Extensive introduction to Container Technology

By: Eng. Mohamed ElEmam

Email: Mohamed.ElEmam.Hussin@gmail.com

Container Technology

History:

1- **1979** - Unix v7 chroot
2- **2000** - Free BSD Jails
3- **2004** - Solaris zones
4- **2007** - Cgroups Linux kernel 2.6.24
5- **2008**- LXC (linuxcontainers)
6- **2013** - Docker (Real start toward container technology)
7- **2015** - Kubernetes

What is a Container?

A container is a products of operating system virtualization. They provide a lightweight virtual environment that groups and isolate a set of processes and resources memory, CPU, disk, etc., from the host and any other containers. The isolation guarantee that any processes inside container cannot see any other resources or processes outside the container.

- Centralization uses the kernel on the host operating system to run multiple root file systems.
- Each root file system calls a container.
- Each container has its own (process, memory, devices, network stack)

Control groups (cgroups)

Cgroups are kernel mechanisms to restrict and measure physical resources allocations to each process group. Using cgroups, you can allocate physical resources such as CPU time, network, and memory.

Name Space

Linux processes form a single hierarchy, with all processes rooting at init.

Usually privileged processes in this tree can trace or kill other processes.

Linux namespace enables us to have many hierarchies of processes with their own “subtrees” such that processes in one subtree can’t access or even know of those in another.

Linux namespaces comprise some of the fundamental technologies behind most modern-day container implementations. At a high level, they allow for isolation of global system resources between independent processes. For example, the PID namespace isolates the process ID number space. This means that two processes running on the same host can have the same PID!

Cgroups vs namespace

- Name Space: isolate file system of applications
- Cgroups: isolate physical resources.

- **cgroup**: Control Groups provide a mechanism for aggregating/partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behaviour.
- **Namespace**: wraps a global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

In short:

- **Cgroups** = limits how much you can use;
- **namespaces** = limits what you can see (and therefore use)

Cgroups involve resource metering and limiting:

- memory
- CPU
- block I/O
- network

Namespaces provide processes with their own view of the system

Multiple namespaces:

- pid
- net
- mnt

Namespaces

Docker uses a technology called namespaces to provide the isolated workspace called the *container*. When you run a container, Docker creates a set of *namespaces* for that container.

These namespaces provide a layer of isolation. Each aspect of a container runs in a separate namespace and its access is limited to that namespace.

Docker Engine uses namespaces such as the following on Linux:

- **The pid namespace:** Process isolation (PID: Process ID).
- **The net namespace:** Managing network interfaces (NET: Networking).
- **The ipc namespace:** Managing access to IPC resources (IPC: InterProcess Communication).
- **The mnt namespace:** Managing filesystem mount points (MNT: Mount).
- **The uts namespace:** Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

Control groups

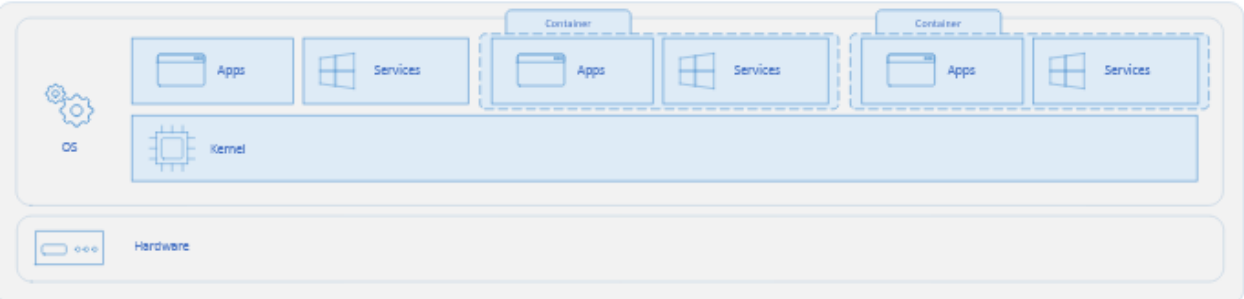
Docker Engine on Linux also relies on another technology called *control groups* (cgroups). A cgroup limits an application to a specific set of resources. Control groups allow Docker Engine to share available hardware resources to containers and optionally enforce limits and constraints. For example, you can limit the memory available to a specific container.

Container vs. Virtual Machine

When people think about virtualization, virtual machines (VMs) often come to mind. In fact, virtualization can take many forms, and containers are one of those. So what's the difference between VMs and containers?

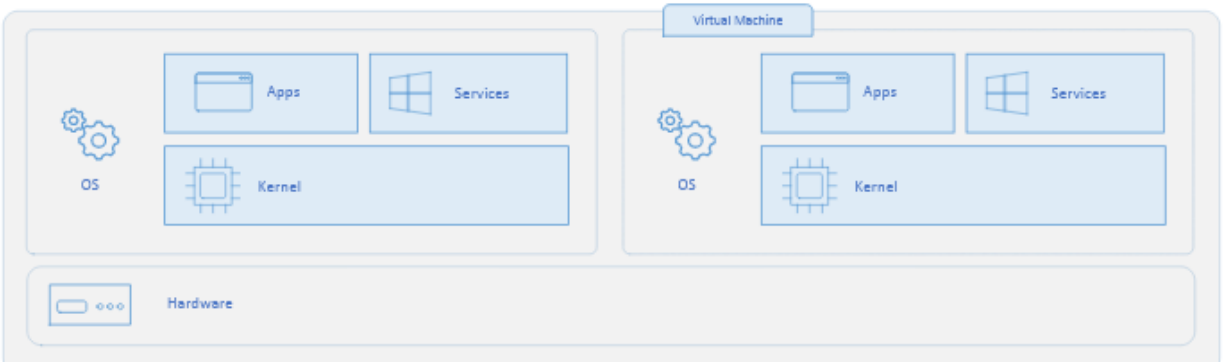
Container architecture

A container is an isolated, lightweight silo for running an application on the host operating system. Containers build on top of the host operating system's kernel (which can be thought of as the buried plumbing of the operating system), and contain only apps and some lightweight operating system APIs and services that run in user mode, as shown in this diagram.



Virtual machine architecture

In contrast to containers, VMs run a complete operating system–including its own kernel–as shown in this diagram.



