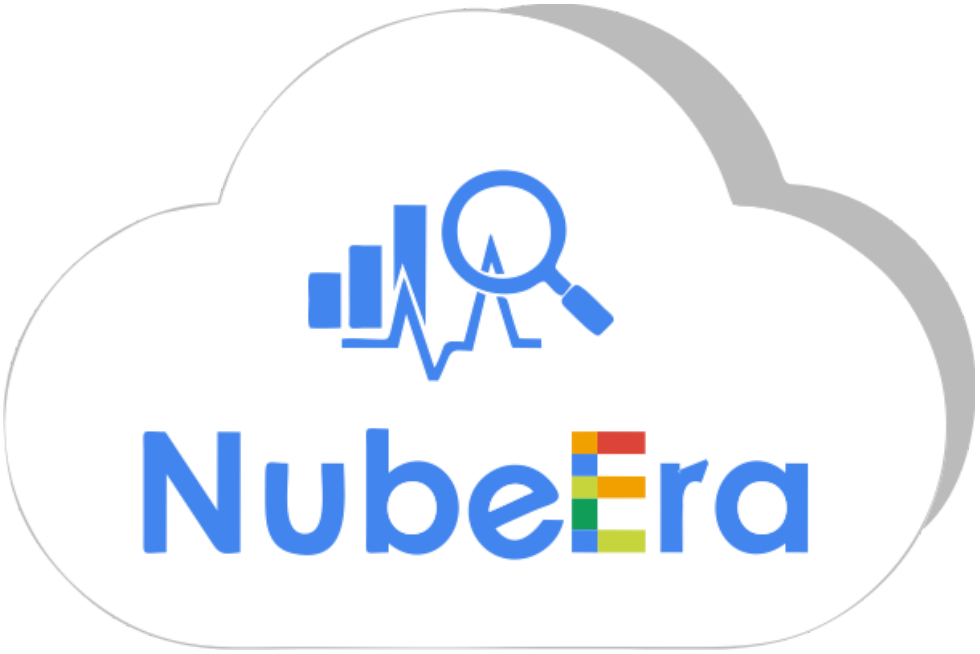
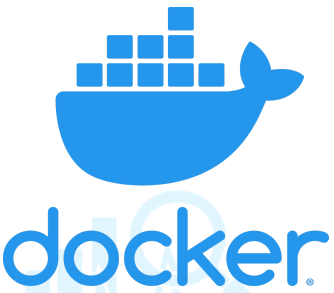




Subject: Introduction to Docker

Trainer:

Document No.

1.1 What is Docker?

In 2019, “DOCKER” refers to several things. This includes an open source community project which started in 2013; tools from the open source project; Docker Inc., the company that is the primary supporter of that project; and the tools that the company formally supports.

Here’s a quick explanation:

- The IT software “Docker” is containerization technology that enables the creation and use of Linux® containers.
- The open source Docker community works to improve these technologies to benefit all users—freely.
- The company, Docker Inc., builds on the work of the Docker community, makes it more secure, and shares those advancements back to the greater community. It then supports the improved and hardened technologies for enterprise customers.

With DOCKER, you can treat containers like extremely lightweight, modular virtual machines. And you get flexibility with those containers—you can create, deploy, copy, and move them from environment to environment, which helps optimise your apps for the cloud.

In nutshell, Docker is a computer program, command-line tool, a containerization platform, product & a company.

Docker as a Computer program…

Docker is a computer program that performs operating-system-level virtualization, also known as “containerization”. It was first released in 2013 and is developed by Docker, Inc.

Docker as a CLI tool

Docker is a command-line program, a background daemon, and a set of remote services that take a logistical approach to solving common software problems and simplifying your experience installing, running, publishing, and removing software. It accomplishes this using a UNIX technology called containers.

Docker as a Platform

Docker is a containerization platform that packages your application and all its dependencies together in the form of a docker container to ensure that your application works seamlessly in any environment.Docker Container is a standardised unit which can be created on the fly to deploy a particular application or environment. It could be an Ubuntu container, CentOs container, etc. to full-fill the requirement from an operating system point of view. Also, it could be an application oriented container like CakePHP container or a Tomcat-Ubuntu container etc.

Docker as a Product

Docker is the leader in the containerization market, combining an enterprise-grade container platform with world-class services to give developers and IT alike the freedom to build, manage and secure applications without the fear of technology or infrastructure lock-in.

How does Docker work?

