**Kubernetes:**

**Kubelet**: A process which runs on each node in the cluster. Kubelet talks to the master server and gets a list of containers to run and then runs, manages, and reports container status back to the master server.

**Pod**: The primary unit of Kubernetes scheduling and management. A Pod is list of containers that are always run together on one node. The containers in a pod share an IP address and a network stack, but are otherwise isolated from each other.

**Container**: A Docker container, it has an isolated process space, can expose ports, can define environment variables and a run command. Containers can mount volumes from the host, a cloud volume, or other storage provider. A container can be constrained by CPU and memory limits to prevent it from overrunning the node it runs on. A container can define health checks which are automatically run by kubelet to determine if the container is alive and ready. If a container fails its LivenessProbe, kubelet will automatically restart it. If a container fails its ReadinessProbe, Kuberenetes will remove it from all Services it is a part of.

**Deployment**: An object which manages a group of identical pods. Each pod in a Deployment runs the same code but may be running on different nodes. A Deployment is defined with a number of replicas to maintain, so that if a pod or node fails then Kubernetes will automatically allocate and schedule a replacement pod. When a Deployment is modified, Kubernetes uses a RollingUpdateStrategy to slowly roll out the new version while also removing old pods.

**Service**: An abstraction allowing a client to talk to any one of a set of equivalent pods(usually the Pods are in a Deployment). When a Service is created, Kubernetes allocates a cluster internal DNS name and IP address for it. This allows a client to use a single DNS name or IP address to connect to any healthy pod without knowing which node the pod is running on. A Service can be of type ClusterIP,NodePort, or LoadBalancer. A **ClusterIP** is an virtual IP address accessible only inside the Kubernetes cluster. It is stable address with a DNS name that clients inside the cluster can use as a hostname to access the service. A **NodePort**Service has a ClusterIP, but it also exposes the same port externally on all nodes in the cluster and forwards traffic to the cluster IP. NodePort is used to make Kubernetes services accessible to clients outside of the cluster which can access it via any node of the cluster on the NodePort. A **LoadBalancer** service creates an actual LoadBalancer in your cloud provider(an AWS ELB, a GCP LB, or an Azure LB) which can proxy requests to a NodePort service.

**Namespace**: A logical separation of applications or environments. Most objects like Deployments, Pods, and Services exist within a specific namespace. Users can create their own namespaces. Clusters include a pre-created “default” namespace and a “kube-system” namespace for cluster management objects like the Kubernetes scheduler and kube-dns pods.

**ReplicationController**: An older, less used abstraction over a group of identical pods. It is similar to a Deployment, but with fewer features, primarily lacking RollingUpdate and rollback capabilities.

**StatefulSet**: Similar to a Deployment, but newer, a StatefulSet gives each replica in it a unique identity so they can be addressed and managed individually. StatefulSets are particularly useful when a set of pods each needs their own stable, persistent storage even as the pods are scaled up or restarted, for instance for storage services like Cassandra, Mongo, or Kafka. Pods in a Deployment are all treated equally while Pods in a StatefulSet have certain features like a specific ordinal number and a fixed scale up and scale down sequence. Eg, StatefulSet will always create pod-1, pod-2, then pod-3, and when scaling down will destroy them in the reverse order.

**Ingress**: An object that defines rules which can handle HTTP traffic coming into the Kubernetes cluster. Common uses for an Ingress include: sharing port 80 or 443 among many different services, doing HTTP host or path based routing to different services, and terminating SSL. An **IngressController** is a pod which manages and deploys the rules to a set of pods which handle the traffic. The Ingress can be implemented by different IngressControllers, the most popular of which is the nginx IngressController.

**ConfigMap**: A key-value store used for configuration information. A container can import any key-value pair from any ConfigMap in the same namespace. A ConfigMap entry can be accessed as an environment variable in the container or a container can get all the key-value pairs of a ConfigMap by mounting it as a volume. Changes to a ConfigMap are automatically reflected in pods which use a volume mount.

