## **What Is Docker ?**

Docker is a tool designed to make it easy to develop, build, deploy & manage applications by using containers. Contianers allow a developer to package up an application with all of it needs such as libraries & other dependencies and ship it all out as one package in different platformss.

**It’s a Isolated area of an OS with resource usage limits applied.**

**What can I use Docker for?**

**Fast, consistent delivery of your applications**

* Docker streamlines the development lifecycle by allowing developers to work in standardized environments using local containers which provide your applications and services. Containers are great for continuous integration n and continuous delivery (CI/CD) workflows.
* Consider the following example scenario:
* Your developers write code locally and share their work with their colleagues using Docker containers.
* They use Docker to push their applications into a test environment and execute automated and manual tests.
* When developers find bugs, they can fix them in the development environment and redeploy them to the test environment for testing and validation.
* When testing is complete, getting the fix to the customer is as simple as pushing the updated image to the production environment.

**Responsive deployment and scaling**

* Docker’s container-based platform allows for highly portable workloads. Docker containers can run on a developer’s local laptop, on physical or virtual machines in a data center, on cloud providers, or in a mixture of environments.
* Docker’s portability and lightweight nature also make it easy to dynamically manage workloads, scaling up or tearing down applications and services as business needs dictate, in near real time.

**Running more workloads on the same hardware**

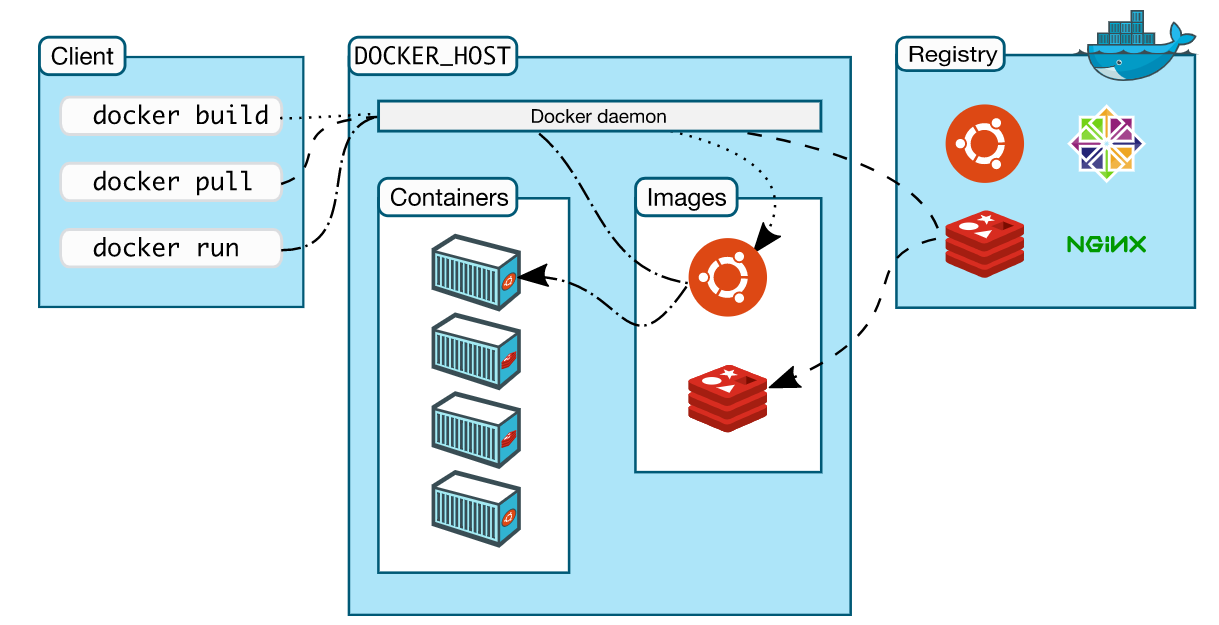
* Docker is lightweight and fast. It provides a viable, cost-effective alternative to hypervisor-based virtual machines, so you can use more of your compute capacity to achieve your business goals. Docker is perfect for high density environments and for small and medium deployments where you need to do more with fewer resources.

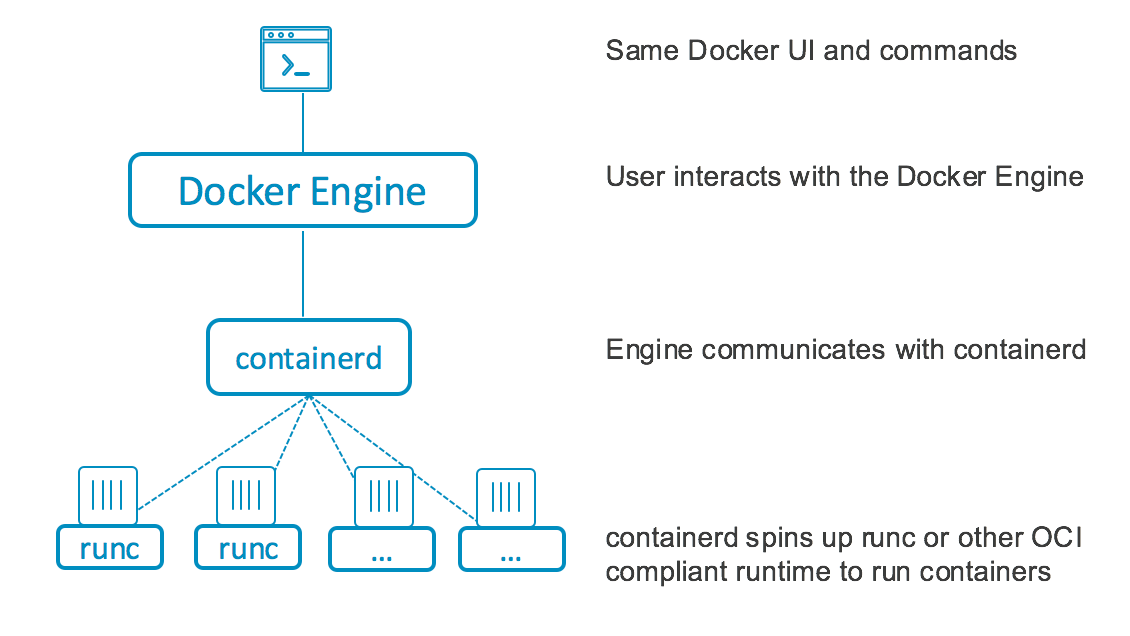
Benefits

* Reduced IT management resources
* Reduced size of snapshots
* Quicker spinning up apps
* Reduced and simplified security updates
* Less code to transfer, migrate, and upload workloads

## **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.





We use the command line to create a new container $ docker container create, the client takes the command and makes the appropriate API request to the **containers/create endpoint** here in the engine. And engine here pulls together all of the required kernel stuff (namespace, control group…etc) && pops out the container.