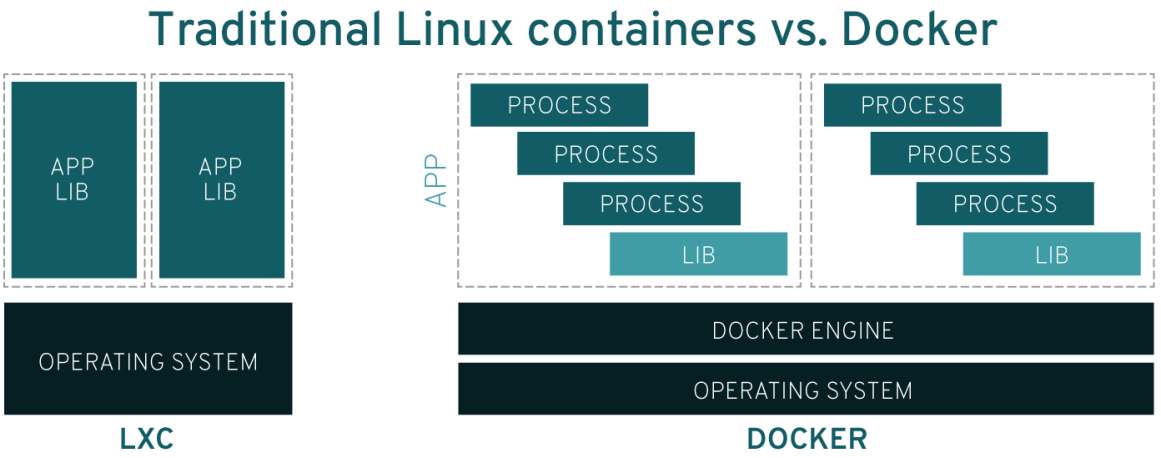
Docker technology uses the Linux kernel and features of the kernel, like Cgroups and namespaces, to segregate processes so they can run independently. This independence is the intention of containers - the ability to run multiple processes and apps separately from one another to make better use of your infrastructure while retaining the security you would have with separate systems.

Container tools, including Docker, provide an image-based deployment model. This makes it easy to share an application, or set of services, with all of their dependencies across multiple environments. Docker also automates deploying the application (or combined sets of processes that make up an app) inside this container environment.

These tools built on top of Linux containers—what makes Docker user-friendly and unique—gives users unprecedented access to apps, the ability to rapidly deploy, and control over versions and version distribution.

1.2 Is Docker technology the same as traditional Linux containers?

No. Docker technology was initially built on top of the LXC technology—what most people associate with “traditional” Linux containers—though it’s since moved away from that dependency. LXC was useful as a lightweight virtualization, but it didn’t have a great developer or user experience. Docker technology brings more than the ability to run containers—it also eases the process of creating and building containers, shipping images, and versioning of images (among other things).



Traditional Linux containers use an init system that can manage multiple processes. This means entire applications can run as one. The Docker technology encourages applications to be broken down into their separate processes and provides the tools to do that. This granular approach has its advantages.

Benefits of Docker containers

Modularity

The Docker approach to containerization is focused on the ability to take down a part of an application, to update or repair, without unnecessarily taking down the whole app. In addition to this microservices-based approach, you can share processes amongst multiple apps in much the same way that service-oriented architecture (SOA) works.

Layers and image version control

Each Docker image file is made up of a series of layers. These layers are combined into a single image. A layer is created when the image changes. Every time a user specifies a command, such as run or copy, a new layer gets created.

Docker reuses these layers for new container builds, which makes the build process much faster. Intermediate changes are shared between images, further improving speed, size, and efficiency. Inherent to layering is version control. Every time there’s a new change, you essentially have a built-in changelog—full control over your container images.

Rollback

Perhaps the best part about layering is the ability to roll back. Every image has layers. Don't like the current iteration of an image? Roll it back to the previous version. This supports an agile development approach and helps make continuous integration and deployment (CI/CD) a reality from a tools perspective.

Rapid deployment

Getting new hardware up, running, provisioned, and available used to take days. And the level of effort and overhead was burdensome. Docker-based containers can reduce deployment to seconds. By creating a container for each process, you can quickly share those similar processes with new apps. And, since an OS doesn't need to boot to add or move a container, deployment times are substantially shorter. On top of this, with the speed of deployment, you can easily and cost-effectively create and destroy data created by your containers without concern.

So, Docker technology is a more granular, controllable, microservices-based approach that places greater value on efficiency.

1.3 What are Containers?

Container comprises several building blocks, the two most important being namespaces and cgroups (control groups). Both of them are Linux kernel features.

Namespaces provide logical partitions of certain kinds of system resources, such as mounting point (mnt), process ID (PID), network (net), and so on. To explain the concept of isolation, let's look at some simple examples on the pid namespace.

The following examples are all from Ubuntu 16.04.2 and util-linux 2.27.1. When we type ps axf, we will see a long list of running processes:

```
$ ps axf
PID TTY STAT TIME COMMAND
2 ? S 0:00 [kthreadd]
3 ? S 0:42 \_ [ksoftirqd/0]
5 ? S< 0:00 \_ [kworker/0:0H]
7 ? S 8:14 \_ [rcu_sched]
8 ? S 0:00 \_ [rcu_bh]
```

ps is a utility to report current processes on the system. ps axf is to list all processes in forest.

Now let's enter a new pid namespace with unshare, which is able to disassociate a process resource part-by-part to a new namespace, and check the processes again:

```
$ sudo unshare --fork --pid --mount-proc=/proc /bin/sh
$ ps axf
PID TTY STAT TIME COMMAND
1 pts/0 S 0:00 /bin/sh
2 pts/0 R+ 0:00 ps axf
```

You will find the pid of the shell process at the new namespace becoming 1, with all other processes disappearing. That is to say, you have created a pid container.