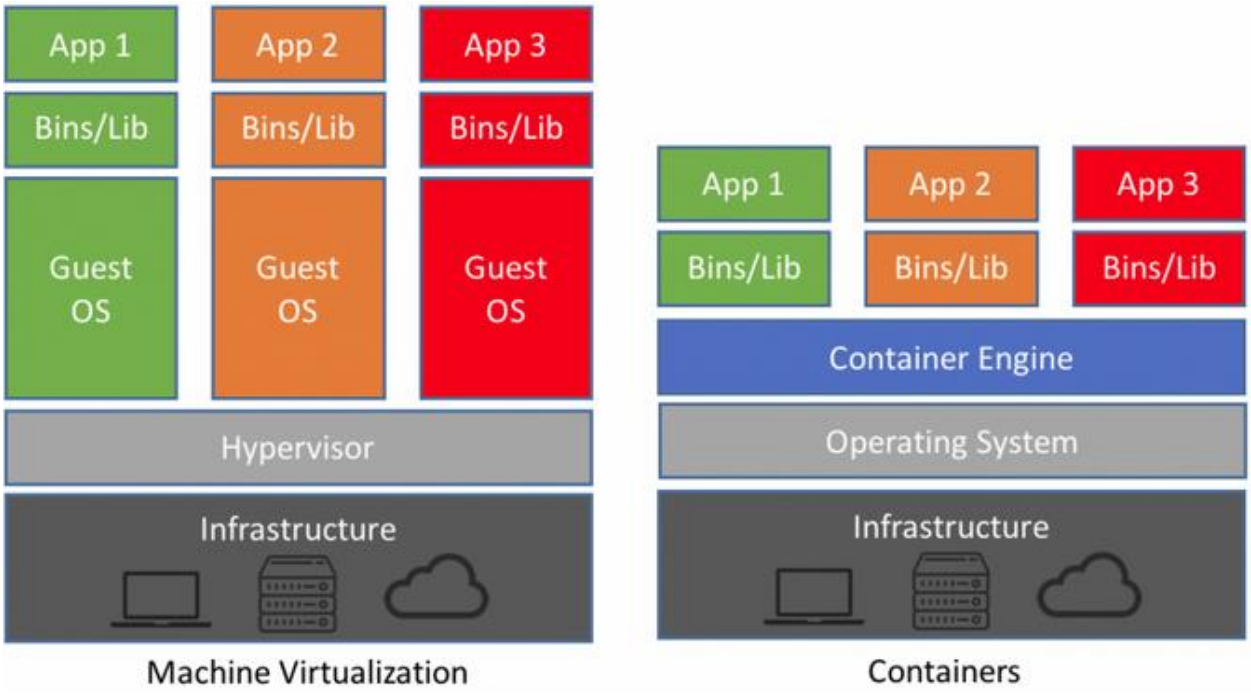
	Virtual machine	Container
Isolation	Provides complete isolation from the host operating system and other VMs. This is useful when a strong security boundary is critical, such as hosting apps from competing companies on the same server or cluster.	Typically provides lightweight isolation from the host and other containers, but doesn't provide as strong a security boundary as a VM.
Operating system	Runs a complete operating system including the kernel, thus requiring	Runs the user mode portion of an operating system, and can be tailored to contain just the needed

	more system resources (CPU, memory, and storage).	services for your app, using fewer system resources.
Guest compatibility	Runs just about any operating system inside the virtual machine	Runs on the same operating system version as the host
Deployment	Deploy individual VMs by using Windows Admin Center or Hyper-V Manager; deploy multiple VMs by using PowerShell or System Center Virtual Machine Manager.	Deploy individual containers by using Docker via command line; deploy multiple containers by using an orchestrator such as Kubernetes Service.
Operating system updates and upgrades	Download and install operating system updates on each VM. Installing a new operating system version requires upgrading or often just creating an entirely new VM. This can be time-consuming, especially if you have a lot of VMs...	Updating or upgrading the operating system files within a container is the same: 1. Edit your container image's build file (known as a Dockerfile) to point to the latest version of the Windows base image. 2. Rebuild your container image with this new base image. 3. Push the container image to your container registry. 4. Redeploy using an orchestrator. The orchestrator provides powerful automation for doing this at scale.
Persistent storage	Use a virtual hard disk (VHD) for local storage for a single VM, or an SMB file share for storage shared by multiple servers	Use Azure Disks for local storage for a single node, or Azure Files (SMB shares) for storage shared by multiple nodes or servers.
Load balancing	Virtual machine load balancing moves running VMs to other servers in a failover cluster.	Containers themselves don't move; instead an orchestrator can automatically start or stop containers on cluster nodes to

		manage changes in load and availability.
Fault tolerance	VMs can fail over to another server in a cluster, with the VM's operating system restarting on the new server.	If a cluster node fails, any containers running on it are rapidly recreated by the orchestrator on another cluster node.
Networking	Uses virtual network adapters.	Uses an isolated view of a virtual network adapter, providing a little less virtualization–the host's firewall is shared with containers–while using less resources.

Virtual Machine: Hardware Visualization

Container: Operating System Virtualization



Container Technology Benefits

1. Platform independence: Build it once, run it anywhere

a major benefit of containers is their portability. A container wraps up an application with everything it needs to run, like configuration files and dependencies. This enables you to easily and reliably run applications on different environments such as your local desktop, physical servers, virtual servers, testing, staging, production environments and public or private clouds. This portability grants organizations a great amount of flexibility, speeds up the development process and makes it easier to switch to another cloud environment or provider, if need be.

2. Resource efficiency and density

since containers do not require a separate operating system, they use up less resources. While a VM often measures several gigabytes in size, a container usually measures only a few dozen megabytes, making it possible to run many more containers than VMs on a single server. Since containers have a higher utilization level with regard to the underlying hardware, you require less hardware, resulting in a reduction of bare metal costs as well as datacenter costs.

3. Effective isolation and resource sharing

although containers run on the same server and use the same resources, they do not interact with each other. If one application crashes, other containers with the same application will keep running flawlessly and won't experience any technical problems. This isolation also decreases security risks: If one application should be hacked or breached by malware, any resulting negative effects won't spread to the other running containers.

4. Speed: Start, create, replicate or destroy containers in seconds

As mentioned before, containers are lightweight and start in less than a second since they do not require an operating system boot. Creating, replicating or destroying containers is also just a matter of seconds, thus greatly speeding up the development process, the time to market and the operational speed. Releasing new software or versions has never been so easy and quick. But the increased speed also offers great opportunities for improving customer experience, since it enables organizations and developers to act quickly, for example when it comes to fixing bugs or adding new features.

5. Immense and smooth scaling

A major benefit of containers is that they offer the possibility of horizontal scaling, meaning you can add more identical containers within a cluster to scale out. With smart scaling, where you only run the containers needed in real time, you can reduce your resource costs drastically and

accelerate your return on investment. Container technology and horizontal scaling has been used by major vendors like Google, Twitter and Netflix for years now.

6. Operational simplicity

Contrary to traditional virtualization, where each VM has its own OS, containers execute application processes in isolation from the underlying host OS. This means that your host OS doesn't need specific software to run applications, which makes it simpler to manage your host system and quickly apply updates and security patches.

7. Improved developer productivity and development pipeline

a container-based infrastructure offers many advantages, promoting an effective development pipeline. As mentioned before, containers ensure that applications run and work as designed locally. This elimination of environmental inconsistencies makes testing and debugging less complicated and less time-consuming since there are fewer differences between running your application on your workstation, test server or in any production environment. The same goes for updating your applications: you simply modify the configuration file, create new containers and destroy the old ones, a process which can be executed in seconds. In addition to these well-known benefits, container tools like Docker offer many other advantages. One of these is version control, making it possible for you to roll-out or roll-back with zero downtime. The possibility to use a remote repository is also a major benefit when working in a project-team, since it enables you to share your container with others.

Container orchestration

Containers are one of the biggest buzzwords in the IT world. Start-ups, SME as well as enterprises are adopting container technology at an explosive rate. Docker is by far the most popular containerization platform, but to successfully adopt containers you will also need to implement a container orchestration system. Kubernetes, based on over 10 years of experience in running containerized workloads at Google, is the clear market leader in container orchestration. You can read all about this amazing project in our blog post about the benefits and business value of Kubernetes.

Container Technology Limitations

- **Not all applications benefit from containers**: Not recommended to run statefull applications inside containers. But in case we want, then must host the data for this app outside the container by integration with external storage.
- **Container Network:** We can build isolated network for each service to application to be fully separated and with different IP range. And integrate with external network.
- **Container Security:** it's a very big weakness in container technology because we can't control the east west traffic between containers or because other service container work on the same host. We can solve it by using 3rd party security tools like **NeuVector** container firewall.
- **Container storage**: By default, the container host all data in the local storage in the Node and can't access by SAN or NAS, all of the data inside a container disappears forever when the container shuts down, unless you save it somewhere else first.
- **Container Monitoring and Logs**: Need some special tools to monitor and collect logs from container and application running inside.
- **Container Backup and DR**: By default No backup or DR solution but there are some workarounds to solve this issue.
- **Graphical applications don't work well**. Docker was designed as a solution for deploying server applications that don't require a graphical interface. While there are some creative strategies (such as X11 video forwarding) that you can use to run a GUI app inside a container, these solutions are clunky at best.
- **Containers don't run at bare-metal speeds**. Containers consume resources more efficiently than virtual machines. But containers are still subject to performance overhead due to overlay networking, interfacing between containers and the host system and so on. If you want 100 percent bare-metal performance, you need to use bare metal, not containers.