**Kernel Internals**

* **NamesSpaces**

Namespace are about isolation.

These let us take an operating system and carve it into multiple isolated virtual operating systems. Its bit like hypervisor & VM. SO in the hypervisor world we take a single physical machine with all of its resources like CPU and RAM, we carve out one or more VM and each one gets a down slice of virtual CPU, mem, networking, storage …

Well in the container world we use the namespaces to take a single operating system with all of its resources which tends to be higher level contucts like file-systems , process tree and users and we carve all of that up into multiple virtual operating system called containers, well each container gets its own virtual or containerized root filesystems, its own process tree, its own eth0 interface. So each container looks and feels exactly like a regular OS. Only its not.

Each Contianer is the collection of below namespaces

* + - Process ID [Pid namespace]
    - Network namespace
    - Filesystem/mount [mnt]
    - Inter-proc comm [ipc]
    - UTS [uts] >>> gives hostname
    - User [user]
* **Control Group**

We use two main building blocks when we are building containers namespaces & control groups. Both of them linux kernel primitives.

Control groups are about grouping objects and setting limits.

Control groups are setting limits like cpu/ram/mememory to each container.

Now a uninion file system or some way of combining a bunch of read only filesystem or block devices, lashing a wribtable layer on top and presenting to the system is a unifie view.

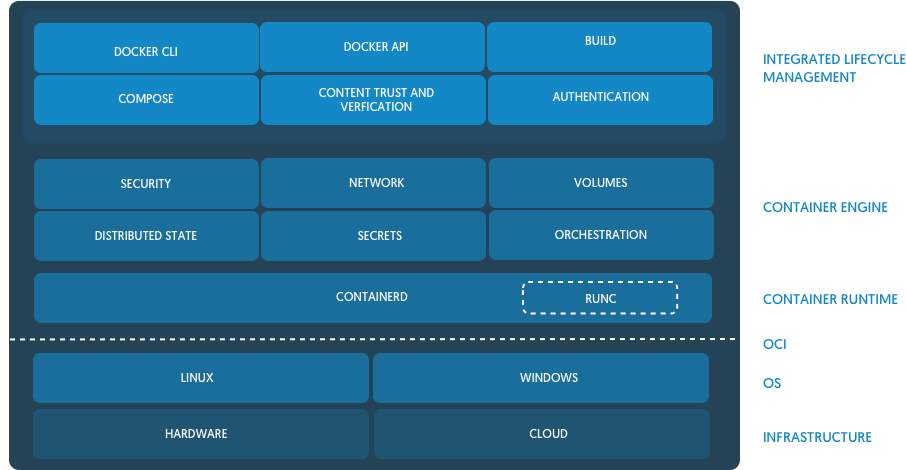
Graphical user interface, application

Description automatically generated

**Architecture**

Diagram

Description automatically generated



Diagram, timeline

Description automatically generated

Graphical user interface, application

Description automatically generated

Docker is a broad set of technologies that are used to work with containers. containerd is an example of a container runtime. A container runtime is that process that does the actual work of creating, running, and destroying containers. Docker uses containerd as its runtime.

As mentioned earlier, Docker technologies allow you to interact with the container runtime. For example, when you execute the following command **$ docker run nginx**

You are using the command-line interface (CLI) tool, docker. When the command set docker run is invoked, docker tells the container runtime to create a container based on the container image, nginx. Behind the scenes, containerd takes over and downloads the associated container image. Then containerd does the work of creating the container from that image.

The host operating system has no concept of a container. However, it does provide the features such as namespaces, cgroups and file system overlays that make a container possible. containerd along with another component called a low-level runtime, in this case, runc, does the work of interacting with the host operating system’s kernel to do the low-level workdsthat goes with creating a container.

**Containers are no longer tightly coupled with the name Docker**. You can be running containers with Docker, or a bunch of other tools which aren’t Docker. docker is just one of the many options, and Docker (the company) backs some of the tools in the ecosystem, but not all.

The main standards around containers that you should be aware of (although you don’t need to know all the details) are:

**The Open Container Initiative (OCI) which publishes specifications for containers and their images.**

The Kubernetes Container Runtime Interface (CRI), which defines an API between Kubernetes and a container runtime underneath. docker is designed to be installed on a workstation or server and comes with a bunch of tools to make it easy to build and run containers as a developer, or DevOps person.The docker command line tool can build container images, pull them from registries, create, start and manage containers.To make all of this happen, the experience you know as docker is now made up of these projects (there are others, but these are the main ones):

**docker-cli:** This is the command-line utility that you interact with using docker ... commands.

**containerd:** This is a daemon process which manages and runs containers. It pushes and pulls images, manages storage and networking, and supervises the running of containers.

**runc:** This is the low-level container runtime, or the thing that actually creates and runs containers). It includes libcontainer, a native Go-based implementation for creating containers.