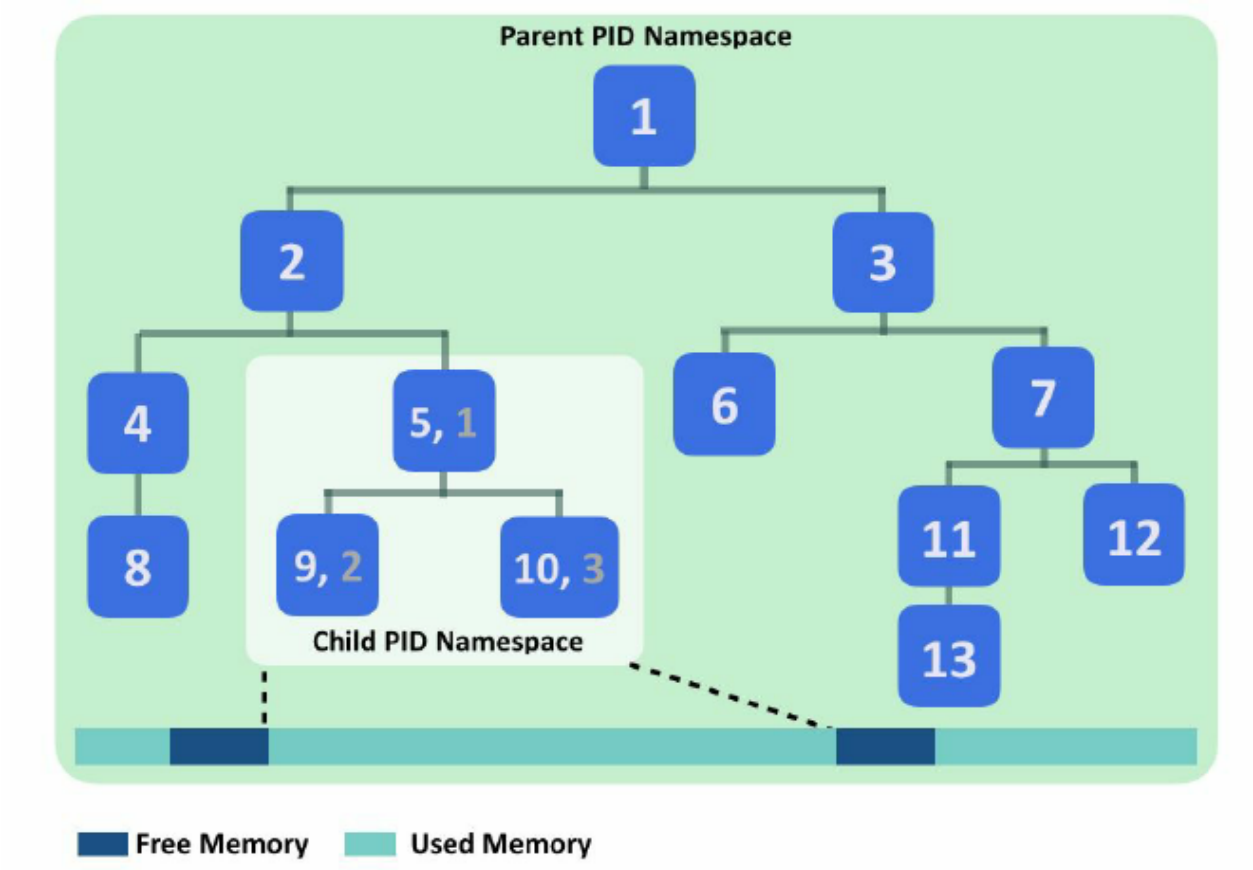
Let's switch to another session outside the namespace, and list the processes again:

