**Creates a Pod:**

**kubectl run my-pod --image=image**

**Label a Pod (to an existing Pod):**

**kubectl label pods my-pod new-label=awesome**

**View Labels of a Pod:**

**kubectl get pods my-pod --show-labels**

**kubectl edit deployment**

**kubectl set image deployment <deployment name> <container name>=<new image name>**

**kubectl delete pod <pod name> <pod name1> <pod name2> ...**

**kubectl explain replicaset**

**kubectl edit rs <rs name>**

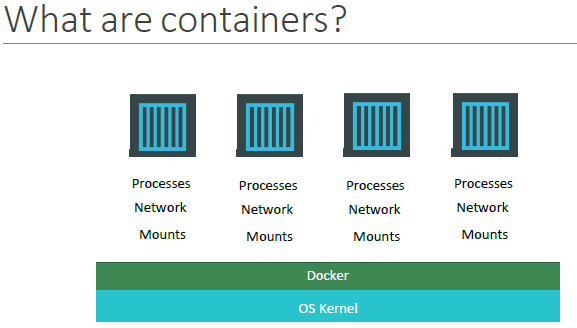
**kubectl scale rs <rs name> --replica=5**

**kubectl expose deployment <deployment name> --type=LoadBlancer --port=80**

**A close up of a text

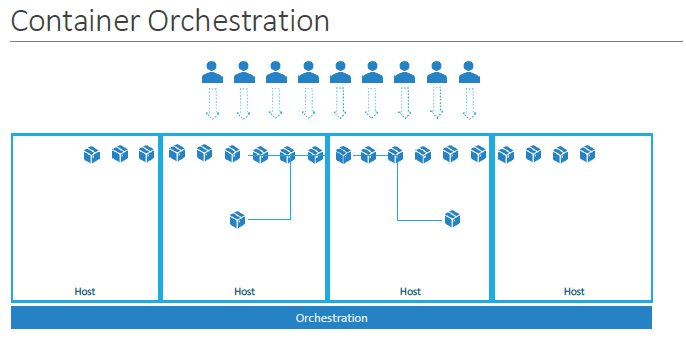
AI-generated content may be incorrect.**

* Kubernetes also known as K8s was built by Google
* Now it’s open-source project and now maintained by Cloud Native Computing Foundation (CNCF).



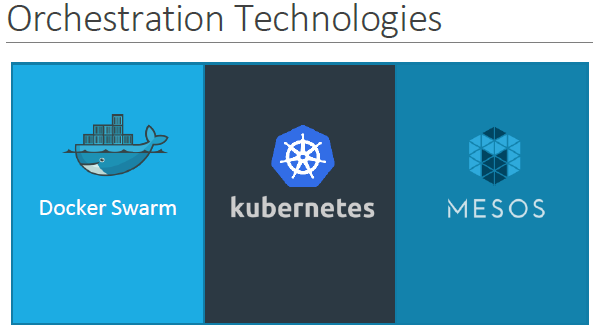
Containers are completely isolated environments, as in they can have their own processes or services, their own network interfaces, their own mounts, just like Virtual machines, except that they all share the same OS kernel.

But it’s also important to note that containers are not new with Docker. Containers have existed for about 10 years now and some of the different types of containers are LXC, LXD, LXCFS etc. Docker utilizes LXC containers.



We now have our application packaged into a docker container. What if our application relies on other containers such as database or messaging services or other backend services? What if the number of users increase and we need to scale our application? We would also like to scale down when the load decreases.

The platform needs to orchestrate the connectivity between the containers and automatically scale up or down based on the load. This whole process of automatically deploying and managing containers is known as Container Orchestration.