**Secret**: Very similar to a ConfigMap, but will support encryption in the future.

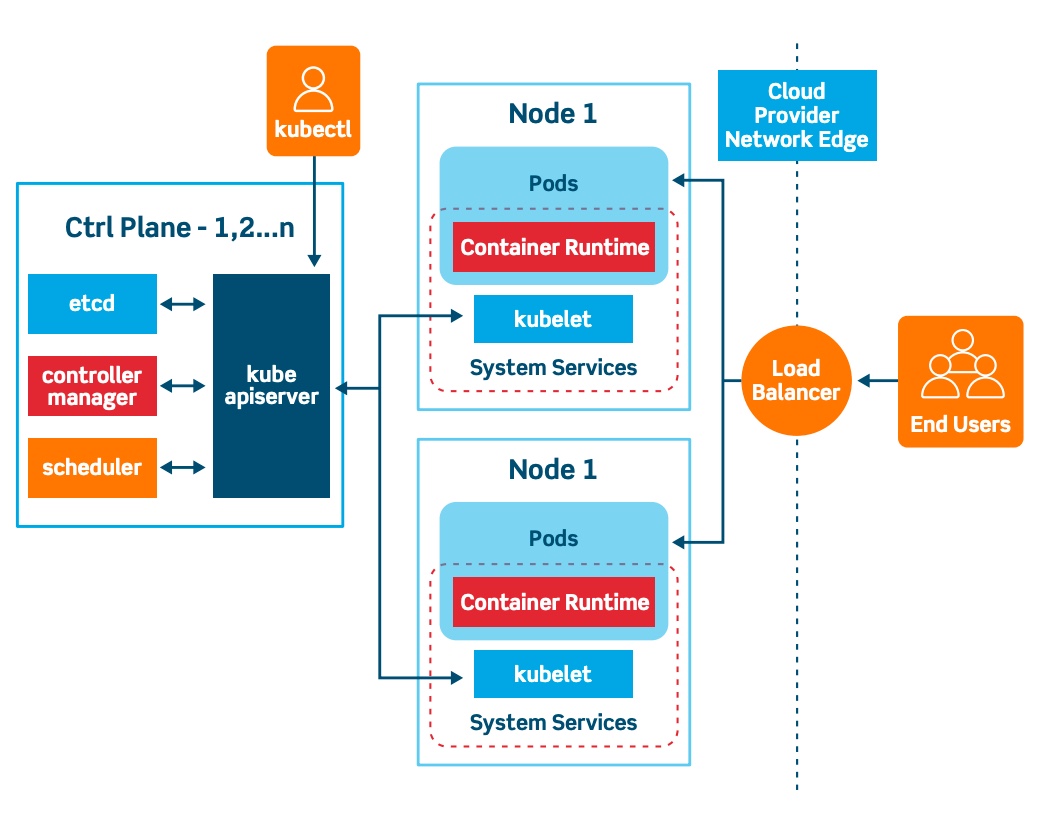
Important concept areas for things like specific cloud providers, network, security, storage, scheduling, API users, system administrators, and Kubernetes internals.

**Kubernetes Master:**It is the primary control unit that manages workloads and communication across the system. Each of its components has a different process which can run on a single master node or on multiple master nodes. Its components are:-

* **Etcd Storage:**It is an open-source key-value data store developed by CoreOS team and can be accessed by all nodes in the cluster. Kubernetes uses ‘Etcd’ to store configuration data of the cluster to represent the overall state of the cluster anytime.
* **API-Server:** The API server is the central management entity that receives REST requests for modifications, serving as a front-end to control cluster. Moreover, this is the only thing that communicates with Etcd cluster, making sure that data is stored in Etcd. kubectl
* **Scheduler:**It helps to schedule the pods on various nodes based on resource utilization and decides where to deploy which service. The scheduler has the information regarding the resources available to the members as well as the one which is left for configuring the service to run.
* **Controller Manager:** It runs a number of distinct controller processes in the background to regulate the shared state of the cluster and perform a routine task. When there is any change in the service, the controller spots the change and starts working towards the new desired state.

**Worker Node:**This is also known as Kubernetes node or Minion node which contains the sufficient information to manage networking between containers such as Docker, communication between the master node as assigning the resources to the containers as per scheduled

* **Kubelet:** Kubelet ensures that all containers in the node are running and are in the healthy state. Kubelet monitors the state of a pod, if it is not in the desired state. If in-case node fails, replication controller observes this change and launches pods on another healthy pod.
* **Container:** Containers are the lowest level of Microservice, placed inside the pod and needs external IP address to view the outside process.
* **Kube Proxy:** It acts as a network proxy and a load balancer. Additionally, it forwards the request to the correct pods across isolated networks in a cluster.
* **cAdvisor:** Acts as an assistant who is responsible for monitoring and gathering data about resource usage and performance metrics on each node.