**In reality, when you run a container with docker, you’re actually running it through the Docker daemon, containerd, and then runc.**

**Dockershim: Docker in Kubernetes**

* Kubernetes includes a component called dockershim, which allows it to support Docker.
* Kubernetes prefers to run containers through any container runtime which supports its Container Runtime Interface (CRI). But Docker, being older than Kubernetes, doesn’t implement CRI. So that’s why the dockershim exists, to basically bolt Docker onto Kubernetes. Or Kubernetes onto Docker, whichever way round you prefer to think of it. Going forward, Kubernetes will remove support for Docker directly, and prefer to use only container runtimes that implement its Container Runtime Interface. This probably means using containerd or CRI-O.
* But this doesn’t mean that Kubernetes won’t be able to run Docker-formatted containers. Both containerd and CRI-O can run Docker-formatted (actually OCI-formatted) images, they just do it without having to use the docker command or the Docker daemon.

**Docker images**

* What many people refer to as Docker images, are actually images packaged in the **Open Container Initiative (OCI) format**. So if you pull an image from Docker Hub, or another registry, you should be able to use it with the docker command, or on a Kubernetes cluster, or with the podman utility, or any other tool that supports the OCI image format spec.
* This is the benefit of having an open standard – anybody can write software that supports the standard.

**Container Runtime Interface (CRI)**

* CRI is the protocol that Kubernetes uses to control the different runtimes that create and manage containers.
* CRI is an abstraction for any kind of container runtime you might want to use. So CRI makes it easier for Kubernetes to use different container runtimes.
* Instead of the Kubernetes project needing to manually add support for each runtime, the CRI API describes how Kubernetes interacts with each runtime. So then, it’s down to the runtime to actually manage containers. As long as it obeys the CRI API, it can do whatever it likes.
* So if you prefer to use containerd to run your containers, you can. Or, if you prefer to use CRI-O, then you can. This is because both of these runtimes implement the CRI specification.
* If you’re an end user (like a developer), the implementation mostly shouldn’t matter. There are subtle differences between different CRI implementations but they are intended to be pluggable and seamlessly changeable.
* Your choice of runtime might be important if you pay to get support (security, bug fixes etc) from a vendor. For example, Red Hat’s OpenShift uses CRI-O, and offers support for it. Docker provides support for their own containerd.

**containerd**

* containerd is a high-level container runtime that came from Docker, and implements the CRI spec. It pulls images from registries, manages them and then hands over to a lower-level runtime, which actually creates and runs the container processes.
* containerd was separated out of the Docker project, to make Docker more modular. So Docker uses containerd internally itself. When you install Docker, it will also install containerd. containerd implements the Kubernetes Container Runtime Interface (CRI), via its cri plugin.

**CRI-O**

* CRI-O is another high-level container runtime which implements the Container Runtime Interface (CRI). It’s an alternative to containerd. It pulls container images from registries, manages them on disk, and launches a lower-level runtime to run container processes.
* Yes, CRI-O is another container runtime. It was born out of Red Hat, IBM, Intel, SUSE and others.
* It was specifically created from the ground up as a container runtime for Kubernetes. It provides the ability to start, stop and restart containers, just like containerd.

**Open Container Initiative (OCI)**

* The OCI is a group of tech companies who maintain a specification for the container image format, and how containers should be run.
* The idea behind the OCI is that you can choose between different runtimes which conform to the spec. Each of these runtimes have different lower-level implementations.
* For example, you might have one OCI-compliant runtime for your Linux hosts, and one for your Windows hosts.
* This is the benefit of having one standard that can be implemented by many different projects. This same “one standard, many implementations” approach is in use everywhere, from Bluetooth devices to Java APIs.

**runc**

* runc is an OCI-compatible container runtime. It implements the OCI specification and runs the container processes. runc is called the reference implementation of OCI.
* runc provides all of the low-level functionality for containers, interacting with existing low-level Linux features, like namespaces and control groups. It uses these features to create and run container processes.

**A couple of alternatives to runc are:**

* crun a container runtime written in C (by contrast, runc is written in Go.)
* kata-runtime from the Katacontainers project, which implements the OCI specification as individual lightweight VMs (hardware virtualisation)
* gVisor from Google, which creates containers that have their own kernel. It implements OCI in its runtime called runsc.

**The Beginnings**

* At its release in 2013, Docker was a self-contained project with everything you needed to build and run containers. What it lacked was an easy way to orchestrate container deployments in the cloud.
* By the end of 2013, a group of Googlers were already addressing this with a prototype of what would become Kubernetes. Kubernetes is intended to simplify the operation of containerised workloads across large fleets of machines.
* Back in those early days, Kubernetes was inextricably linked to Docker. It used Docker directly to interact with containers, even though it only needed a subset of functionality – the parts responsible for actually running containers.
* Docker’s developer-centric UI got in the way of Kubernetes. It had to bypass the human-friendly aspects of the project using a dedicated tool, Dockershim. The issues were compounded by the differing directions in which Docker and Kubernetes were headed. Docker launched Swarm, its own Kubernetes alternative, offering orchestration as a built-in Docker “mode”.

**The Rise of Containerd**

* As Kubernetes grew and more third-party tools arose around Docker, the limitations of its architecture became clear. At the same time, the Open Container Initiative (OCI) began standardising container formats and runtimes. This resulted in an OCI specification defining a container which could be used by multiple runtimes, of which Docker is an example.
* Docker then extracted its container runtime out into a new project, containerd. This includes Docker’s functionality for executing containers, handling low-level storage and managing image transfers. Containerd was donated to the Cloud Native Computing Foundation (CNCF) in order to provide the container community with a basis for creating new container solutions.
* The emergence of containerd makes it easier for projects like Kubernetes to access the low-level “Docker” elements they need. Instead of actually using Docker, they now have a more accessible interface to the container runtime. The OCI standardisation of container technologies means other runtimes can be used too.

**Understanding Containerd’s Role**

* To fully understand containerd, it’s necessary to look at the nature of containers. Containers are really an abstraction over various Linux kernel features. In order to run a container, you need to use syscalls to set up the containerised environment. The steps vary by platform and distribution.
* Containerd drops in to abstract this low-level wiring. It’s intended as a “client layer” that container software then builds atop of. This might be developer-oriented software, like Docker, or cloud-oriented devops tools such as Kubernetes.
* Previously, Kubernetes development was left with two bad options: keep writing shims around the hefty Docker interface, or start interacting with Linux kernel features directly. By breaking containerd out of Docker, a third alternative became available: use containerd as a system abstraction layer, without involving Docker.

