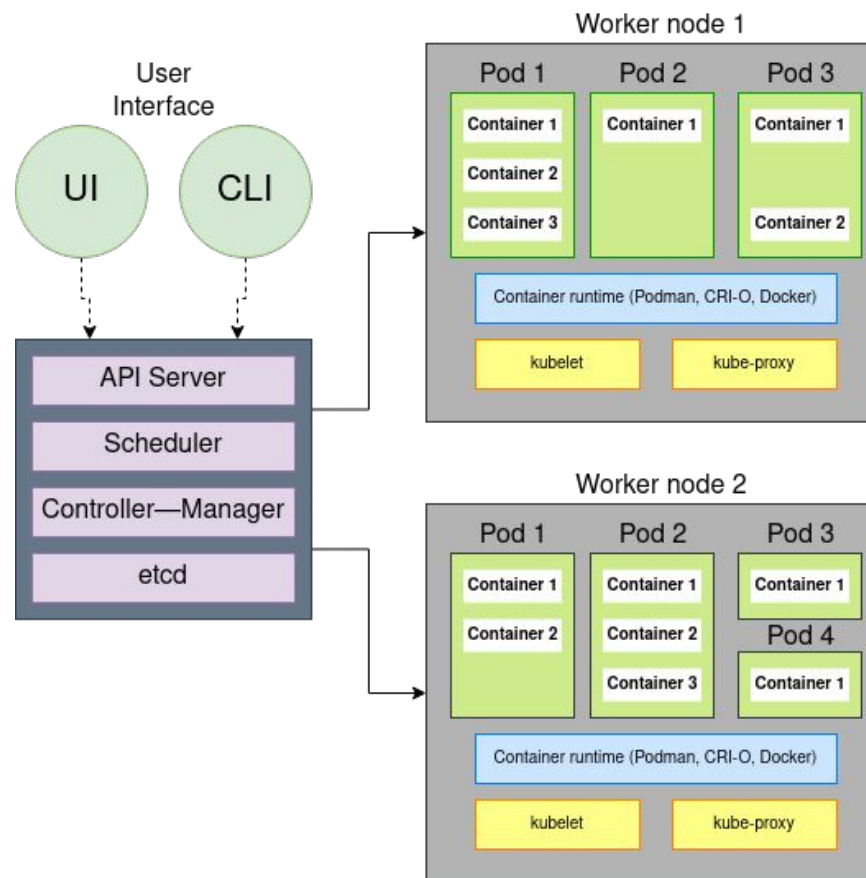
Controller Manager:

In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed.

Each controller tries to move the current cluster state closer to the desired state. The controller tracks at least one Kubernetes resource type, and these objects have a spec field that represents the desired state.

Controller examples:

- Node controller
- Service controller
- Endpoints controller
- Namespace controller
- Deployment controller
- StatefulSet controller



Kubernetes: Control plane Components

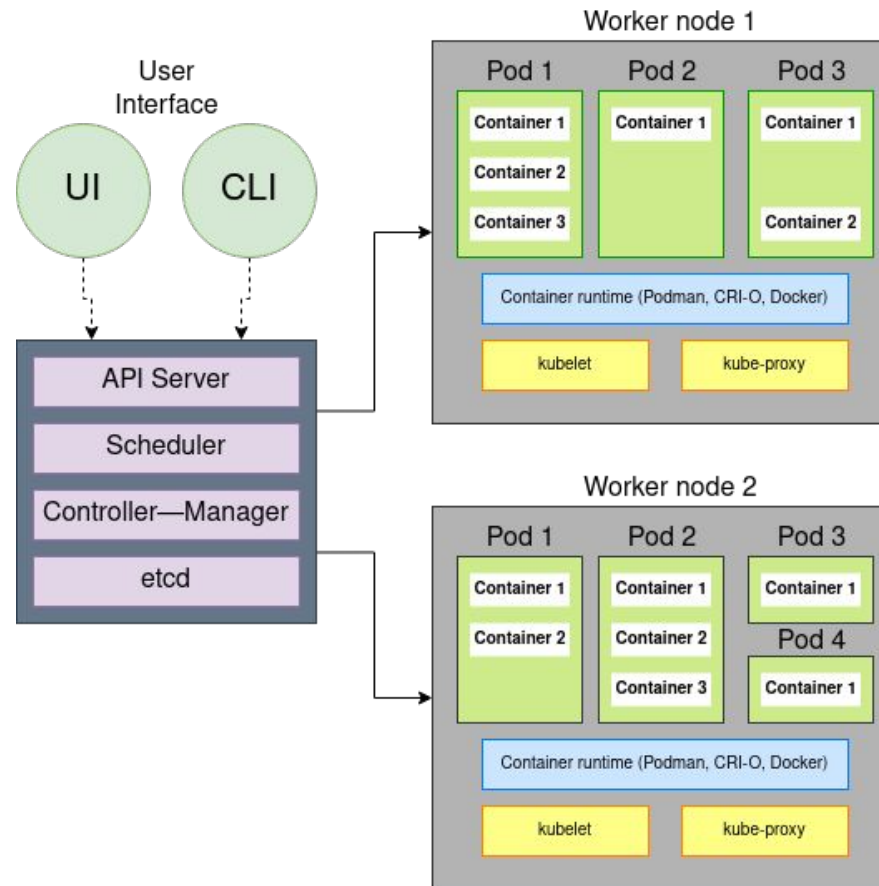
Controller Manager:

In Kubernetes, controllers are control loops that watch the state of your cluster, then make or request changes where needed.