Docker

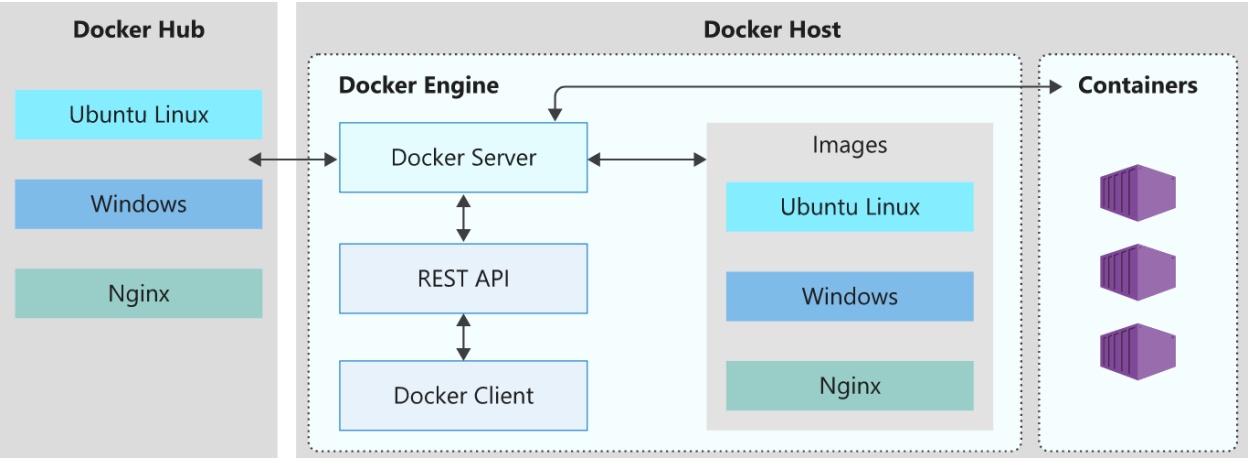
What is Docker?

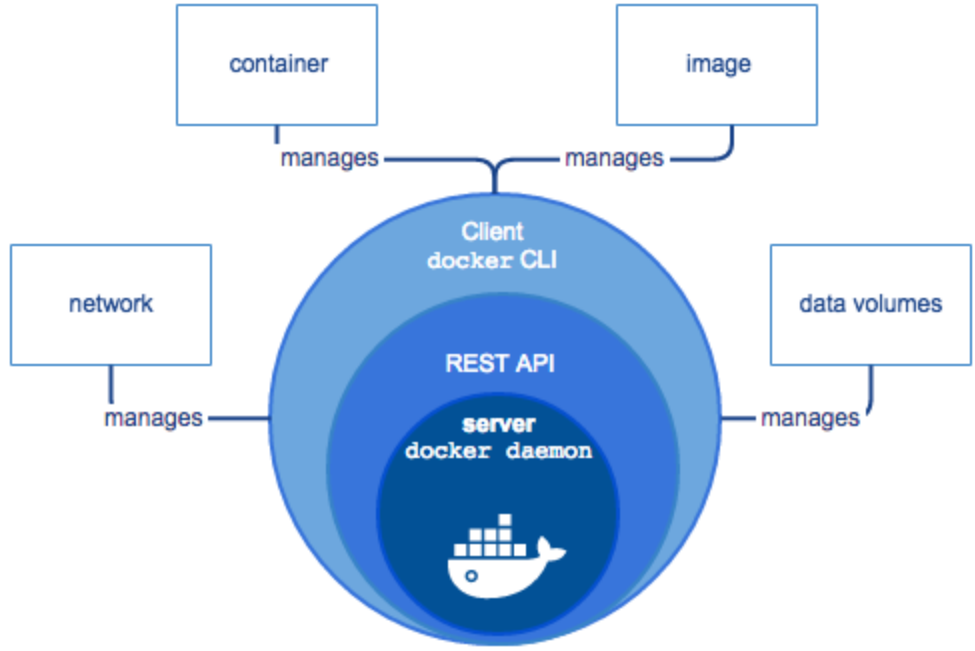
Docker is a containerization platform/engine used to develop, ship, and run containers. Docker doesn't use a hypervisor, and you can run Docker on your desktop or laptop if you're developing and testing applications. The desktop version of Docker supports Linux, Windows,

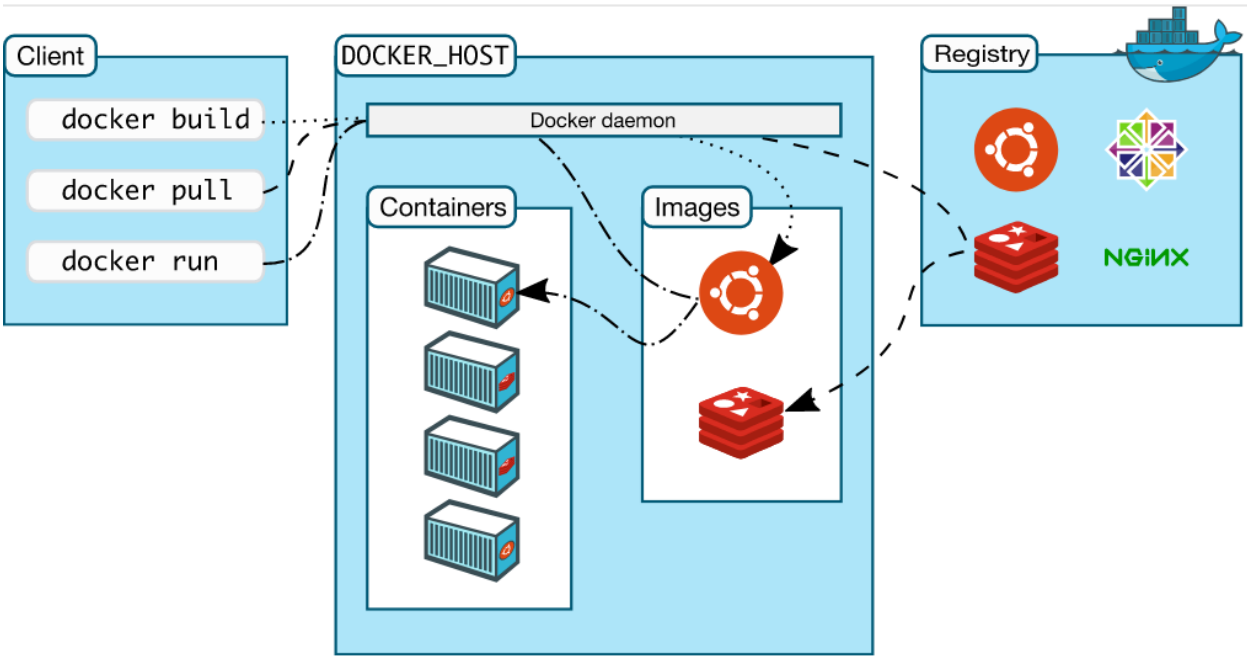
and macOS. For production systems, Docker is available for server environments, including many variants of Linux and Microsoft Windows Server 2016 and above. Many clouds, including Azure, supports Docker.

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.







Docker Engine/Daemon

The Docker Engine consists of several components configured as a client-server implementation where the client and server run simultaneously on the same host. The client communicates with the server using a REST API, which allows the client to also communicate with a remote server instance.