**Here’s a summary of how the three technologies combine:**

**Docker –** A developer-oriented software with a high level interface that lets you easily build and run containers from your terminal. It now uses containerd as its container runtime.

**Containerd –** An abstraction of kernel features that provides a relatively high level container interface. Other software projects can use this to run containers and manage container images.

**Kubernetes –** A container orchestrator that works with multiple container runtimes, including containerd. Kubernetes is focused on deploying containers in aggregate across one or more physical “nodes.” Historically, Kubernetes was tied to Docker.

Containerd is only one container backend. Other containers implementing the Open Containers Runtime specification include runC and CRI-O. These runtimes can also be used with Docker and Kubernetes; each has its own distinctions.

**The OCI**

* The OCI is the body responsible for defining container standards. Its work has been instrumental in facilitating the interoperability between different component technologies.
* The OCI’s image specification defines what a container should look like. The runtime specification sets out an interface for running containers. Projects like containerd then implement these specifications.
* Importantly, one of the OCI’s priorities is to support the container usage experience popularised by Docker. Its images must be executable on the target platform without any user-defined arguments (e.g. docker run hello-world:latest). OCI images must therefore contain sufficient metadata to enable this automatic configuration.
* You may also see references to the Container Runtime Interface (CRI). This is a Kubernetes-specific abstraction over the OCI specification. The CRI builds on the OCI specs to enable support for interchangeable container runtimes within Kubernetes.

**What About My Docker Images?**

* Images you build with Docker aren’t really “Docker images” at all. As Docker now uses the containerd runtime, your images are built in the standardised Open Container Initiative (OCI) format.
* You shouldn’t need to worry about incompatibilities between your Docker images and the environment they’re used in. Images you build with Docker can still be deployed using Kubernetes. This is because Kubernetes also supports OCI images, through its use of containerd (and other standards-compliant runtimes). It’s up to the runtime to handle the pulling and running of images, not the high level interface which tools like Docker and Kubernetes provide.

**Kubernetes and Docker**

* Kubernetes deprecated the Docker runtime in late 2020. It will be removed in a future release, currently scheduled for late 2021. After that, Kubernetes will no longer offer Docker runtime support. An alternative runtime compatible with the OCI specs, such as containerd, will need to be used instead.
* This announcement prompted concern about the implications for developers. The change shouldn’t impact most existing workflows. As we’ve already seen, Docker produces OCI-compliant images which OCI-compliant runtimes can run. Any images you build with docker build will still work within Kubernetes, even after the Docker runtime is removed.
* Two different technologies are being considered – the Docker command-line interface used to create and run containers, and the Docker runtime which the command-line interface wraps around.

**The Docker Engine**

Diagram

Description automatically generated

First, let us look take a look at Docker Engine and its components so we have a basic idea of how the system works. Docker Engine allows you to develop, assemble, ship, and run applications using the following components:



**The Docker daemon**[**🔗**](https://docs.docker.com/get-started/overview/#the-docker-daemon)

The Docker daemon (dockerd) 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.

 A persistent background process that manages Docker images, containers, networks, and storage volumes. The Docker daemon constantly listens for Docker API requests and processes them.

**Docker Engine REST API**: An API used by applications to interact with the Docker daemon; it can be accessed by an HTTP client.

**The Docker client**[**🔗**](https://docs.docker.com/get-started/overview/#the-docker-client)

The Docker client (docker) 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 registries**[**🔗**](https://docs.docker.com/get-started/overview/#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.

When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

**DockerHost**

The Docker host provides a complete environment to execute and run applications. It comprises of the Docker daemon, Images, Containers, Networks, and Storage. As previously mentioned, the daemon is responsible for all container-related actions and receives commands via the CLI or the REST API. It can also communicate with other daemons to manage its services. The Docker daemon pulls and builds container images as requested by the client. Once it pulls a requested image, it builds a working model for the container by utilizing a set of instructions known as a build file. The build file can also include instructions for the daemon to pre-load other components prior to running the container, or instructions to be sent to the local command line once the container is built.

**Docker objects**[**🔗**](https://docs.docker.com/get-started/overview/#docker-objects)

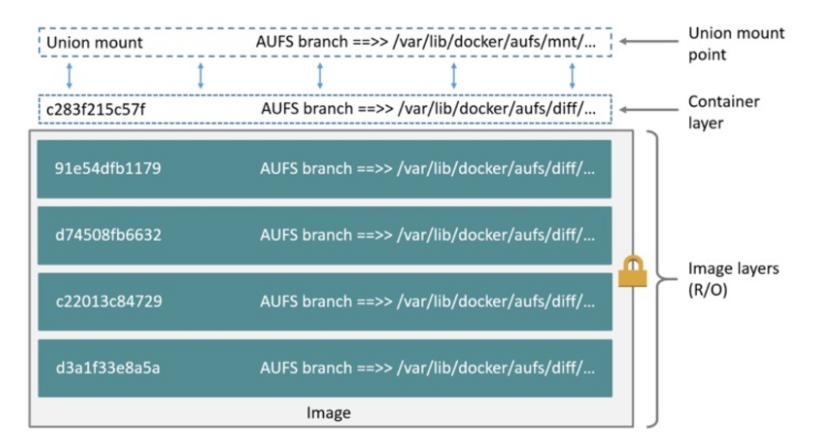
When you use Docker, you are creating and using images, containers, networks, volumes, plugins, and other objects. This section is a brief overview of some of those objects.

**Image**

* Build TIME [R/O]
* An image is executable package that includes everything needed to run an application, the code, runtime evn libraries, dependencies, config files etc.
* Complete OS without kernel
* An image is a read-only template with instructions for creating a Docker container. It is a software file containing a snapshot of a full file system everything necessary to launch a viable virtual server in included. An image might contains of just of a base OS like ubuntu or alpine but image also includes additional layers with s/w app like web servers & databases. When image is launched as acontainer an extra writable layer is automatically added

**Contianer**

* RUN TIME [R/W]
* A container is a runnable instance of an image. Runtime of an image. Instead of modifying the file inside the image, the container locates it in writable layer & makes the changes ther.
* Container get full READ WRITE access without having write access to the image.
* Container is a thin writable layer on top of the read only image. Every container gets it own file system process tree , network stack.