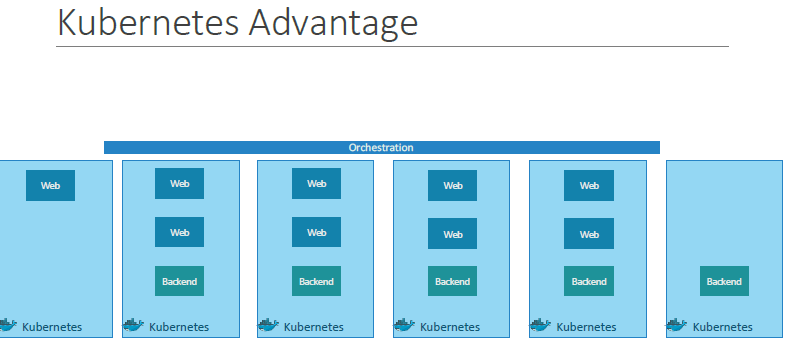
* Docker Swarm from Docker
* Kubernetes from Google
* Mesos from Apache.

While Docker Swarm is easy to setup and get started, it lacks some of the advanced autoscaling features required for complex applications.

Mesos on the other hand is quite difficult to setup and get started but supports many advanced features.

Kubernetes - the most popular of it all – is a bit difficult to setup and get started but provides a lot of options to customize deployments and supports deployment of complex architectures.

Kubernetes is now supported on all public cloud service providers like GCP, Azure and AWS.

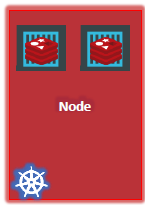


**Features of Kubernetes:**

* Container Orchestration
* Scalability:
  + When demand increases/decreases scale (deploy) the number of nodes up/down without having to take down the application
* Load Balancing:
  + The user traffic is load balanced across the various containers
* High Availability:
  + As we have multiple instances of our application running on different nodes
* Rollouts & Rollback

**Kubernetes - Cluster Architecture:**

* Follows client-server architecture.
* Wherein, we have master installed on one machine and the node on separate machines.

****

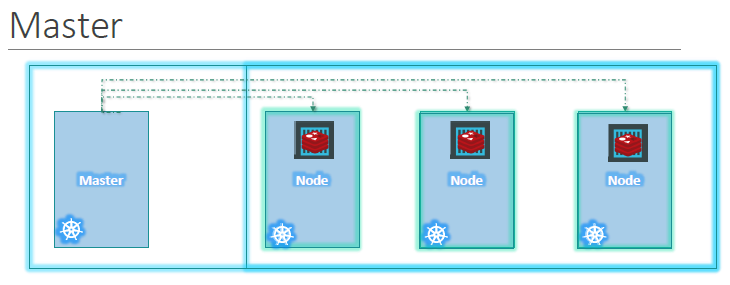
* A node is a machine – physical or virtual – on which Kubernetes is installed.
* Containers will be launched by Kubernetes.
* It’s also knowing as Minions in the past.

But what if the node on which our application is running fails? Well, obviously our application goes down. So, we need to have more than one node.

A screenshot of a computer

AI-generated content may be incorrect.

* A cluster is a set of nodes grouped together.
* This way even if one node fails, we have our application still accessible from the other nodes.
* Moreover, having multiple nodes helps in sharing load as well.



Q.) Now we have a cluster, but who is responsible for managing the cluster?

Q.) Where is the information about the members of the cluster stored?

Q.) How are the nodes monitored?

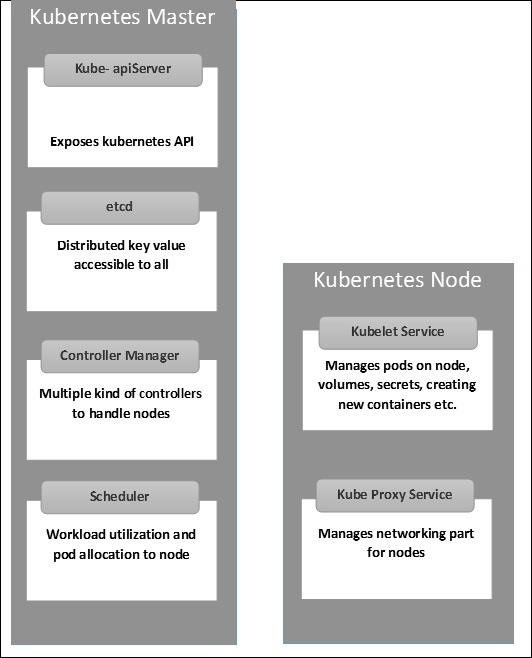
Q.) When a node fails how do we move the workload of the failed node to another worker node?