Each controller tries to move the current cluster state closer to the desired state. The controller tracks at least one Kubernetes resource type, and these objects have a spec field that represents the desired state.

Controller examples:

- Node controller
- Service controller
- Endpoints controller
- Namespace controller
- Deployment controller
- StatefulSet controller



Kubernetes: Control plane Components

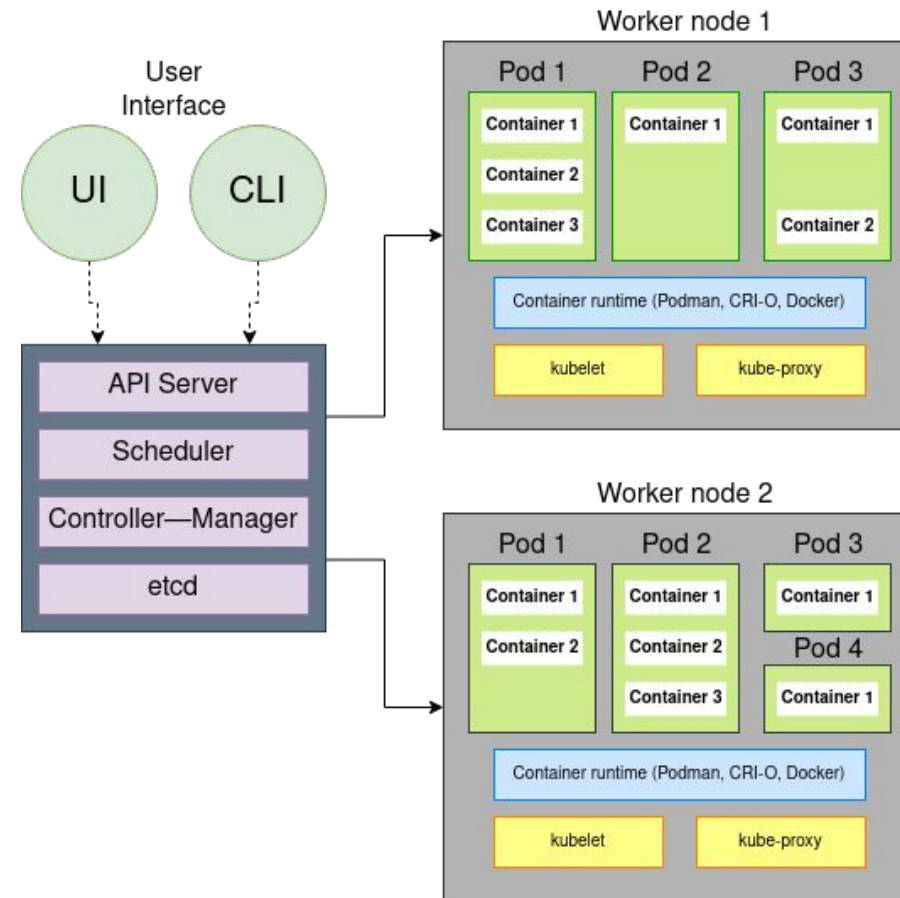
Scheduler:

The Scheduler is a control plane process that assigns pods to nodes. It watches for newly created pods that have no nodes assigned.

For every pod that the Scheduler discovers, the Scheduler becomes responsible for finding the best node for that pod to run on.

Nodes that meet the scheduling requirements for a pod get called feasible nodes. If none of the nodes are suitable, the pod remains unscheduled until the Scheduler can place it.

Once it finds a feasible node, it runs a set of functions to score the nodes, and the node with the highest score gets selected. It then notifies the API server about the selected node. They call this process binding.