**Networking**

Docker implements networking in an application-driven manner and provides various options while maintaining enough abstraction for application developers. There are basically two types of networks available – the default Docker network and user-defined networks. By default, you get three different networks on the installation of Docker – none, bridge, and host. The none and host networks are part of the network stack in Docker.

The bridge network automatically creates a gateway and IP subnet and all containers that belong to this network can talk to each other via IP addressing. This network is not commonly used as it does not scale well and has constraints in terms of network usability and service discovery.

The other type of networks is user-defined networks. Administrators can configure multiple user-defined networks. There are three types:

**Bridge network**: Similar to the default bridge network, a user-defined Bridge network differs in that there is no need for port forwarding for containers within the network to communicate with each other. The other difference is that it has full support for automatic network discovery.

**Overlay network:** An Overlay network is used when you need containers on separate hosts to be able to communicate with each other, as in the case of a distributed network. However, a caveat is that swarm mode must be enabled for a cluster of Docker engines, known as a swarm, to be able to join the same group.

**Macvlan network**: When using Bridge and Overlay networks a bridge resides between the container and the host. A Macvlan network removes this bridge, providing the benefit of exposing container resources to external networks without dealing with port forwarding. This is realized by using MAC addresses instead of IP addresses.

**Storage**

You can store data within the writable layer of a container but it requires a storage driver. Being non-persistent, it perishes whenever the container is not running. Moreover, it is not easy to transfer this data. In terms of persistent storage, Docker offers four options:

**Data Volumes:** Data Volumes provide the ability to create persistent storage, with the ability to rename volumes, list volumes, and also list the container that is associated with the volume. Data Volumes sit on the host file system, outside the containers copy on write mechanism and are fairly efficient.

**Data Volume Container:** A Data Volume Container is an alternative approach wherein a dedicated container hosts a volume and to mount that volume to other containers. In this case, the volume container is independent of the application container and therefore can be shared across more than one container.

**Directory Mounts**: Another option is to mount a host’s local directory into a container. In the previously mentioned cases, the volumes would have to be within the Docker volumes folder, whereas when it comes to Directory Mounts any directory on the Host machine can be used as a source for the volume.

**Storage Plugins:** Storage Plugins provide the ability to connect to external storage platforms. These plugins map storage from the host to an external source like a storage array or an appliance. A list of storage plugins can be found on Docker’s Plugin page.

Storage Plugins

There are storage plugins from various companies to automate the storage provisioning process. For example,

HPE 3PAR

EMC (ScaleIO, XtremIO, VMAX, Isilon)

NetApp

There are also plugins that support public cloud providers like:

Azure File Storage

Google Compute Platform.

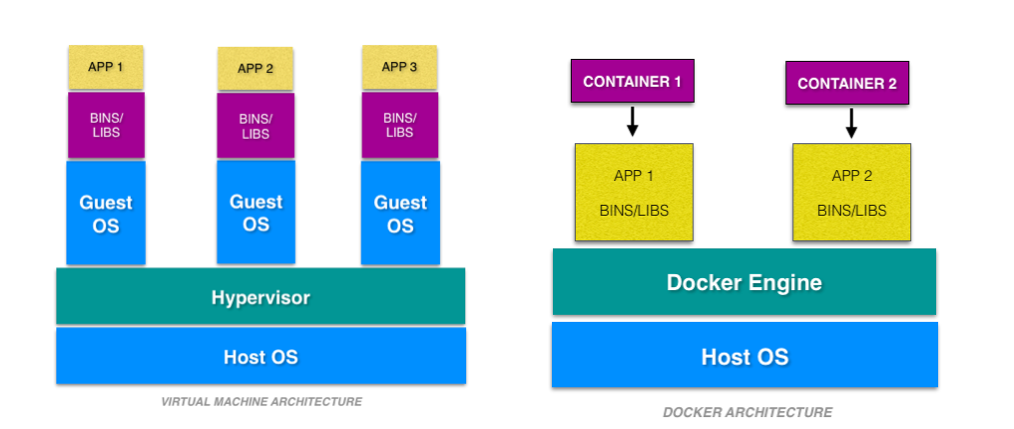
**Service Discovery**

Service Discovery allows containers to find out about the environment they are in and find other services offered by other containers.

It is an important factor when trying to build scalable and flexible applications.

## **Docker vs Virtual Machine**

1. Docker vs. Virtual Machines: OS Support and Architecture



Virtual machines have host OS and the guest OS inside each VM. Guest OS can be any OS, like Linux or Windows, irrespective of host OS. In contrast, Docker containers host on a single physical server with a host OS, which shares among them. Sharing the host OS between containers makes them light and increases the boot time. Docker containers are considered suitable to run multiple applications over a single OS kernel; whereas, virtual machines are needed if the applications or services required to run on different OS.

1. Docker vs. Virtual Machines: Security

Virtual machines are stand-alone with their kernel and security features. Therefore, applications needing more privileges and security run on virtual machines.

On the flip side, providing root access to applications and running them with administrative premises is not recommended in the case of Docker containers because containers share the host kernel. The container technology has access to the kernel subsystems; as a result, a single infected application is capable of hacking the entire host system.

3. Docker vs. Virtual Machines: Portability

Virtual machines are isolated from their OS, and so, they are not ported across multiple platforms without incurring compatibility issues. At the development level, if an application is to be tested on different platforms, then Docker containers must be considered.

Docker containers packages are self-contained and can run applications in any environment, and since they don’t need a guest OS, they can be easily ported across different platforms. Docker containers can be easily deployed in servers since containers being lightweight can be started and stopped in very less time compared to virtual machines.