<https://towardsdatascience.com/learn-enough-docker-to-be-useful-b7ba70caeb4b>

Swarm:

Docker swarm is a Clustering and orchestration tool. It is used for scheduling containers across multiple nodes.

1) you have 10 machines (virtual machine or physical machines). You want to use all of them as underlying hardware and kernel resources for starting multiple containers.

2) Now you have a requirement to start an application stack which has web tier, app tier and db tier.

3) all these three tiers would need to communicate with each other.

Swarm basically allows you to achieve

#1. You can combine multiple nodes as a cluster and then send “docker run” command to this cluster (actually to the Swarm manager node) Swarm will handle scheduling(starting) this container on one of the nodes. There are different orchestration tools which are available but Swarm is built in and works across all platforms where Docker engine can run. Also, if you develop using Docker native commands those commands will mostly work as it is against a swarm cluster.

Now you can just use Docker swarm mode [Swarm mode overview](https://docs.docker.com/engine/swarm/) and initiate a cluster with “docker swarm init”

Now when it comes to running an application stack which has multiple components, you would need Docker Compose. So in our scenario you would use Docker Compose to achieve –

#2. You will define a spec file where you would define how a container has to built for each of your components - db, app and web tier. You can also specify how will those interact with each other. How many instance of each and many other things. You should checkout the overview and Docker Compose file specs on this official doc page of [Docker Compose](https://docs.docker.com/compose/). Think of it as multi-conyainer version of Dockerfile (though it is not just that, it is more). You can use Docker Compose against a Docker swarm cluster.

**Docker Compose is a client side tool where Docker swarm is a server side feature in [docker's client server architecture](https://docs.docker.com/engine/docker-overview/" \l "docker-architecture" \t "_blank).**

[Docker network](https://docs.docker.com/engine/userguide/networking/) helps you achieve

#3 where you would want your containers to be able to talk to each other (and also to external world). There are default network drivers included and when you fire “docker run”, it uses defaults which are good in most cases and works while you are learning docker. But as you go more advanced or if you would want to deploy containers in production, you would like to isolate containers and application stack from each other or expose the ports of your container on host or on specific physical interface. That is where docker networks will play a major role.

Docker network is also a server side feature. You can setup a docker network on a swarm cluster as well. There are certain networks (e.g. [overlay](https://docs.docker.com/engine/userguide/networking/#overlay-networks-in-swarm-mode), [docker\_gwbridge](https://docs.docker.com/engine/userguide/networking/" \l "the-dockergwbridge-network" \t "_blank)) which can only be setup when running docker in swarm mode. Some networks (e.g. overlay or [ingress](https://docs.docker.com/engine/swarm/ingress/)) allow you to setup networking across all nodes in a cluster. Docker network is also relevant when you are setting up a container which has to be accessed from outside world. Then you need to [publish the container’s port](https://docs.docker.com/engine/userguide/networking/default_network/binding/) to host port.

First, I’m going to shed some light on Docker metaphors.

Metaphors help us make sense of new things. For example, the metaphor of a physical container helps us quickly grasp the essence of a virtual container.

Containers are hugely helpful for improving security, reproducibility, and scalability in software development and data science.

A Docker container is a Docker image brought to life

**Container**:

Like a physical plastic container, a Docker container:

**Holds things** — Something is either inside the container or outside the container.

**Is portable** — It can be used on your local machine, your coworker’s machine, or a cloud provider’s servers (e.g. AWS). Sort of like that box of childhood knickknacks you keep moving with you from home to home.

**Has clear interfaces for access** — Our physical container has a lid for opening and putting things in and taking things out. Similarly, a Docker container has several mechanisms for interfacing with the outside world. It has ports that can be opened for interacting through the browser. You can configure it to interact with data through the command line.

**Can be obtained from a remote location** — You can get another empty plastic container from Amazon.com when you need it. Amazon gets its plastic containers from manufacturers who stamp them out by the thousands from a single mold. In the case of a Docker container, an offsite registry keeps an image, which is like a mold, for your container. Then when you need a container you can make one from the image.

**Living Instance**

A second way you can think of a Docker container is as an instance of a living thing. An instance is something that exists in some form. It’s not just code. It’s code that has brought something to life. Like other living things, the instance will eventually die — meaning the container will shut down.