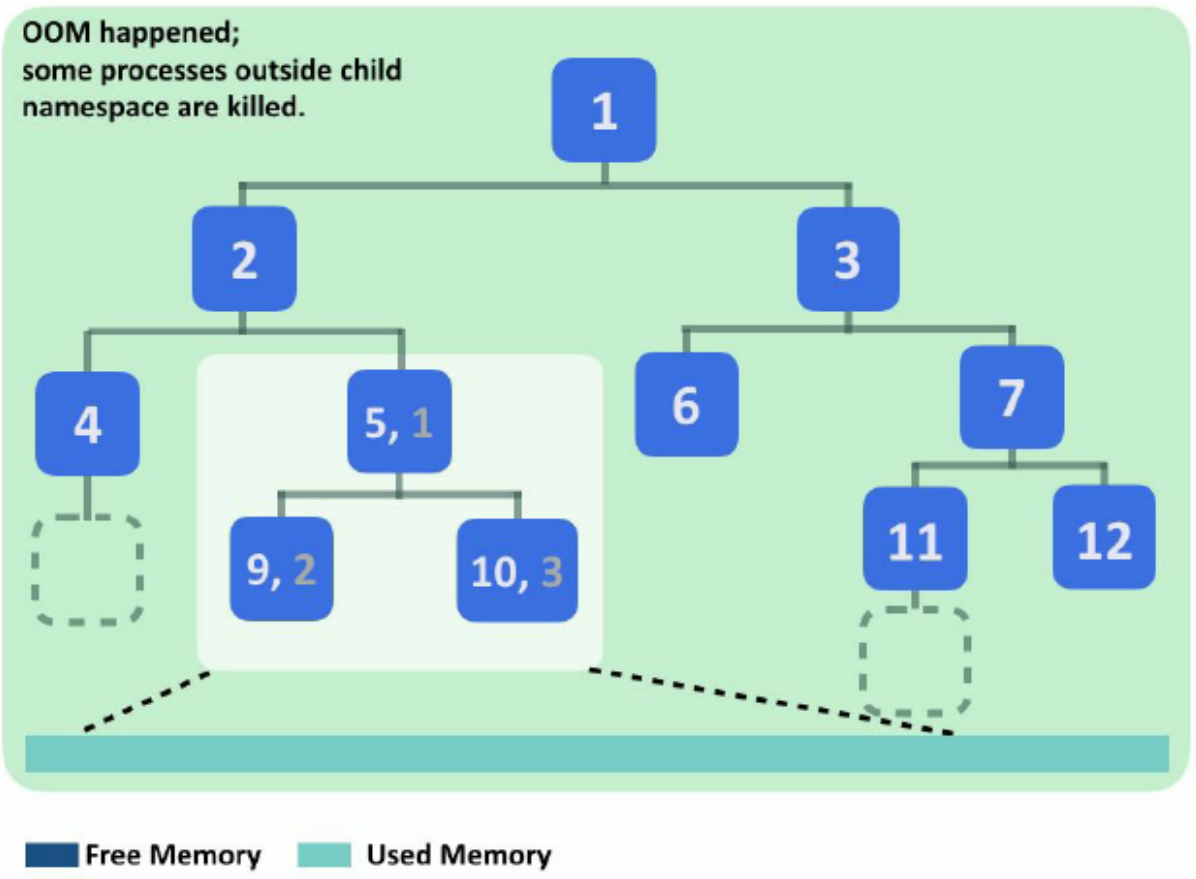
```
$ ps axf // from another terminal
PID TTY COMMAND
...
25744 pts/0 \_ unshare --fork --pid --mount-proc=/proc
/bin/sh
25745 pts/0 \_ /bin/sh
3305 ? /sbin/rpcbind -f -w
6894 ? /usr/sbin/ntpd -p /var/run/ntpd.pid -g -u
113:116
```

You can still see the other processes and your shell process within the new namespace.

With the pid namespace isolation, processes in different namespaces cannot see each other. Nonetheless, if one process eats up a considerable amount of system resources, such as memory, it could cause the system to run out of memory and become unstable. In other words, an isolated process could still disrupt other processes or even crash a whole system if we don't impose resource usage restrictions on it.

The following diagram illustrates the PID namespaces and how an out-of-memory (OOM) event can affect other processes outside a child namespace. The bubbles are the process in the system, and the numbers are their PID. Processes in the child namespace have their own PID. Initially, there is still free memory available in the system. Later, the processes in the child namespace exhaust the whole memory in the system. The kernel then starts the OOM killer to release memory, and the victims may be processes outside the child namespace





In light of this, cgroups are utilized here to limit resource usage. Like namespaces, it can set constraints on different kinds of system resources. Let's continue from our pid namespace, stress the CPU with yes > /dev/null, and monitor it with top:

```
$ yes > /dev/null & top
$ PID USER PR NI VIRT RES SHR S %CPU %MEM
TIME+ COMMAND
3 root 20 0 6012 656 584 R 100.0 0.0
0:15.15 yes
1 root 20 0 4508 708 632 S 0.0 0.0
0:00.00 sh
4 root 20 0 40388 3664 3204 R 0.0 0.1
0:00.00 top
```

Our CPU load reaches 100% as expected. Now let's limit it with the CPU cgroup. Cgroups are organised as directories under /sys/fs/cgroup/ (switch to the host session first):

```
$ ls /sys/fs/cgroup
blkio cpuset memory perf_event
cpu devices net_cls pids
cpuacct freezer net_cls,net_prio systemd
cpu,cpuacct hugetlb net_prio
```

Each of the directories represents the resources they control. It's pretty easy to create a cgroup and control processes with it: just create a directory under the resource type with any name, and append the process IDs you'd like to control to tasks. Here we want to throttle the CPU usage of our yes process, so create a new directory under cpu and find out the PID of the yes process:

```
$ ps x | grep yes
11809 pts/2 R 12:37 yes
$ mkdir /sys/fs/cgroup/cpu/box && \
echo 11809 > /sys/fs/cgroup/cpu/box/tasks
```