* That’s where the Master comes in. The master is another node with Kubernetes installed in it and is configured as a Master.
* The master watches over the nodes in the cluster and is responsible for the actual orchestration of containers on the worker nodes.

**Components:**

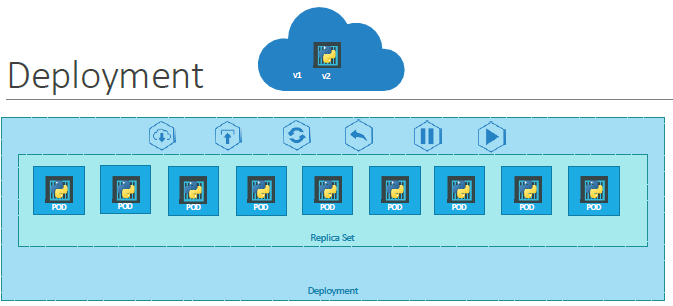
A computer screen shot of a diagram

AI-generated content may be incorrect.



When we install Kubernetes on a System, we are installing the following components:

* **API server**
  + The users, management devices, CLI all talk to the API server to interact with the Kubernetes cluster.
* **ETCD**
  + Distributed key-value to store all data used to manage the cluster.
* **Scheduler**
  + Distributing work on containers across multiple nodes.
* **Controllers**
  + Brain behind orchestration.
  + They are responsible for noticing and responding when nodes, containers go down.
  + Makes decisions to bring up new containers in such cases.
  + The key controllers are:
    - replication controller
    - endpoint controller
    - namespace controller
    - service account controller
* **Container runtime**
  + Underlying software that is used to run containers.
  + In our case it will be Docker. There are other container runtime alternatives available such as Rocket or CRIO.
* **Kubelet**
  + Agent that runs on each node in the cluster (make sure containers are running in pods)
  + Responsible for interacting with the master to provide health information of the worker node and carry out actions requested by the master on the worker nodes.
* **Kube-Proxy:**
  + Maintains network rules for communication with Pods.
  + This is a proxy service which runs on each node and helps in making services available to the external host.
  + It helps in forwarding the request to correct containers and can perform primitive load balancing.
  + It manages pods on node, volumes, secrets, creating new containers health checkup, etc.



Say for example we have a web server that needs to be deployed in a production environment. We need not ONE, but many such instances of the web server running for obvious reasons.

And when newer versions of application builds become available on the docker registry, we would like to UPGRADE our docker instances seamlessly.

However, when we upgrade our instances, we do not want to upgrade all of them at once. This may impact users accessing our applications, so we may want to upgrade them one after the other. And that kind of upgrade is known as **Rolling Updates**.

Suppose one of the upgrades we performed resulted in an unexpected error and we are asked to undo the recent update. We would like to be able to roll back the changes that were recently carried out.

Finally, say for example we would like to make multiple changes to our environment such as upgrading the underlying Webserver versions, as well as scaling our environment and modifying the resource allocations etc. We do not want to apply each change immediately after the command is run, instead we would like to apply a pause to our environment, make the changes and then resume so that all changes are rolled out together.

All these capabilities are available with the Kubernetes Deployments.

So far, we discussed about PODs, which deploy single instances of our application such as the web application. Each container is encapsulated in PODs. Multiple such PODs are deployed using Replication Controllers or Replica Sets.

And then comes **Deployment** which is a Kubernetes object that comes higher in the hierarchy. The deployment provides us with capabilities to upgrade the underlying instances seamlessly using **rolling updates, undo changes, and pause and resume** changes to deployments.

Q.) How do we create a deployment.

**The template has a POD definition inside it.**

The deployment automatically creates a ReplicaSet. The ReplicaSet ultimately create pods.

A screenshot of a computer program

AI-generated content may be incorrect.

* We used the kubectl run command to create a POD.
* This command in fact creates a deployment and not just a POD.
* This is why the output of the command says Deployment nginx created.
* This is another way of creating a deployment by only specifying the image name and not using a definition file.
* A ReplicaSet and pods are automatically created in the backend.
* Using a definition file is recommended though as we can save the file, check it into the code repository and modify it later as required.