With Docker you can run multiple containers simultaneously on a host machine. And like other software programs, Docker containers can be run, inspected, stopped, and deleted.

Docker images are more like blueprints, cookie cutters, or molds. Images are the immutable master template that is used to pump out containers that are all exactly alike.

Small images are faster to pull over the network and faster to load into memory when starting containers or services. There are a few rules of thumb to keep image size small:

<https://blog.docker.com/2018/08/containers-replacing-virtual-machines/>

### Point #1: Containers Are More Agile than VMs

At this stage of container maturity, there is very little doubt that containers give both developers and operators more agility. Containers deploy quickly, deliver immutable infrastructure and solve the age-old “works on my machine” problem. They also replace the traditional patching process, allowing organizations to respond to issues faster and making applications easier to maintain.

### Point #2: Containers Enable Hybrid and Multi-Cloud Adoption

Once containerized, applications can be deployed on any infrastructure – on virtual machines, on bare metal, and on various public clouds running different hypervisors. Many organizations start with running containers on their virtualized infrastructure and find it easier to then migrate to the cloud without having to change code.

### Point #3: Integrate Containers with Your Existing IT Processes

Most enterprise organizations have a mature virtualization environment which includes tooling around backups, monitoring, and automation, and people and processes that have been built around it. By running Docker Enterprise on virtualized infrastructure, organizations can easily integrate containers into their existing practices and get the benefits of points 1 and 2 above.

### Point #4: Containers Save on VM Licensing

Containerized applications share common operating system and software libraries which greatly improves CPU utilization within a VM. This means an organization can reduce the overall number of virtual machines needed to operate their environment and increase the number of applications that can run on a server. Docker Enterprise customers often see 50% increased server consolidation after containerizing which means less hardware costs and savings on VM and OS licensing.

### What About Bare Metal?

Just as organizations have reasons for using different servers or different operating systems, there are reasons that some organizations will want to run containers directly on bare metal. This is often due to performance or latency concerns or for licensing and cost reasons.

### What About Security?

Docker is name of a project and a company. the company docker started the project called docker (oversimplification) which using containerization ( a loose name to a collection of technologies) creates containers that can isolate certain resources on the machine and let you run your software in it.

Virtual Machine is a virtual (not real) representation of a machine in software. It results from usage of “virtualization” technology that enables taking a “slice” of your physical hardware(cpu cores, memory, disk) and represent i..

At the most basic level, VMs contain complete operating system AND whatever applications run on it. VM=OS virtualization. More overhead required for the OS, in addition to the application.

Docker is an example of application-only virtualization. Less overhead as compared to a true VM.

Docker originally used [LinuX Containers](http://lxc.sourceforge.net/) (LXC), but later switched to [runC](https://github.com/opencontainers/runc) (formerly known as **libcontainer**), which runs in the same operating system as its host. This allows it to share a lot of the host operating system resources. Also, it uses a layered filesystem ([AuFS](http://aufs.sourceforge.net/)) and manages networking.

AuFS is a layered file system, so you can have a read only part and a write part which are merged together. One could have the common parts of the operating system as read only (and shared amongst all of your containers) and then give each container its own mount for writing.

So, let's say you have a 1 GB container image; if you wanted to use a full VM, you would need to have 1 GB times x number of VMs you want. With Docker and AuFS you can share the bulk of the 1 GB between all the containers and if you have 1000 containers you still might only have a little over 1 GB of space for the containers OS (assuming they are all running the same OS image).

A full virtualized system gets its own set of resources allocated to it, and does minimal sharing. You get more isolation, but it is much heavier (requires more resources). With Docker you get less isolation, but the containers are lightweight (require fewer resources). So you could easily run thousands of containers on a host, and it won't even blink. Try doing that with Xen, and unless you have a really big host, I don't think it is possible.

A full virtualized system usually takes minutes to start, whereas Docker/LXC/runC containers take seconds, and often even less than a second.

There are pros and cons for each type of virtualized system. If you want full isolation with guaranteed resources, a full VM is the way to go. If you just want to isolate processes from each other and want to run a ton of them on a reasonably sized host, then Docker/LXC/runC seems to be the way to go.

For more information, check out [this set of blog posts](http://web.archive.org/web/20150326185901/http:/blog.dotcloud.com/under-the-hood-linux-kernels-on-dotcloud-part) which do a good job of explaining how LXC works.