We've just added yes into the newly created CPU group box, but the policy remains unset, and processes still run without restriction. Set a limit by writing the desired number into the corresponding file and check the CPU usage again:

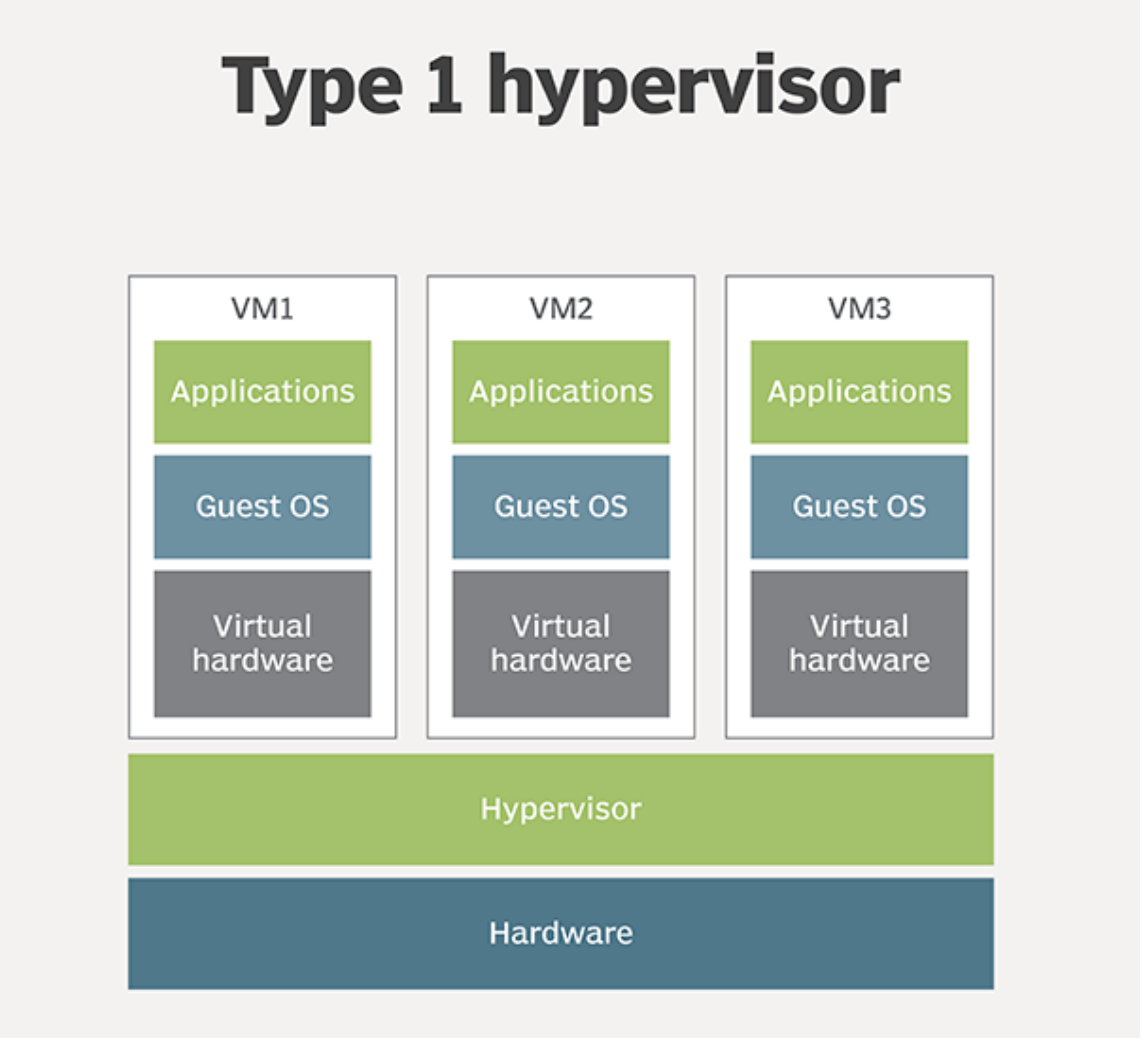
```
$ echo 50000 > /sys/fs/cgroup/cpu/box/cpu.cfs_quota_us
$ PID USER PR NI VIRT RES SHR S %CPU %MEM
TIME+ COMMAND
3 root 20 0 6012 656 584 R 50.2 0.0
0:32.05 yes
1 root 20 0 4508 1700 1608 S 0.0 0.0
0:00.00 sh
4 root 20 0 40388 3664 3204 R 0.0 0.1
0:00.00 top
```

The CPU usage is dramatically reduced, meaning that our CPU throttle works. These two examples elucidate how Linux container isolate system resources. By putting more confinements in an application, we can definitely build a fully isolated box, including file system and networks, without encapsulating an operating system within it.