The Docker client

The Docker client is a command-line application named Docker that provides us with a command line interface (CLI) to interact with a Docker server. The Docker command uses the Docker REST API to send instructions to a local or remote server and functions as the primary interface we use to manage our containers.

The Docker server

The Docker server is a daemon named dockerd. The dockerd daemon responds to requests from the client via the Docker REST API and can interact with other daemons. The Docker server is also responsible for tracking the lifecycle of our containers.

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.

Container Image

A container image is a tar file containing tar files. Each of the tar file is a layer. Once all tar files have been extract into the same location then you have the container's filesystem.

Docker objects

There are several objects that you'll create and configure to support your container deployments. These include networks, storage volumes, plugins, and other service objects. We won't cover all of these objects here, but it's good to keep in mind that these objects are items that we can create and deploy as needed.

Kubernetes

What is Kubernetes?

Kubernetes is an open source orchestration tool developed by Google for managing micro-services or containerized applications across a distributed cluster of nodes. Kubernetes provides highly resilient infrastructure with zero downtime deployment capabilities, automatic rollback, scaling, and self-healing of containers (which consists of auto-placement, auto-restart, auto-replication, and scaling of containers on the basis of CPU usage).

Kubernetes Components and Architecture

Kubernetes Master Components

Below are the main components found on the master node:

- **API server** - Kubernetes API server is provides APIs to support lifecycle orchestration (scaling, updates, and so on) for different types of applications. It also acts as the gateway to the cluster, so the API server must be accessible by clients from outside the cluster integration with CLI and GUI.
- **Controller-manager** - The Controller Manager is the engine or daemon that runs the core control loops, watches the state of the cluster, and makes changes to drive status toward the desired state.
- **Cloud-controller-manager** - is responsible for managing controller processes with dependencies on the underlying cloud provider (if applicable). For example, when a controller needs to check if a node was terminated or set up routes, load balancers or volumes in the cloud infrastructure all that is handled by the cloud-controller-manager.
- **Scheduler** - is responsible for the creation and scheduling of containers across the nodes in the cluster; it takes various constraints into account, such as resource limitations or guarantees, and affinity and anti-affinity specifications.
- **etcd cluster** - etcd is a critical part of the Kubernetes stack. etcd stores the state of the Kubernetes cluster, including node and workload information.

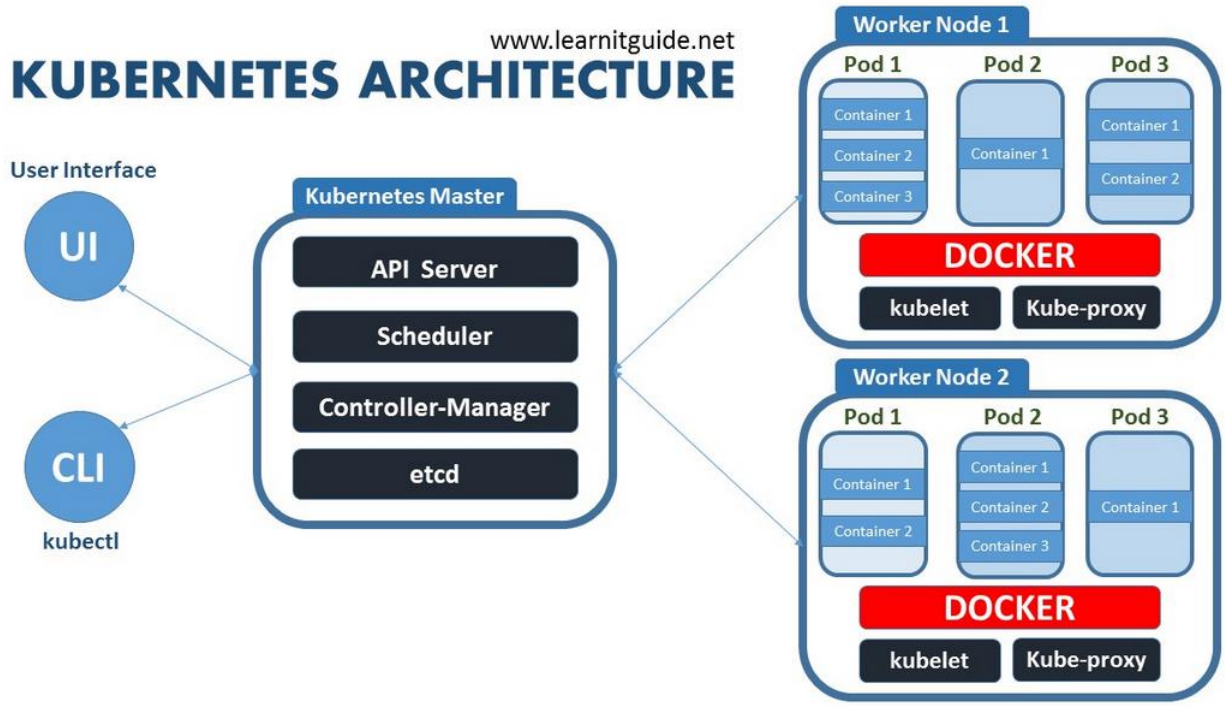
Node (worker) components

Below are the main components found on a (worker) node:

- **kubelet** - the main service on a node, connect between Master and Node and ensuring that pods and their containers are healthy and running in the desired state. This component also reports to the master on the health of the host where it is running.
- **kube-proxy** - a proxy service that runs on each worker node to deal with individual host subnetting and expose services to the external world. It performs request forwarding to the correct pods/containers across the various isolated networks in a cluster.

Kubectl

kubectl command is a line tool that interacts with kube-apiserver and send commands to the master node. Each command is converted into an API call.



Kubernetes Concepts

Making use of Kubernetes requires understanding the different abstractions it uses to represent the state of the system, such as services, pods, volumes, namespaces, and deployments.

- **Pod** - generally refers to one or more containers that should be controlled as a single application. A pod encapsulates application containers, storage resources, a unique network ID and other configuration on how to run the containers.
- **Service** - pods are volatile, that is Kubernetes does not guarantee a given physical pod will be kept alive (for instance, the replication controller might kill and start a new set of pods). Instead, a service represents a logical set of pods and acts as a gateway, allowing (client) pods to send requests to the service without needing to keep track of which physical pods actually make up the service.
- **Volume** - similar to a container volume in Docker, but a Kubernetes volume applies to a whole pod and is mounted on all containers in the pod. Kubernetes guarantees data is preserved across container restarts. The volume will be removed only when the pod gets destroyed. Also, a pod can have multiple volumes (possibly of different types) associated.
- **Namespace** - a virtual cluster (a single physical cluster can run multiple virtual ones) intended for environments with many users spread across multiple teams or projects, for isolation of concerns. Resources inside a namespace must be unique and cannot access resources in a different namespace. Also, a namespace can be allocated a resource quota to avoid consuming more than its share of the physical cluster's overall resources.
- **Deployment** - describes the desired state of a pod or a replica set, in a yaml file. The deployment controller then gradually updates the environment (for example, creating or deleting replicas) until the current state matches the desired state specified in the deployment file. For example, if the yaml file defines 2 replicas for a pod but only one is currently running, an extra one will get created. Note that replicas managed via a deployment should not be manipulated directly, only via new deployments.