4**. Docker vs. Virtual Machines: Performance**

Virtual machines are more resource-intensive than Docker containers as the virtual machines need to load the entire OS to start. The lightweight architecture of Docker containers is less resource-intensive than virtual machines.

In the case of virtual machines, resources like CPU, memory, and I/O may not be allocated permanently to containers — unlike in the case of containers, where the resource usage with the load or traffic.

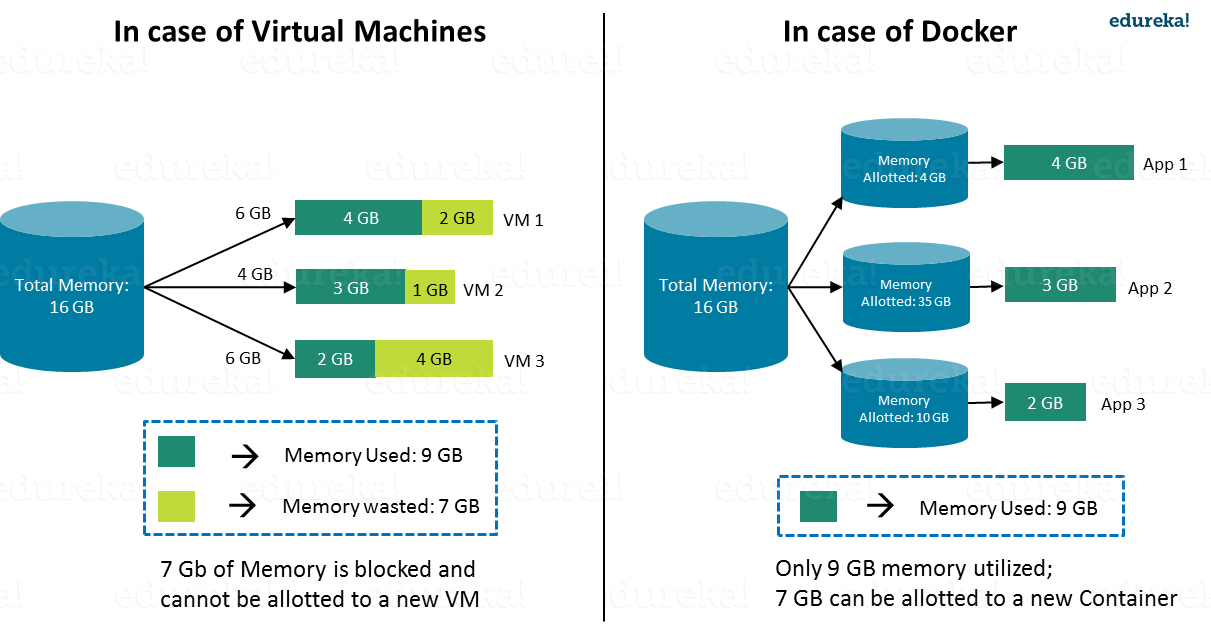
Scaling up and duplicating containers in simple and easy as compared to virtual machines because there is no need to install an operating system in them.

|  |  |  |
| --- | --- | --- |
|  | Docker | VM |
| **Boot-Time** | Boots in a few seconds. | It takes a few minutes for VMs to boot. |
| **Runs on** | Dockers make use of the execution engine. | VMs make use of the hypervisor. |
| **Memory Efficiency** | No space is needed to virtualize, hence less memory. | Requires entire OS to be loaded before starting the surface, so less efficient. |
| **Isolation** | Prone to adversities as no provisions for isolation systems. | Interference possibility is minimum because of the efficient isolation mechanism |
| **Deployment** | Deploying is easy as only a single image, containerized can be used across all platforms. | Deployment is comparatively lengthy as separate instances are responsible for execution. |
| **Usage** | Docker has a complex usage mechanism consisting of both third party and docker managed tools. | Tools are easy to use and simpler to work with. |

Virtual Machine and Docker Container are compared on the following three parameters:

* Size – This parameter will compare Virtual Machine & Docker Container on their resource they utilize.
* Startup – This parameter will compare on the basis of their boot time.
* Integration – This parameter will compare on their ability to integrate with other tools with ease.

**SIZE**



Consider a situation depicted in the above image.  I have a host system with 16 Gigabytes of RAM and I have to run 3 Virtual Machines on it. To run the Virtual Machines in parallel, I need to divide my RAM among the Virtual Machines. Suppose I allocate it in the following way:

* 6 GB of RAM to my first VM,
* 4 GB of RAM to my second VM, and
* 6 GB to my third VM.

In this case, I will not be left with anymore RAM even though the usage is:

* My first VM uses only **4 GB** of RAM – Allotted **6 GB** – **2 GB** Unused & Blocked
* My second VM uses only **3 GB** of RAM – Allotted **4 GB** – **1 GB** Unused & Blocked
* My third VM uses only **2 GB** of RAM – Allotted **6 GB** – **4 GB** Unused & Blocked

This is because once a chunk of memory is allocated to a Virtual Machine, then that memory is blocked and cannot be re-allocated. I will be wasting **7 GB** (**2 GB + 1 GB + 4 GB**) of RAM in total and thus cannot setup a new Virtual Machine. This is a major issue because RAM is a costly hardware.