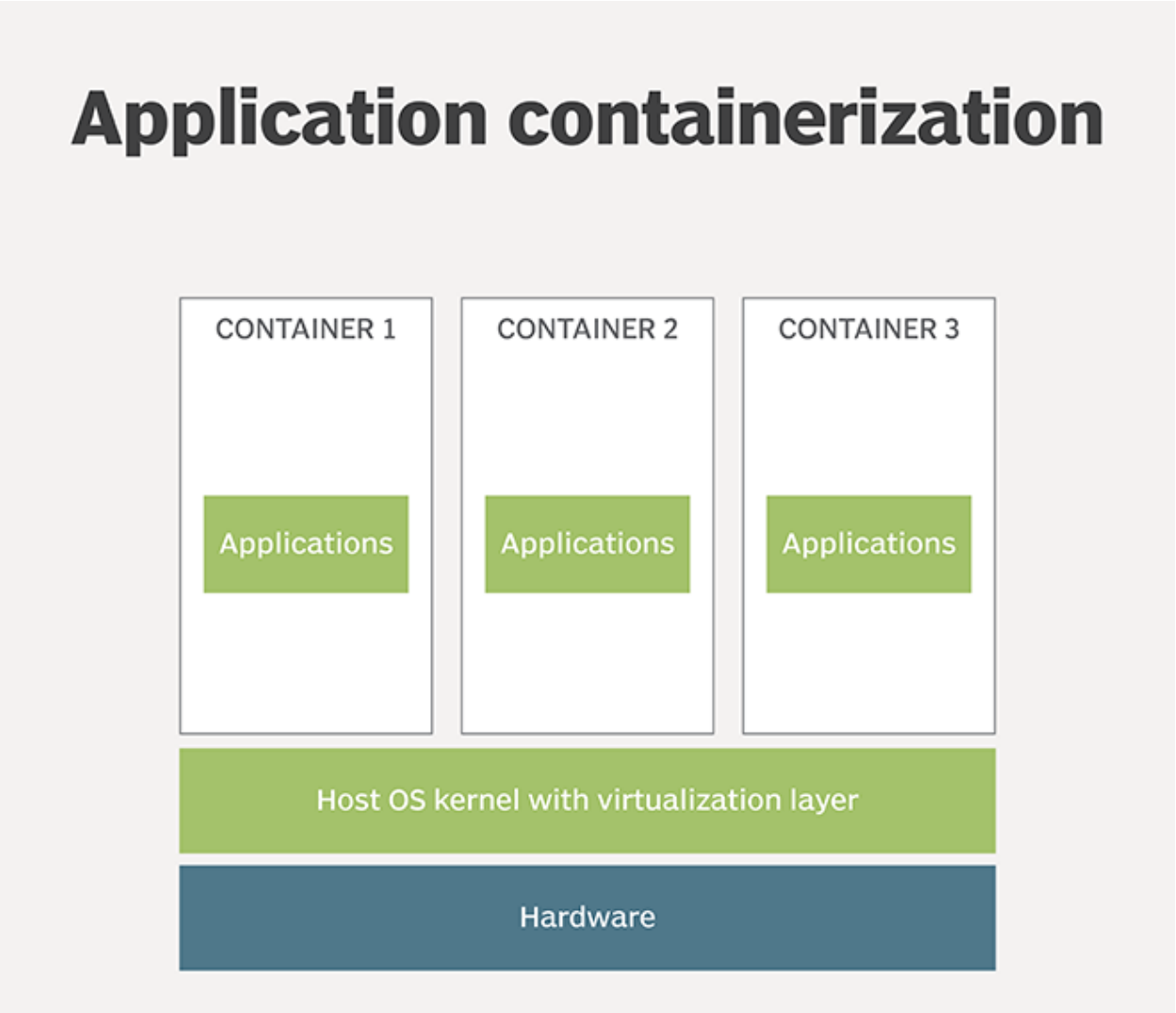
A basic physical application installation needs server, storage, network equipment and other physical hardware on which an OS is installed. A software stack – an application server, a database and more – enables the application to run. An organisation must either provision resources for its maximum workload and potential outages and suffer significant waste outside those times or, if provisioned resources are set for average workload, expect traffic peaks to lead to performance issues.

VMs get around some of these problems. A VM creates a logical system that sits atop the physical platform (see Figure 1). A Type 1 hypervisor, such as VMware ESXi or Microsoft Hyper-V, provides each VM with virtual hardware. The VM runs a guest OS, and the application software stack interprets everything below it the same as a physical stack. Virtualization utilizes resources better

than physical setups, but the separate OS for each VM creates significant redundancy in base functionality.