Kubernetes: Worker node components

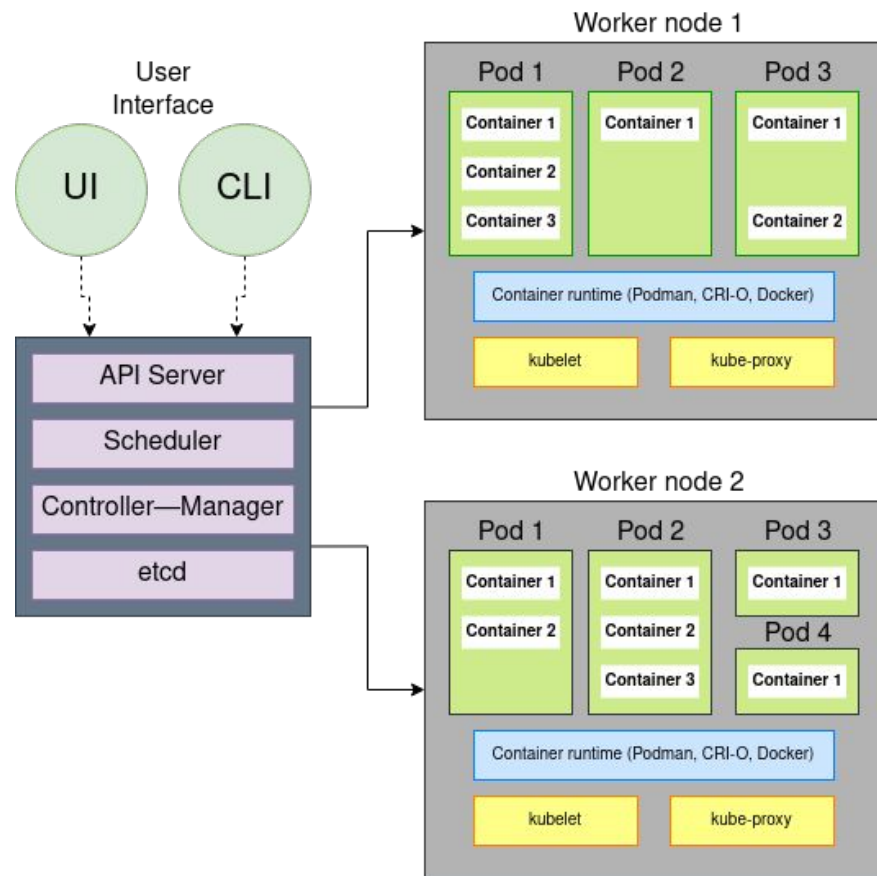
Kubelet:

Kubelet is an agent that runs on each node in the cluster and is responsible for everything running on a worker node.

It ensures that the containers run in the pod.

The main functions of kubelet service are:

- Register the node it's running on by creating a node resource in the API server.
- Continuously monitor the API server for pods that got scheduled to the node.
- Start the pod's containers by using the configured container runtime.
- Continuously monitor running containers and report their status, events, and resource consumption to the API server.
- Run the container liveness probes, restart containers when the probes fail and terminate containers when their pod gets deleted from the API server (notifying the server about the pod termination).



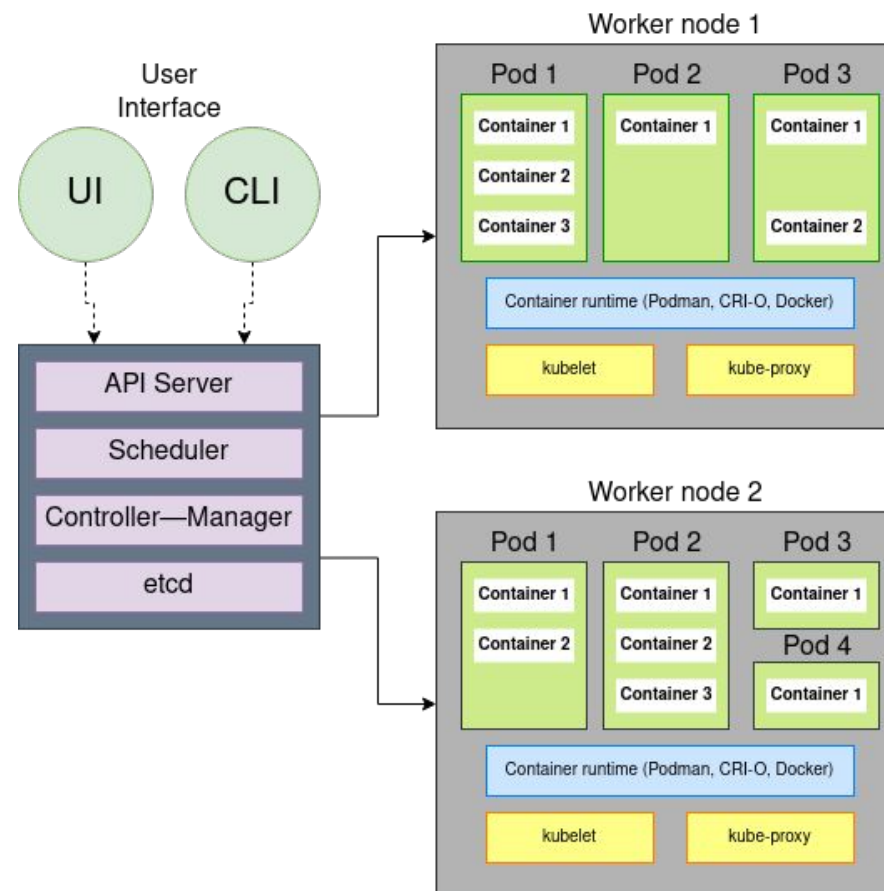
Kubernetes: Worker node components

Service proxy:

The service proxy (kube-proxy) runs on each node and ensures that one pod can talk to another pod, one node can talk to another node, and one container can talk to another container.