Container Components

1- **Nano OS** (Atomic, CoreOS, Photon, RedHat, Windows …etc.,)
2- **Container Engine** (Docker, rocket, cri-o …etc.,)

3- **Orchestrator** (Kubernetes or Swarm)
4- **Image Repository** (Nexus, Docker Hub, Harbor …etc.,)
5- **Platform** (Network, Security, Storage, Monitoring)
6- **Version Control**
7- **CI/CD**
8- **CAAS**
9- **Serverless**

Docker CLI

Management Commands:

container **Manage containers**

image **Manage images**

network **Manage networks**

node **Manage Swarm nodes**

plugin **Manage plugins**

secret **Manage Docker secrets**

service **Manage services**

stack **Manage Docker stacks**

swarm **Manage Swarm**

system **Manage Docker**

volume **Manage volumes**

Commands:

attach	**Attach to a running container**
build	**Build an image from a Dockerfile**
commit	**Create a new image from a container's changes**
cp	**Copy files/folders between a container and the local filesystem**
create	**Create a new container**

diff	**Inspect changes on a container's filesystem**
events	**Get real time events from the server**
exec	**Run a command in a running container**
export	**Export a container's filesystem as a tar archive**
history	**Show the history of an image**
images	**List images**
import	**Import the contents from a tarball to create a filesystem image**
info	**Display system-wide information**
inspect	**Return low-level information on Docker objects**
kill	**Kill one or more running containers**
load	**Load an image from a tar archive or STDIN**
login	**Log in to a Docker registry**
logout	**Log out from a Docker registry**
logs	**Fetch the logs of a container**
pause	**Pause all processes within one or more containers**
port	**List port mappings or a specific mapping for the container**
ps	**List containers**
pull	**Pull an image or a repository from a registry**
push	**Push an image or a repository to a registry**
rename	**Rename a container**
restart	**Restart one or more containers**
rm	**Remove one or more containers**
rmi	**Remove one or more images**
run	**Run a command in a new container**
save	**Save one or more images to a tar archive (streamed to STDOUT by default)**
search	**Search the Docker Hub for images**
start	**Start one or more stopped containers**
stats	**Display a live stream of container(s) resource usage statistics**
stop	**Stop one or more running containers**
tag	**Create a tag TARGET_IMAGE that refers to SOURCE_IMAGE**
top	**Display the running processes of a container**
unpause	**Unpause all processes within one or more containers**
update	**Update configuration of one or more containers**
version	**Show the Docker version information**
wait	**Block until one or more containers stop, then print their exit codes**

An image, is a blueprint from which an arbitrary number of brand-new containers can be started. Images can't change (well, you could point the same tag to different images, but let's not go there), but you can start a container from an image, perform operations in it and save another image based on the latest state of the container. No "currently running commands" are

saved in an image. When you start a container it's a bit like booting up a machine after it was powered down.

It's like a powered down computer (with software installed), which is ready to be executed with a single command. Only instead of starting the computer, you create a new one from scratch (container) which looks exactly like the one you chose (image).

- To download or install a new image
    1- Docker pull (Image Name) will go to repository and download it
    2- Docker run (Image Name) first will search in local repository if exist will run the container if not will go to public repository and download it then run.
- To run container in background (docker run –d (Image Name))
- **docker run -d -p 30000:80 nginx**
    1- **-p** to publish container to be accessible from outside container inside local netwok)
    2- **30000** port I spasified
    3- **80** nginx port
- To go and manage inside CLI of container
    1- **BASH:** docker exec -it <ContainerName or ID> /bin/bash
    2- **PowerShell:** docker exec <ContainerName or ID> powershell –c
- To check all running containers (docker ps)
- To check all history of running containers (docker ps -a)
- To remove container (docker rm <ContainerName or ID>)
- To remove image (docker rmi <Image ID>)
- To list all images (docker images)
- By default when we turn off the container the work will not be saved it will revert back to its original state
- I can create more than container from Image
- **To solve it must to use shared storage and mount a volume:** use volumes and mount the file during container start docker run -v my.ini:/etc/mysql/my.ini percona (and similar with docker-compose). Be aware, you can repeat this as often as you like, so mount several configs into your container (so the runtime-version of the image). You will create those configs on the host before running the container and need to ship those files with the container, which is the downside of this approach (portability)
- **Docker run –v /root/data:/usr/share/nginx/html –p 5000:80 nginx**
    1- Run Docker with Mount the files under **/root/data** inside container path **:/usr/share/nginx/html**

- **So the master container doesn't save the data instead I save my configuration data outside the container and when I run the container I mount the my configuration into the container**
- **#Docker commit <Container ID>:** to save a new version from my modified container as image

Docker Cluster by Swarm

- **Docker swarm init:** to create Docker cluster.
- **Docker node ls:** will list all nodes in cluster.
- **Docker swarm leave:** to leave Docker cluster.

Swarm mode routing mesh

Docker Engine swarm mode makes it easy to publish ports for services to make them available to resources outside the swarm. All nodes participate in an ingress **routing mesh**. The routing mesh enables each node in the swarm to accept connections on published ports for any service running in the swarm, even if there's no task running on the node.

Example Mesh Service: if I have cluster with 3 nodes (1 manager and 2 worker) and I have Docker service on one node I can access this service from node1 IP or node2 IP or node3 IP.

Use the Docker CLI to create a swarm, deploy application services to a swarm, and manage swarm behavior.

Feature highlights

- **Cluster management integrated with Docker Engine:** Use the Docker Engine CLI to create a swarm of Docker Engines where you can deploy application services. You don't need additional orchestration software to create or manage a swarm.
- **Decentralized design:** Instead of handling differentiation between node roles at deployment time, the Docker Engine handles any specialization at runtime. You can deploy both kinds of nodes, managers and workers, using the Docker Engine. This means you can build an entire swarm from a single disk image.
- **Declarative service model:** Docker Engine uses a declarative approach to let you define the desired state of the various services in your application stack. For example, you might describe an application comprised of a web front end service with message queueing services and a database backend.
- **Scaling:** For each service, you can declare the number of tasks you want to run. When you scale up or down, the swarm manager automatically adapts by adding or removing tasks to maintain the desired state.
- **Desired state reconciliation:** The swarm manager node constantly monitors the cluster state and reconciles any differences between the actual state and your expressed desired state. For example, if you set up a service to run 10 replicas of a container, and a worker machine hosting two of those replicas crashes, the manager creates two new replicas to replace the replicas that crashed. The swarm manager assigns the new replicas to workers that are running and available.
- **Multi-host networking:** You can specify an overlay network for your services. The swarm manager automatically assigns addresses to the containers on the overlay network when it initializes or updates the application.

- **Service discovery:** Swarm manager nodes assign each service in the swarm a unique DNS name and load balances running containers. You can query every container running in the swarm through a DNS server embedded in the swarm.
- **Load balancing:** You can expose the ports for services to an external load balancer. Internally, the swarm lets you specify how to distribute service containers between nodes.
- **Secure by default:** Each node in the swarm enforces TLS mutual authentication and encryption to secure communications between itself and all other nodes. You have the option to use self-signed root certificates or certificates from a custom root CA.
- **Rolling updates:** At rollout time you can apply service updates to nodes incrementally. The swarm manager lets you control the delay between service deployment to different sets of nodes. If anything goes wrong, you can roll back to a previous version of the service.

Docker service vs Docker run

Run: The Docker run command creates and starts a container on the local Docker host.

Service: A Docker "service" is one or more containers with the same configuration running under Docker's swarm mode. It's similar to Docker run in that you spin up a container. The difference is that you now have orchestration. That orchestration restarts your container if it stops, finds the appropriate node to run the container on based on your constraints, scale your service up or down, allows you to use the mesh networking and a VIP to discover your service, and perform rolling updates to minimize the risk of an outage during a change to your running application.

- **Docker run (Image Name):** first will search in local repository if exist will run the container if not will go to public repository and download it then run.
- **Docker service create (Image Name):** to create container as a service and can create multiple container from image at the same time on nodes cluster to make load balance and can work with mesh technology.
- **Mesh Service:** if I have cluster with 3 nodes (1 manager and 2 worker) and I have Docker service on one node I can access this service from node1 IP or node2 IP or node3 IP.
- **Docker service scale mysite=5 :** this will make 5 containers from container mysite.
- **Docker service update --image=Emam/mysite-v2 mysite:** (Emam /mysite-v2: public image on my repository Docker hub) (mysite my service will be updated) <this command will download container and update/replace my container service (mysite)to v2).
- **Docker service rollback mysite :** this will return back to the old version of container mysite.