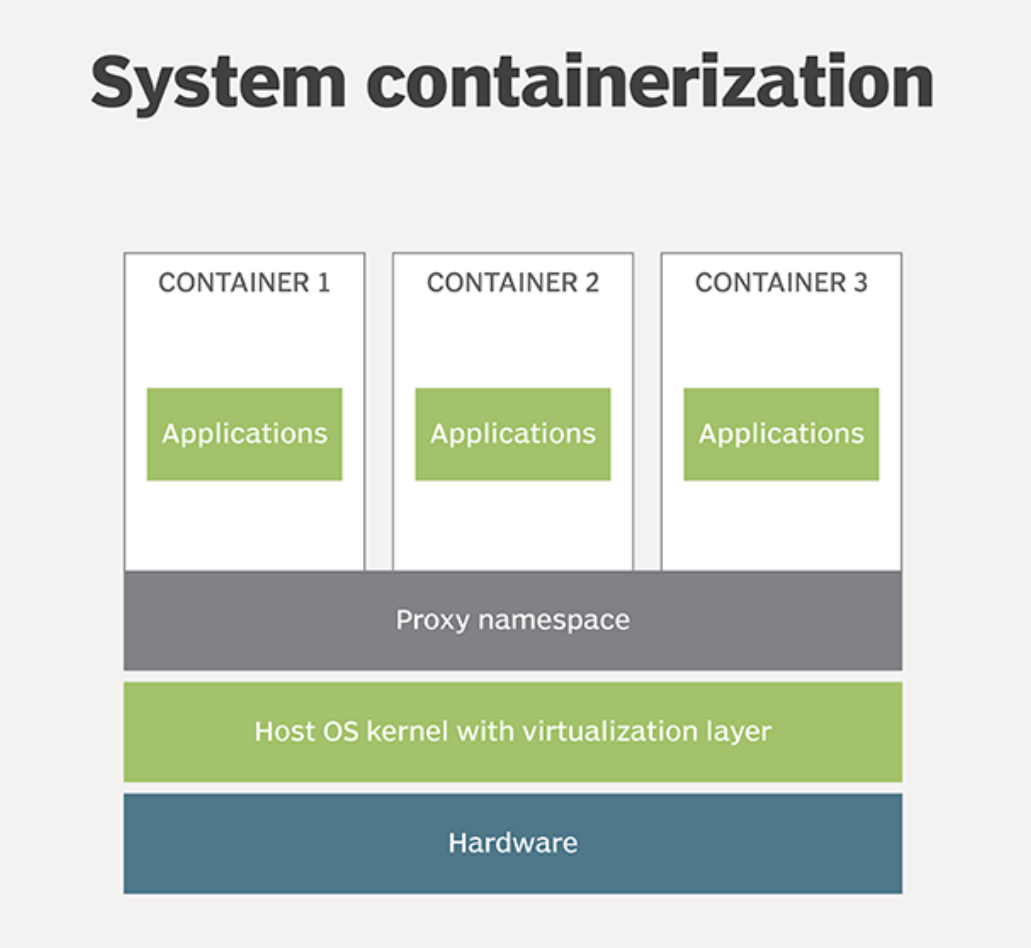
Containers provide greater flexibility than virtual and physical hardware stacks. A basic application container environment, as seen in Figure 2, runs on physical – or virtual and physical – hardware, a host OS and a container virtualization layer directly on the OS. Containers share the OS and its functions instead of running individual OS instances. This greatly reduces the resources required per application. Docker, Rkt (a CoreOS container runtime acquired by Red Hat), Linux Containers and Windows Server Containers operate generally in a similar manner.



The benefits have downsides

OS sharing led to problems in early containers. Code that required raised privilege at the OS level opened businesses to the potential of malicious entities able to gain access to, and attack, the underlying platform to bring down all the containers in the environment. Some organizations ran containers within VMs to combat the issue, but others argued that doing so destroyed the point of containers. Modern container environments mitigate these security issues, but many organizations still host containers in VMs for security or management reasons.

The fact that all container applications must use the same underlying OS is a strength, as well as a weakness, of containerized applications. Every application container sharing a Linux OS, for example, must not only be based on Linux, but also on the same version and often patch level of that Linux distribution. That isn't always manageable in reality, as some applications have specific OS requirements.



System containerization, demonstrated in Figure 3, resolves this tangle. System containers use the shared capabilities of the underlying OS. Where the application needs a certain patch level or functional library that the underlying platform lacks, a proxy namespace captures the call from the application and redirects it to the necessary code or library held within the container itself.

System containerization is available from Virtuozzo. Microsoft also offers a similar approach to isolation via its Hyper-V containers.

What are containers enabling?