It is responsible for watching the API server for changes on services and pod definitions to maintain that the entire network configuration is up to date.

When a service gets backed by more than one pod, the proxy performs load balancing across those pods.



Kubernetes: Worker node components

Container runtime:

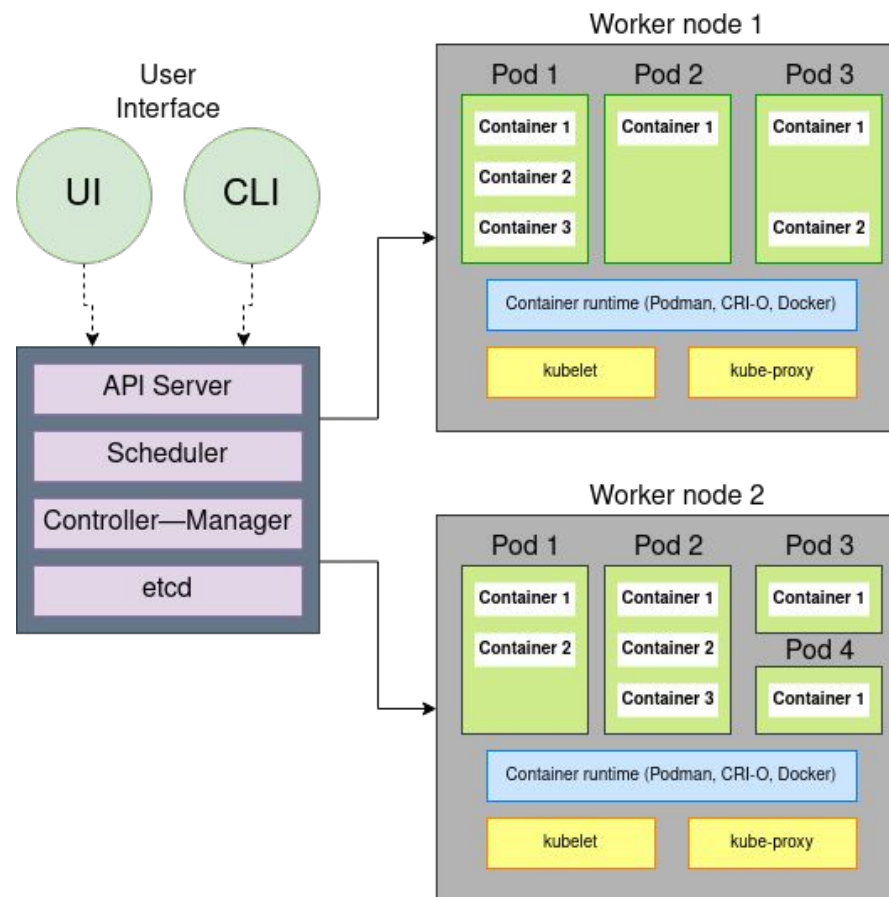
There are two categories of container runtimes:

Lower-level container runtimes: These focus on running containers and setting up the namespace and cgroups for containers.

Higher-level container runtimes (container engine): These focus on formats, unpacking, management, sharing of images, and providing APIs for developers.

Container runtime takes care of:

- Pulls the required container image from an image registry if it's not available locally.
- Prepares a container mount point.
- Alerts the kernel to assign some resource limits like CPU or memory limits.
- Pass system call (syscall) to the kernel to start the container.



Introduction to Docker

The most popular container technology out in the market is Docker container.

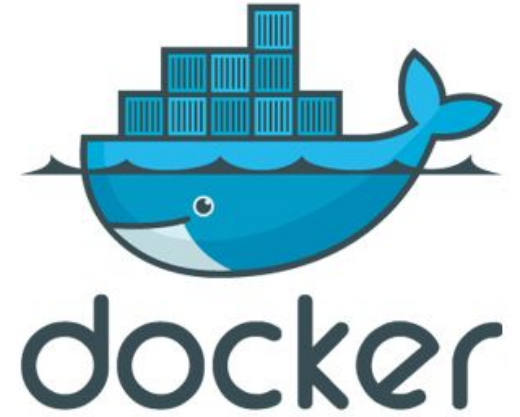
Requirement to set up an end-to-end stack including various different technologies like a web server using node.js, a database such as MongoDB, a messaging system like Redis and an orchestration tool like ansible.

checking compatibility between these various components and the underlying infrastructure this compatibility matrix issue is usually referred to as the matrix from hell.

with docker one can able to run these various component in a separate container with its own libraries and its own dependencies on the same VM and the OS but within separate environments or containers

It only requires to build the docker configuration with a simple docker run command irrespective of what the underlying operating system they run all they needed to do was to make sure they had docker installed on their systems

Introduction to Dockers



The most popular container technology out in the market is Docker container.

Docker is an open platform for developing, shipping, and running applications.