***So, how can I avoid this problem?***

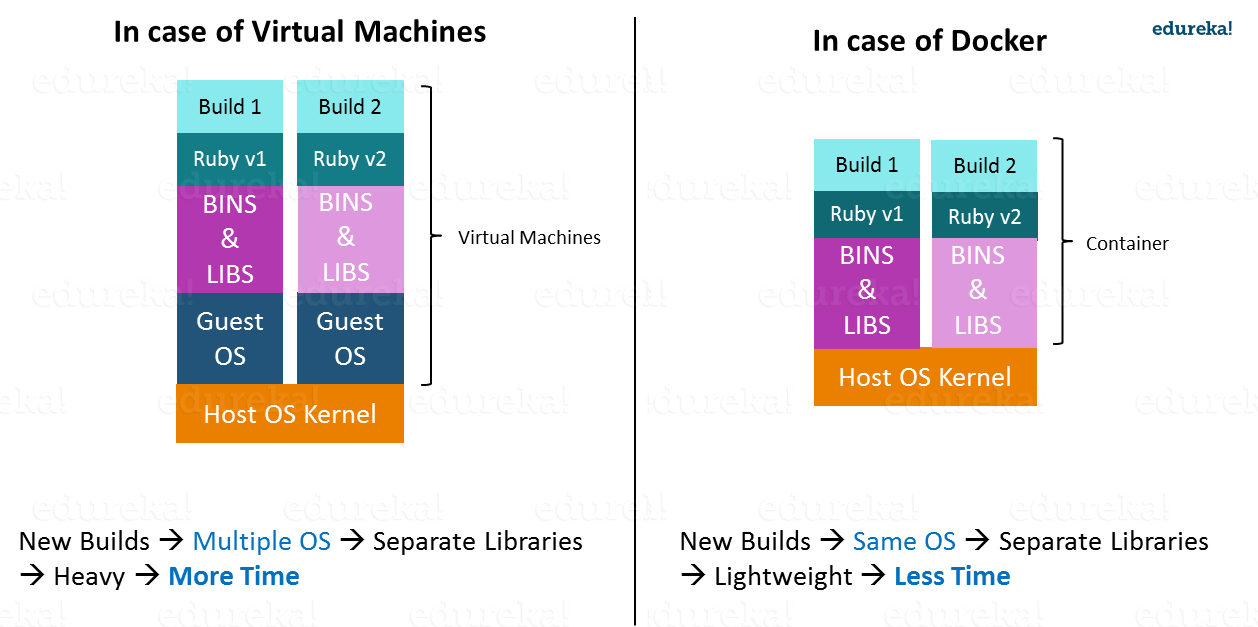
If I use Docker, my CPU will allocates exactly the amount of memory that is required by the Container.

* My first container will use only **4 GB** of RAM – Allotted **4 GB** – **0 GB** Unused & Blocked
* My second container will use only **3 GB** of of RAM – Allotted **3 GB** – **0 GB** Unused & Blocked
* My third container will use only **2 GB** of RAM – Allotted **2 GB** – **0 GB** Unused & Blocked

Since there is no allocated memory (RAM) which is unused, I save **7 GB** (**16** –**4 – 3 – 2**) of RAM by using Docker Container. I can even create additional containers from the leftover RAM and increase my productivity.

So here Docker Container clearly wins over Virtual machine as I can efficiently use my resources as per my need.

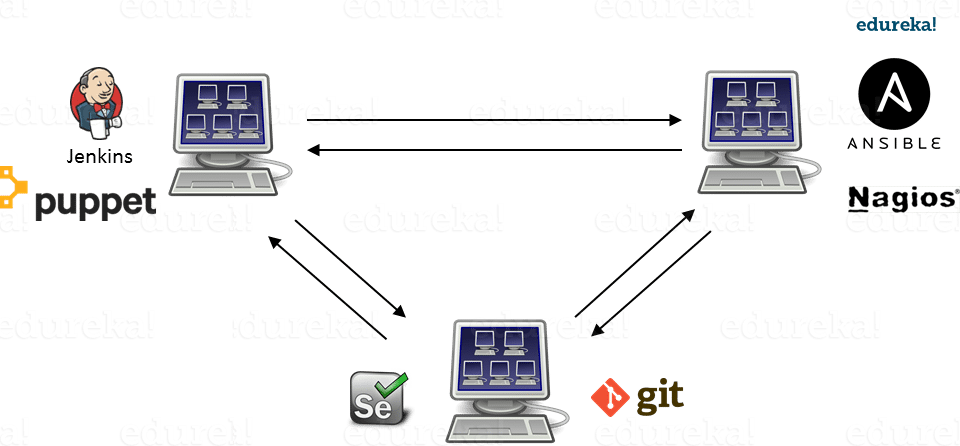
**START-UP**



When it comes to start-up, Virtual Machine takes a lot of time to boot up because the guest operating system needs to start from scratch, which will then load all the binaries and libraries. This is time consuming and will prove very costly at times when quick startup of applications is needed. In case of Docker Container, since the container runs on your host OS, you can save precious boot-up time. This is a clear advantage over Virtual Machine.

Consider a situation where I want to install two different versions of Ruby on my system. If I use Virtual Machine, I will need to set up 2 different Virtual Machines to run the different versions. Each of these will have its own set of binaries and libraries while running on different guest operating systems. Whereas if I use Docker Container, even though I will be creating 2 different containers where each container will have its own set of binaries and libraries, I will be running them on my host operating system. Running them straight on my Host operating system makes my Docker Containers lightweight and faster.

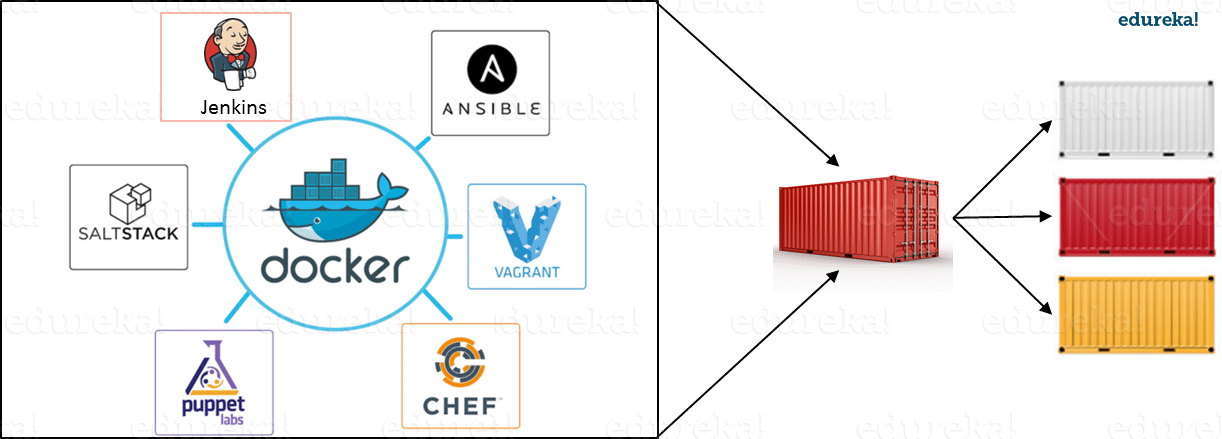
**What About Integration ?**



Integration of different tools using Virtual Machine maybe possible, but even that possibility comes with a lot of complications.