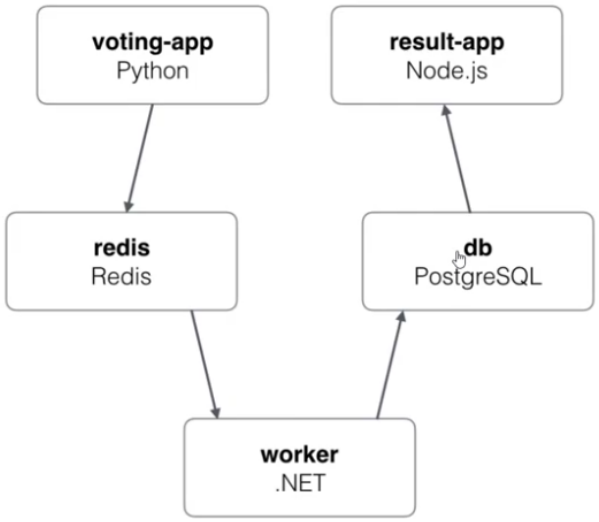
- **Docker service rm mysite:** will remove service container mysite.
- **Docker service ls:** list all Docker services.

Docker Stack

This is a cluster management command, and must be executed on a swarm manager node to deploy different images as service as full project.

- **Docker stack deploy --compose-file docker-compose.yml voting:** (docker-compose.yml<config file>) (voting <stack name>)
- Create and update a stack from a compose file on the swarm.
- **Compose file `.yml` file:** this is a file contain the configuration of stack
- Below example is application from 5 different service **(nodeJs, Python, DB,Redis,.NET)** and I can just create a config file .yml then run the below command to create stack and it will download and create all the requested component (network, services … etc.,)
- **Docker stack deploy --compose-file docker-compose.yml voting:** (docker-compose.yml<config file>) (voting <stack name>)

Architecture

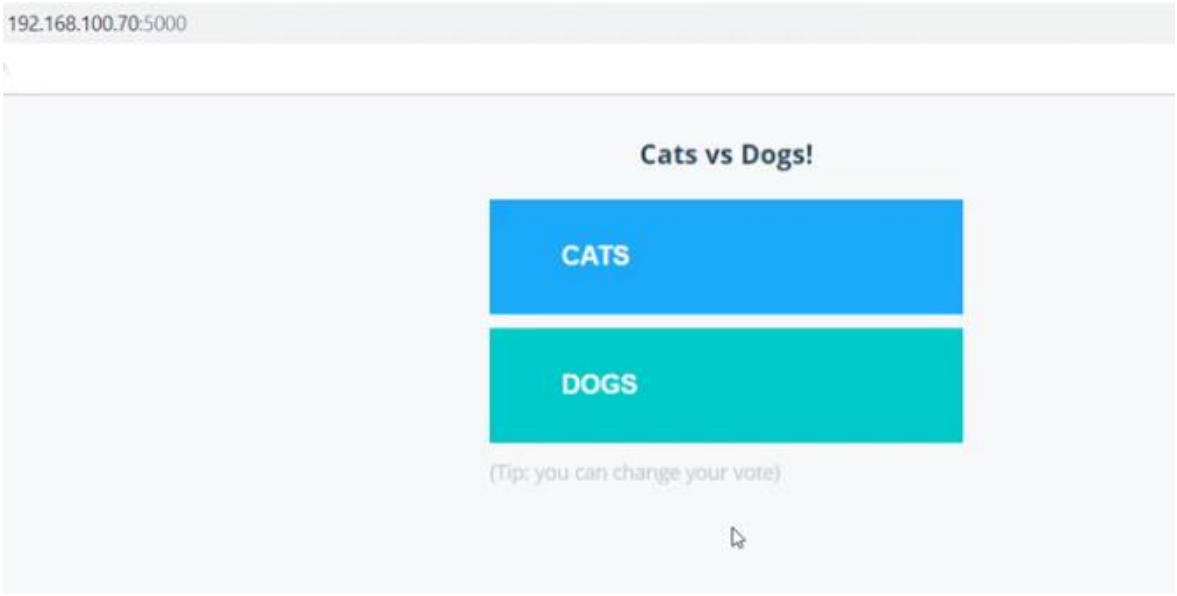


```
[root@master ~]# docker stack deploy --compose-file docker-stack.yml vote
open docker-stack.yml: no such file or directory
[root@master ~]#
[root@master ~]#
[root@master ~]# docker stack deploy --compose-file vote.yml vote
Creating network vote_frontend
Creating network vote_backend
Creating network vote_default
Creating service vote_vote
Creating service vote_result
Creating service vote_worker
Creating service vote_visualizer
Creating service vote_redis
Creating service vote_db
[root@master ~]#
```

It creates 5 services and another one GUI

```
[root@master ~]# docker service ls
ID             NAME              MODE         REPLICAS   IMAGE                                          PORTS
4773rdnxw05d   vote_db           replicated   0/1        postgres:9.4
lcrxjnevkb0w   vote_redis        replicated   0/1        redis:alpine
fgbtxdqdfuns   vote_result       replicated   0/1        dockersamples/examplevotingapp_result:before   *:5001->80/tcp
2h7brm445702   vote_visualizer   replicated   0/1        dockersamples/visualizer:stable                *:8080->8080/tcp
8ghwgzf874r8   vote_vote         replicated   2/2        dockersamples/examplevotingapp_vote:before     *:5000->80/tcp
q42bfcbq1naq   vote_worker       replicated   0/1        dockersamples/examplevotingapp_worker:latest
[root@master ~]#
```

And now accessed the application and working fine



So I created a different containers as a Docker service to make an application of different services.