Docker enables you to separate your applications from your infrastructure so you can deliver software quickly.

With Docker, you can manage your infrastructure in the same ways you manage your applications.

By taking advantage of Docker's methodologies for shipping, testing, and deploying code quickly, you can significantly reduce the delay between writing code and running it in production.

Docker provides the ability to package and run an application in a loosely isolated environment called a container.

Docker Architecture

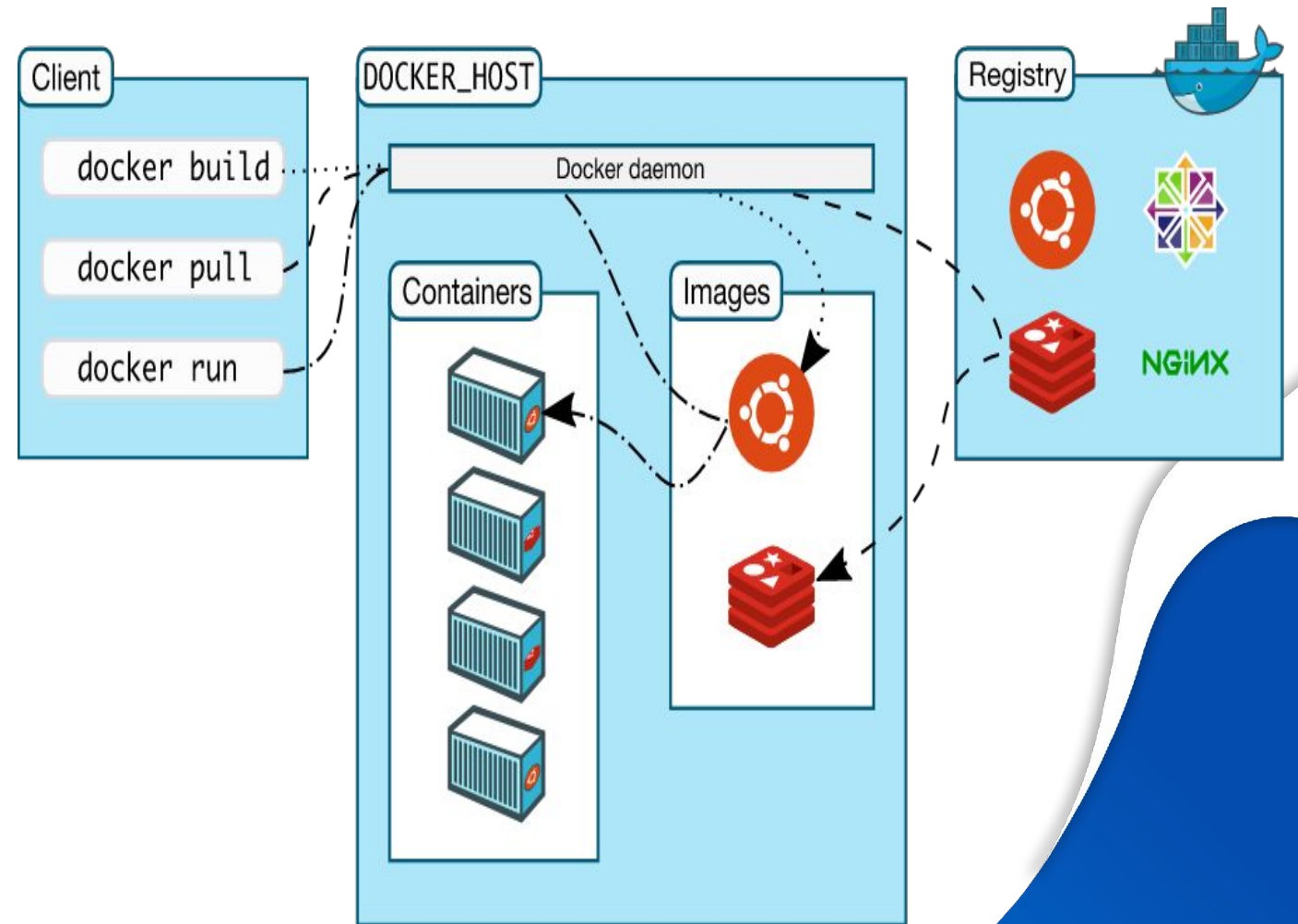
Docker uses a client-server architecture.

The Docker client talks to the Docker daemon, which does the heavy lifting of building, running, and distributing your Docker containers.

The Docker client and daemon can run on the same system, or you can connect a Docker client to a remote Docker daemon.

The Docker client and daemon communicate using a REST API, over UNIX sockets or a network interface.

Another Docker client is Docker Compose, that lets you work with applications consisting of a set of containers.