I can have only a limited number of DevOps tools running in a Virtual Machine. As you can see in the image above, If I want many instances of Jenkins and Puppet, then I would need to spin up many Virtual Machines because each can have only one running instance of these tools. Setting up each VM brings with it, infrastructure problems. I will have the same problem if I decide to setup multiple instances of Ansible, Nagios, Selenium and Git. It will also be a hectic task to configure these tools in every VM.

This is where Docker comes to the rescue. Using Docker Container, we can set up many instances of Jenkins, Puppet, and many more, all running in the same container or running in different containers which can interact with one another by just running a few commands. I can also easily scale up by creating multiple copies of these containers. So configuring them will not be a problem.



**Which is a better choice?**

It won’t be fair to compare Docker and virtual machines since they are intended for different use. Docker, no doubt is gaining momentum these days, but they cannot be said to replace virtual machines. In spite of Docker gaining popularity, a virtual machine is a better choice in certain cases. Virtual machines are considered a suitable choice in a production environment, rather than Docker containers since they run on their own OS without being a threat to the host computer. But if the applications are to be tested then Docker is the choice to go for, as Docker provides different OS platforms for the thorough testing of the software or an application.

Furthermore, Docker containers use docker-engine instead of the hypervisor, like in virtual machines. As the host kernel in not shared, using docker-engine makes containers small, isolated, compatible, high performance-intensive and quickly responsive. Docker containers have comparatively low overhead as they have compatibility to share single kernel and application libraries. Organizations are making use of the hybrid approach mostly as the choice between virtual machines and Docker containers depends upon the kind of workload offered.

Also, not many digital operational companies rely on virtual machines as their primary choice and prefer migrating towards using containers as the deployment is comparatively lengthy and running microservices is also one of the major challenges it possesses. However, they are still some firms that prefer virtual machines over Dockers whereas companies who are interested in enterprise-grade security for their infrastructure prefer to make use of Dockers.

Finally, containers and Docker are not in conflict with virtual machines, they are both complementary tools for different workload and usage. Virtual machines are built for applications that are usually static and don’t change very often. Whereas, the Docker platform is built with a mindset to be more flexible so that containers can be updated easily and frequently.

## **Docker Installation**

**Step 1 : UNINSTALL PODMAN & BUIDAH**

$ sudo yum remove podman

$ sudo yum remove buidah

**Step 2 : DOCKER UNINSTALL**

$ sudo yum remove docker-ce docker-ce-cli containerd.io

$ sudo rm -rf /var/lib/docker

$ sudo rm -rf /var/lib/containerd

**Step 3 : YUM UPDATE**

$ yum update all

**Step 4 : DOCKER INSTALLTION USING REPOSITORY**

/var/lib/docker/

Set up the Repo

$ sudo yum install -y yum-utils

$ sudo yum-config-manager \

--add-repo \

https://download.docker.com/linux/centos/docker-ce.repo

**Step 5 : VERIFICATION**

$ docker --verison

**Step 6 : SERVICE ENABLE/RESTART**

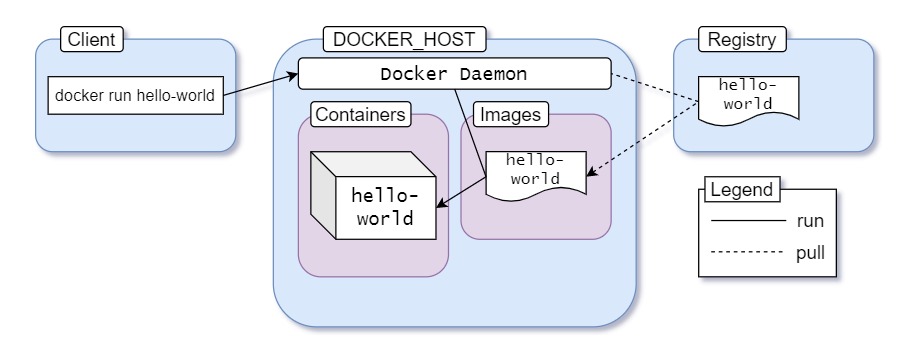
$ sudo systemctl enable docker

$ sudo systemctl start docker

$ sudo systemctl status docker



## **Hello World**



1. You execute docker run hello-world command where hello-world is the name of an image.
2. Docker client reaches out to the daemon, tells it to get the hello-world image and run a container from that.
3. Docker daemon looks for the image within your local repository and realizes that it's not there, resulting in the Unable to find image 'hello-world:latest' locally that's printed on your terminal.
4. The daemon then reaches out to the default public registry which is Docker Hub and pulls in the latest copy of the hello-world image, indicated by the latest: Pulling from library/hello-world line in your terminal.
5. Docker daemon then creates a new container from the freshly pulled image.
6. Finally Docker daemon runs the container created using the hello-world image outputting the wall of text on your terminal.

It's the default behavior of Docker daemon to look for images in the hub that are not present locally. But once an image has been fetched, it'll stay in the local cache. So if you execute the command again, you won't see the following lines in the output:

*Unable to find image 'hello-world:latest' locally*

*latest: Pulling from library/hello-world*

*0e03bdcc26d7: Pull complete*

*Digest: sha256:d58e752213a51785838f9eed2b7a498ffa1cb3aa7f946dda11af39286c3db9a9*

*Status: Downloaded newer image for hello-world:latest*

If there is a newer version of the image available on the public registry, the daemon will fetch the image again. That :latest is a tag. Images usually have meaningful tags to indicate versions or builds