Container Management Platforms

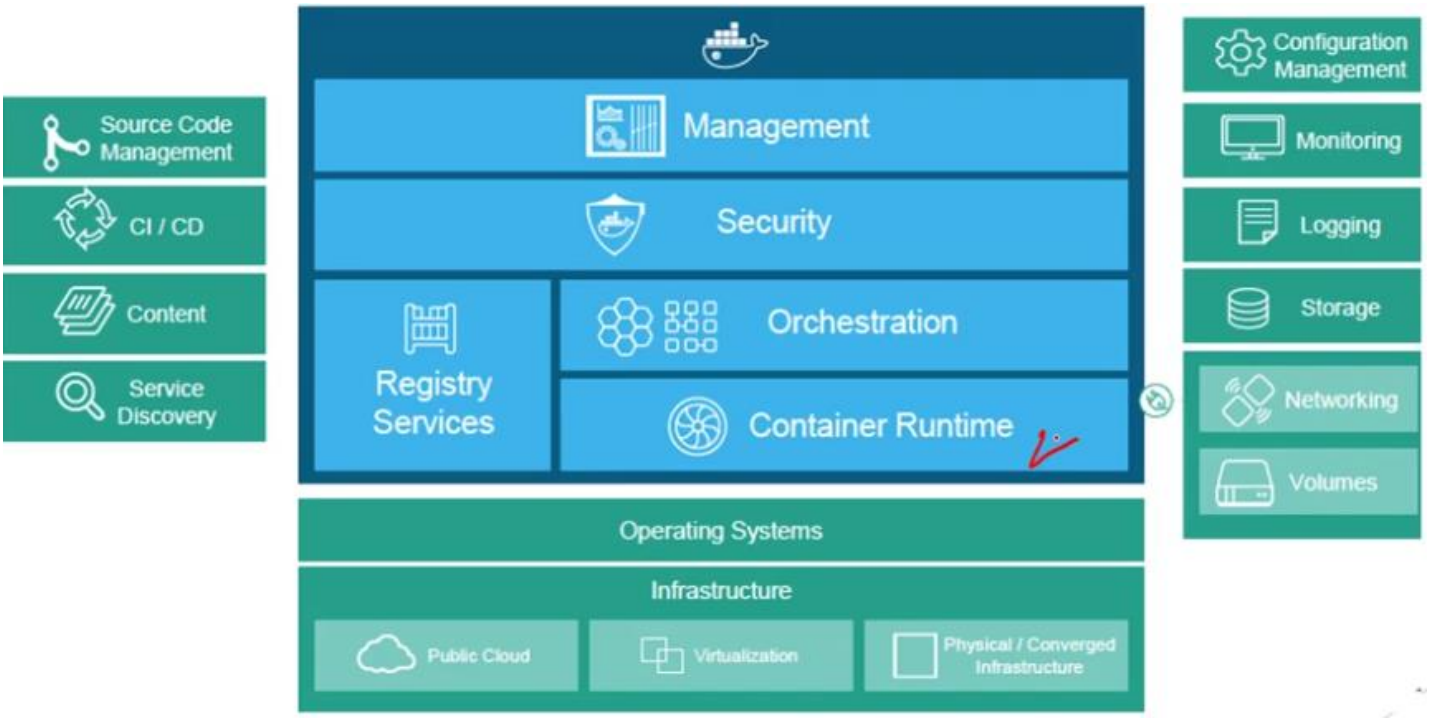
Container Management is the process of organizing, adding or replacing large numbers of software containers. Container management uses software to automatically create, deploy and scale containers. This gives rise to the need for container orchestration—a more specialized tool that automates the deployment, management, scaling, networking, and availability of container-based applications. For example, Kubernetes manages app health, replication, load balancing, and hardware resource allocation for you.

Container management uses a platform to organize software containers, which may also be referred to as operating-system-level virtualizations.

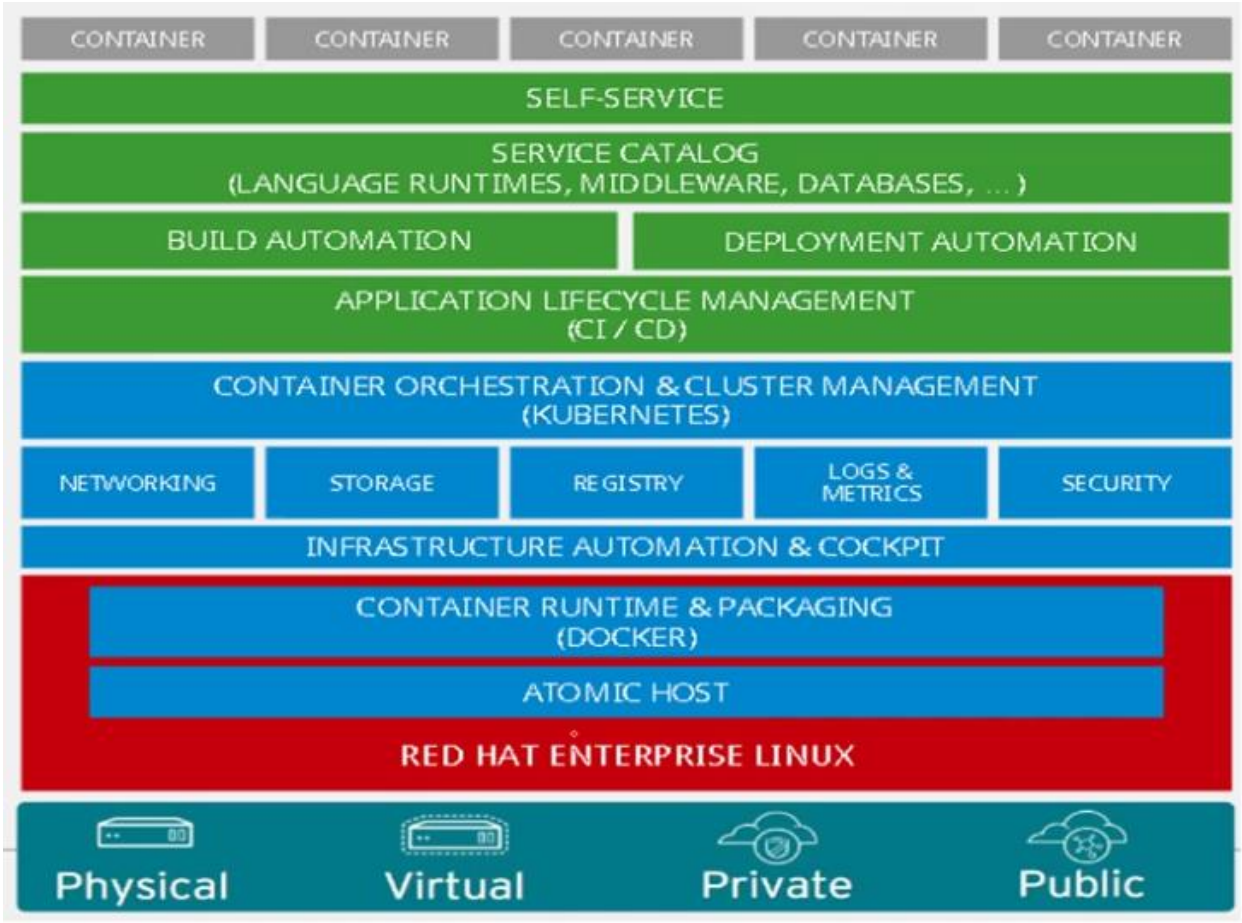
Containers need management services because a vast number of containers can become too complex for an IT team to handle.

- Organize containers and lunch containers instance.
- Orchestrate and cluster container groupings.
- Automate or schedule container execution.
- Replicate container for simultaneous execution.
- Security and control.
- Network, security.
- Storage.