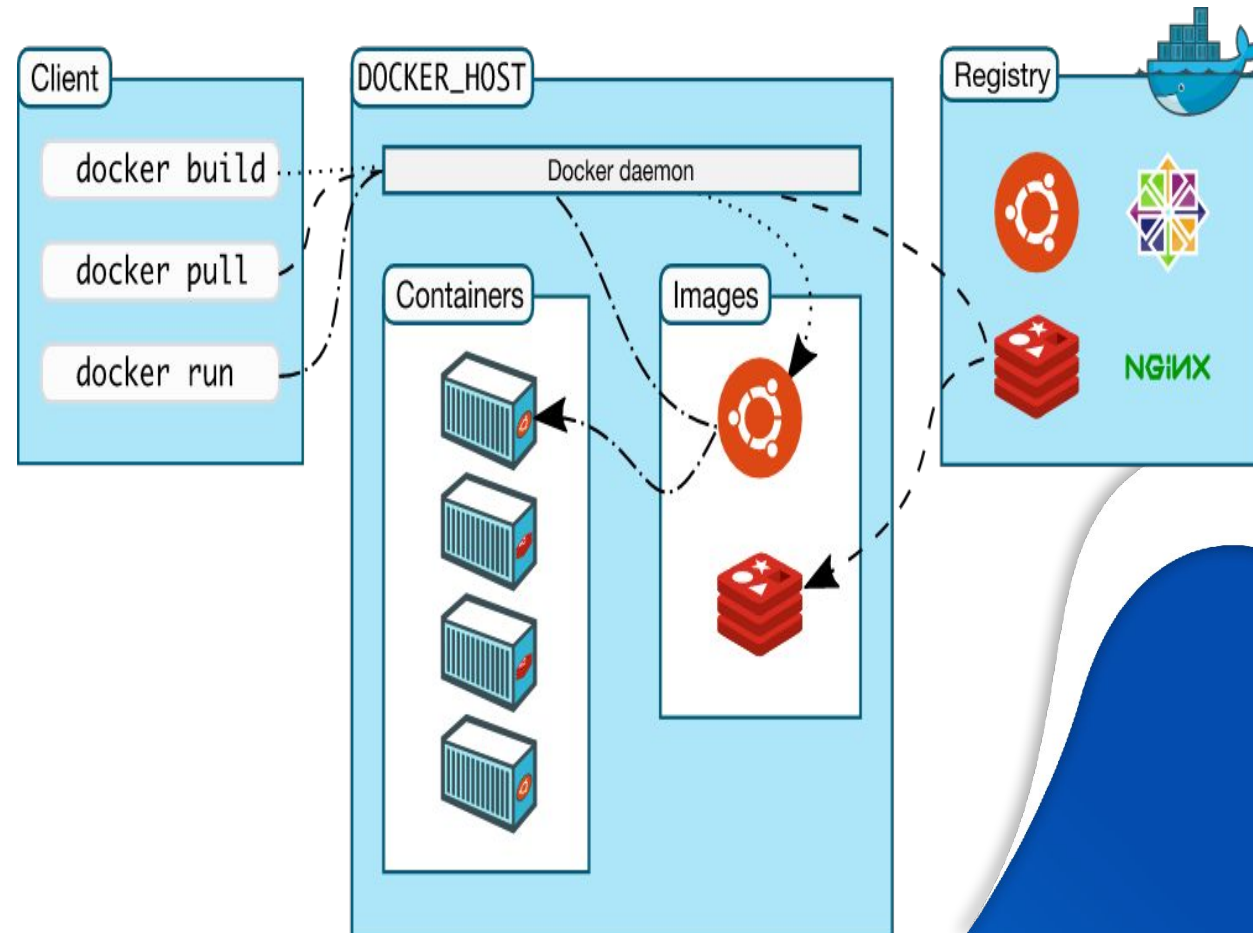
Docker Architecture: Components

The Docker daemon:

The Docker daemon listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.

The Docker client:

The Docker client is the primary way that many Docker users interact with Docker. When you use commands such as `docker run`, the client sends these commands to `dockerd`, which carries them out. The `docker` command uses the Docker API. The Docker client can communicate with more than one daemon.



Docker Architecture: Components

Docker registries:

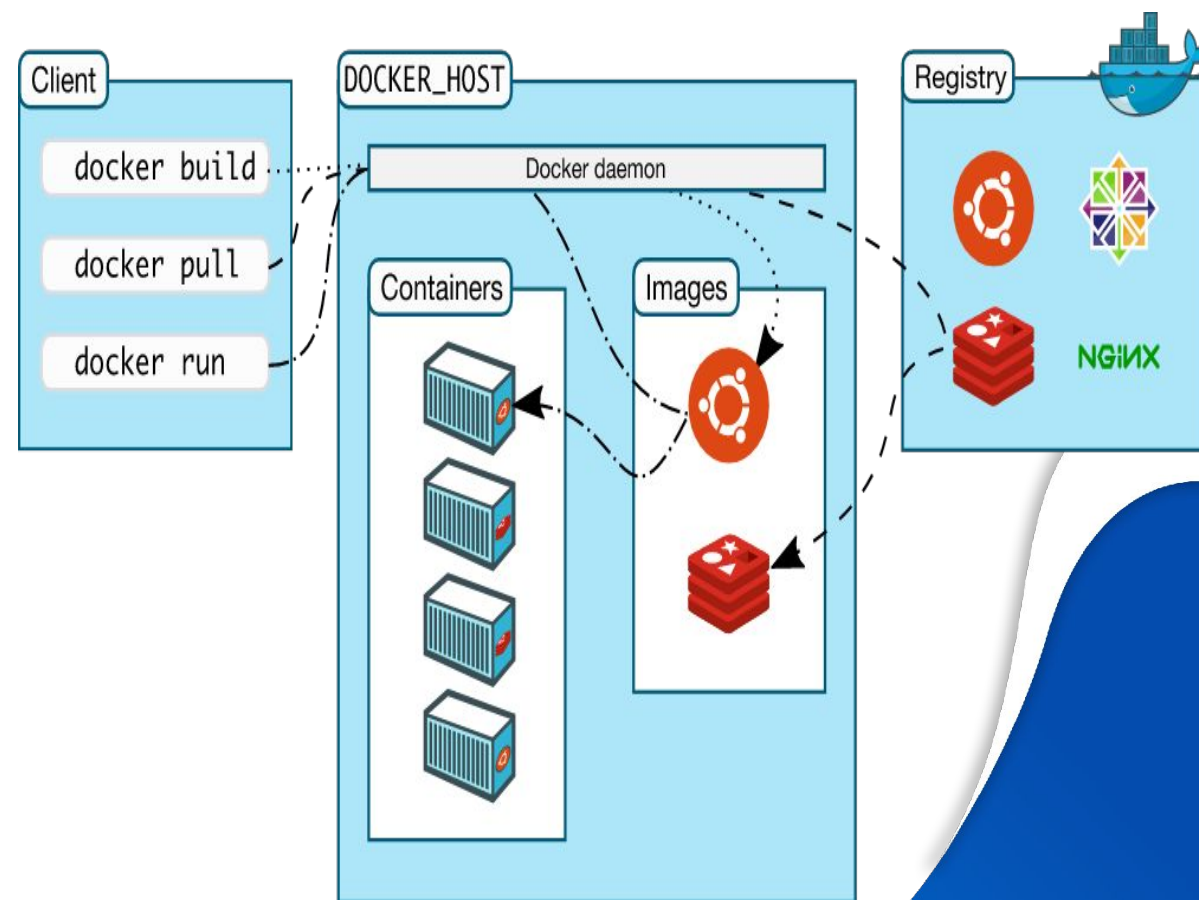
A Docker registry stores Docker images. Docker Hub is a public registry that anyone can use, and Docker is configured to look for images on Docker Hub by default. You can even run your own private registry.

Docker objects:

When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects.

Docker Desktop:

Docker Desktop includes the Docker daemon, the Docker client, Docker Compose, Docker Content Trust, Kubernetes, and Credential Helper. For more information, see Docker Desktop.