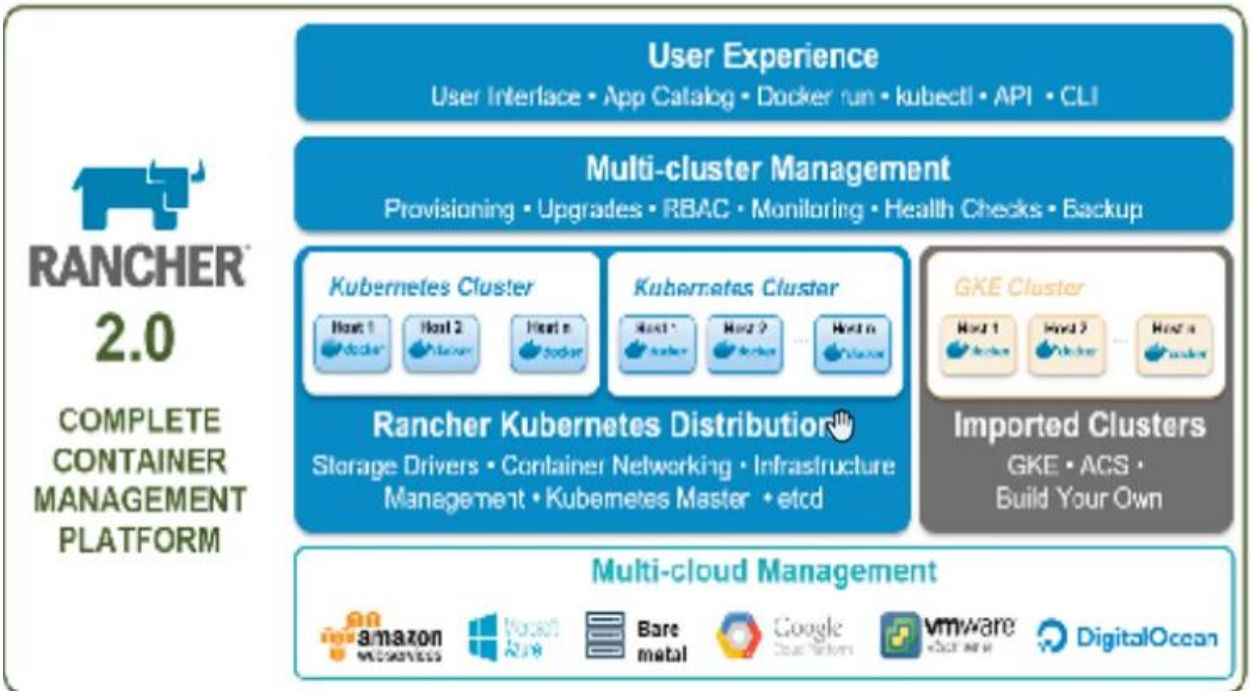
Docker Datacenter Platform



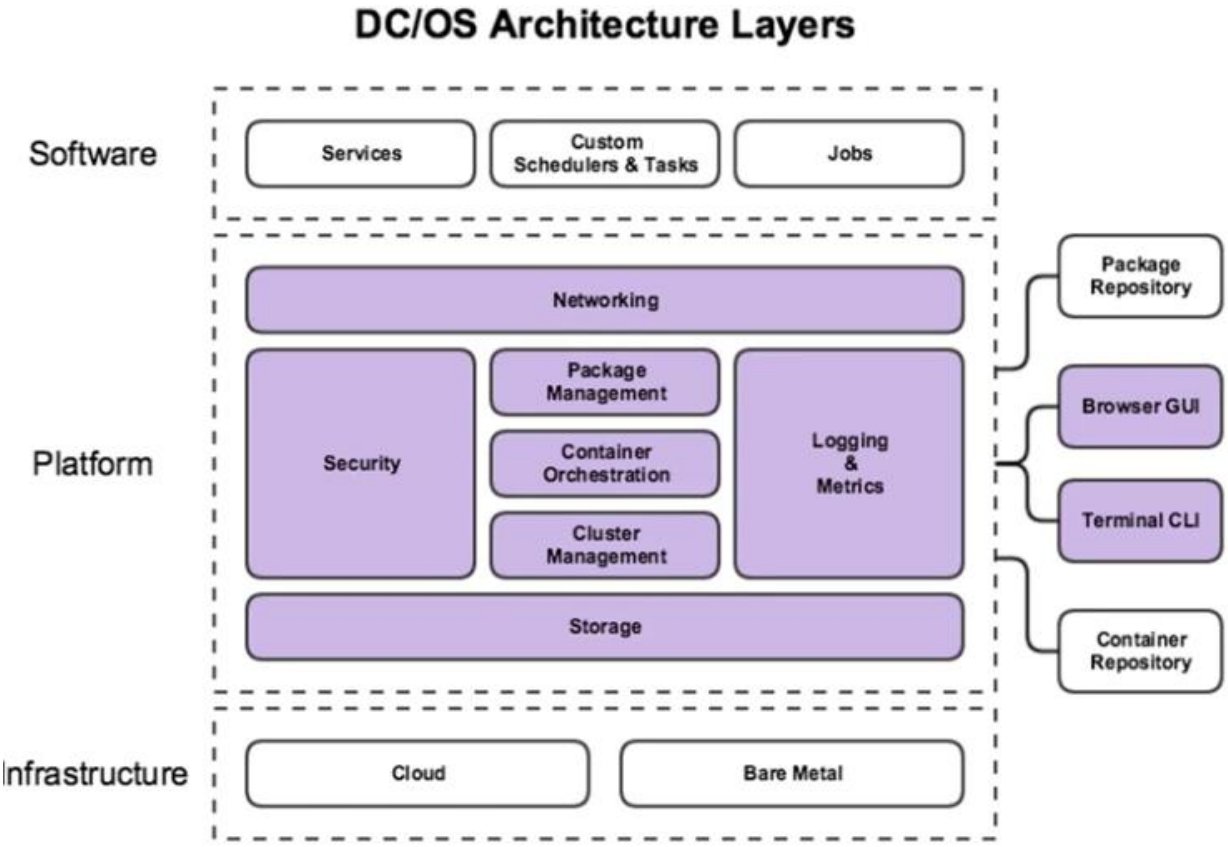
RedHat OpenShift Platform



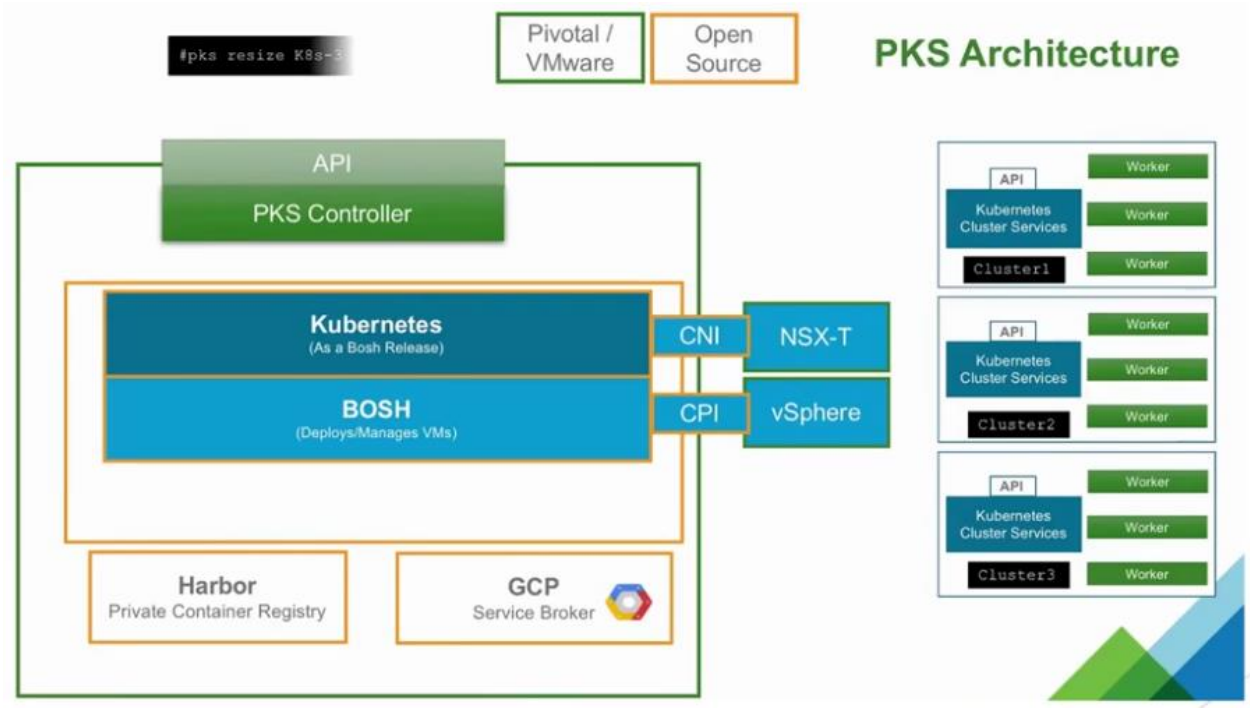
Rancher Platform



MesosPhere DC/OS



VMware & Pivotal PKS (VMware Enterprise PKS) Platform



VMware Enterprise PKS enables operators to provision, operate, and manage enterprise-grade Kubernetes clusters using BOSH and Pivotal Ops Manager.