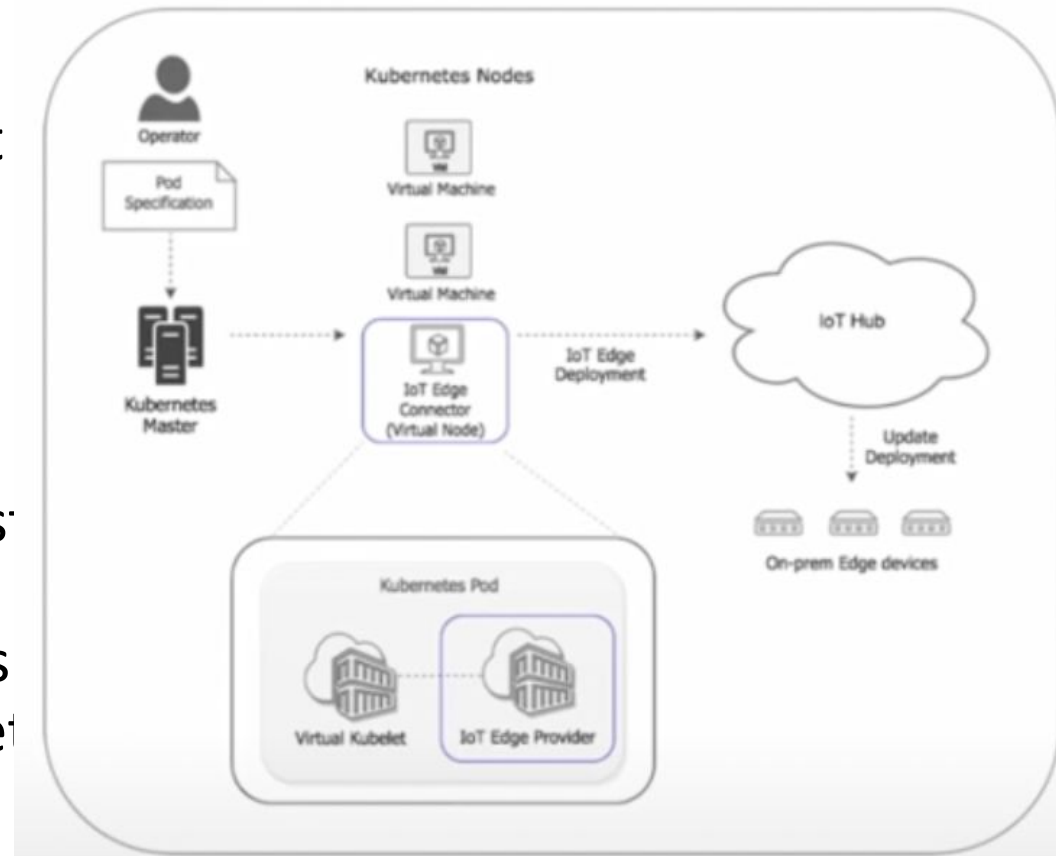


Power of Kubernetes to deploy software on edge devices

Architecture diagram shows works flow from the cloud through the virtual kubelet through the edge provider down to all of your edge devices

First, the virtual kubelet project lets you create a virtual node in your kubernetes cluster, a virtual node is not a VM like most other nodes in the kubernetes cluster instead it is an abstraction of a kubernetes node that is provided by the virtual kubelet

Backing it, is an IOT hub, it can schedule workloads to it and treat it like any other kubernetes node.



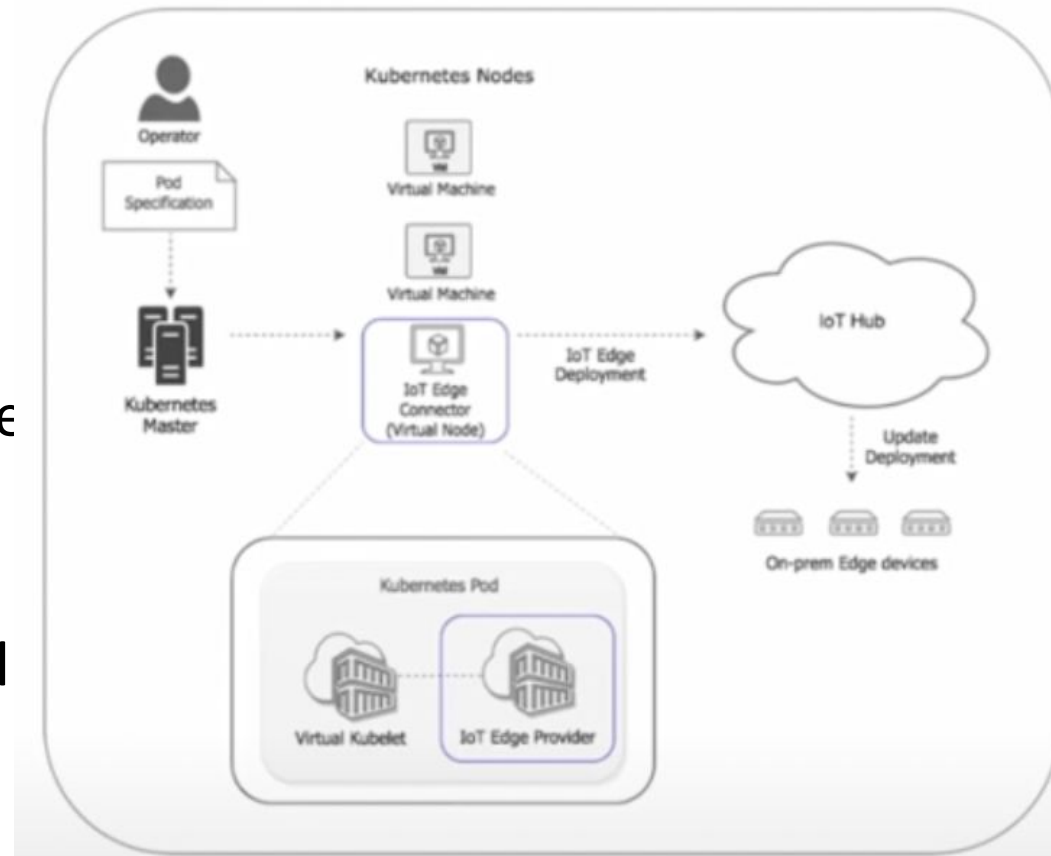
Power of Kubernetes to deploy software on edge devices

When workloads are scheduled to this virtual node, edge provider comes in and that's depicted.


The edge connector or the edge provider which are working in tandem with the virtual kubelet it takes the workload specification that comes in from kubernetes and converts it into an IOT edge deployment.

Then the IOT edge deployment is shipped back to the backing IOT hub for this virtual node.

Lastly, the IOT hub in turn pushes this deployment down to all the targeted devices.



Conclusion

- In this lecture we discussed:
 - Understanding of Kubernetes including
 - Containers
 - Orchestration
 - Concepts of Dockers
 - Power of Kubernetes to deploy software on edge devices
- 
- 