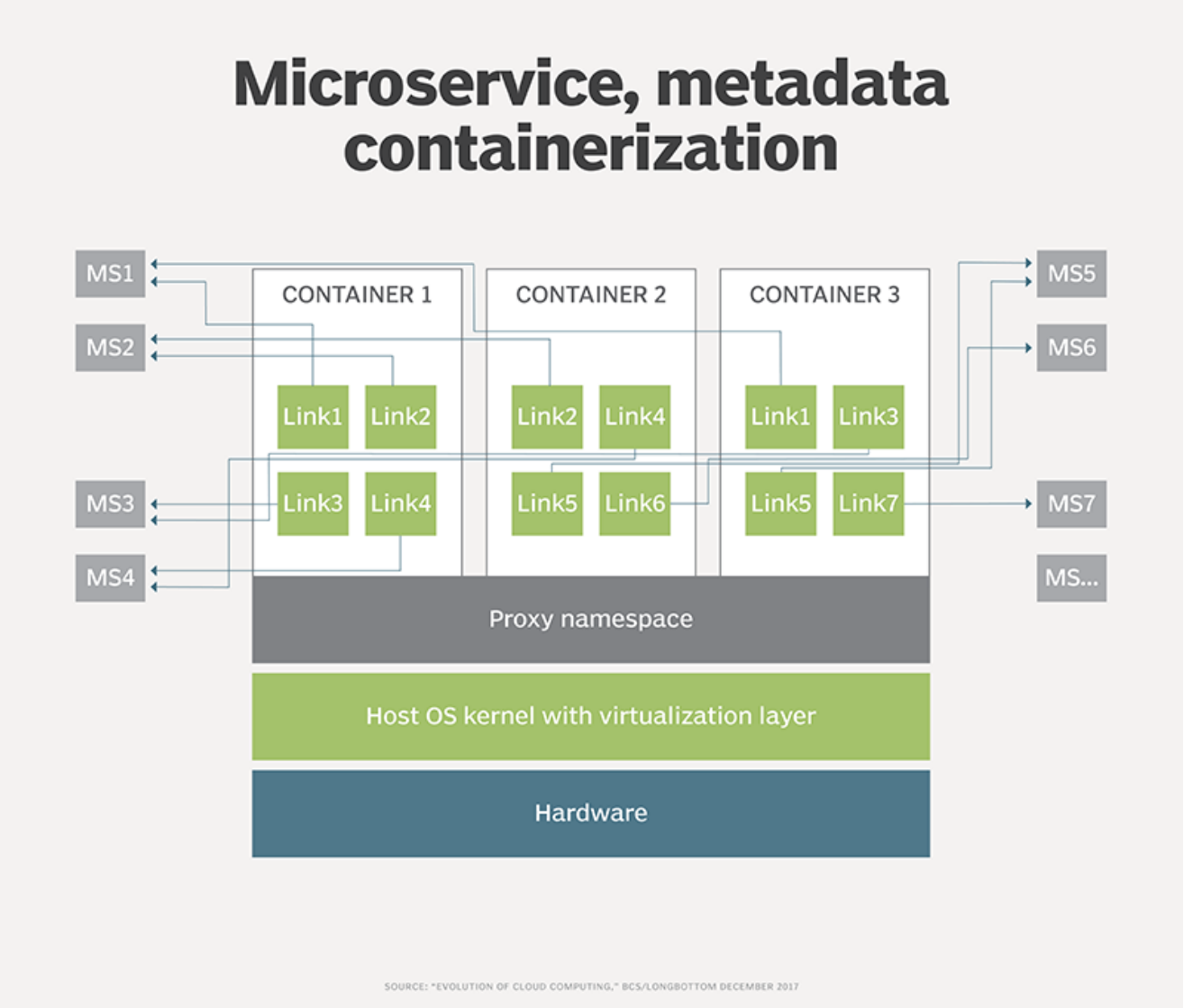
Applications have evolved through physical hardware to VMs to containerized environments. In turn, containerization is ushering in microservices architecture.

Microservices create single-function entities that offer services to a calling environment. For example, functions such as calendars, email and financial clearing systems can live in individual containers available in the cloud for any system that needs them, in contrast to a collection of disparate systems that each contain internal versions of these capabilities.

Performance benefits from hosting such functions in the cloud. Sharing the underlying physical resources elastically with other functions minimizes the likelihood of hitting resource limits.

Microservices also offer flexible, process-based methods to handle business needs in an application architecture. Rather than code that tries to guess at the business process and ends

up constraining it, microservices create a composite application of dynamic functions pulled together in real time that enables a business to respond to market forces more rapidly than monolithic applications can.



As shown in Figure 4, the container doesn't carry around physical code within it, but rather a list of required functions that it pulls together as required. The container manages areas such as technical contracts and process audits.

IT professionals can soon expect to see how containers will evolve from here. Containers exist in a highly dynamic and changing world. They optimize resource utilization and provide much needed flexibility better than their predecessors, so enterprise IT organizations should prepare to use them.

1.4 Difference between VM vs Docker

Let us understand this with a simple analogy.

Virtual machines have a full OS with its own memory management installed with the associated overhead of virtual device drivers. In a virtual machine, valuable resources are emulated for the guest OS and hypervisor, which makes it possible to run many instances of one or more operating systems in parallel on a single machine (or host). Every guest OS runs as an individual entity from the host system. Hence, we can look at it an independent full-fledge house where we don't share any resources as shown below:

In the other hand, Docker containers are executed with the Docker engine rather than the hypervisor. Containers are therefore smaller than Virtual Machines and enable faster start up with better performance, less isolation and greater compatibility possible due to sharing of the host's kernel. Hence, it looks very similar to residental flats system where we share resources of the building.

Docker containers are NOT VMs

- It's not quite like a VM
- Uses the host kernel
- Can't boot a different OS
- Can't have its own modules
- Doesn't need init as PID 1
- Doesn't need syslogd, cron.

It's just a normal process on the host machine

- Contrast with VMs which are opaque

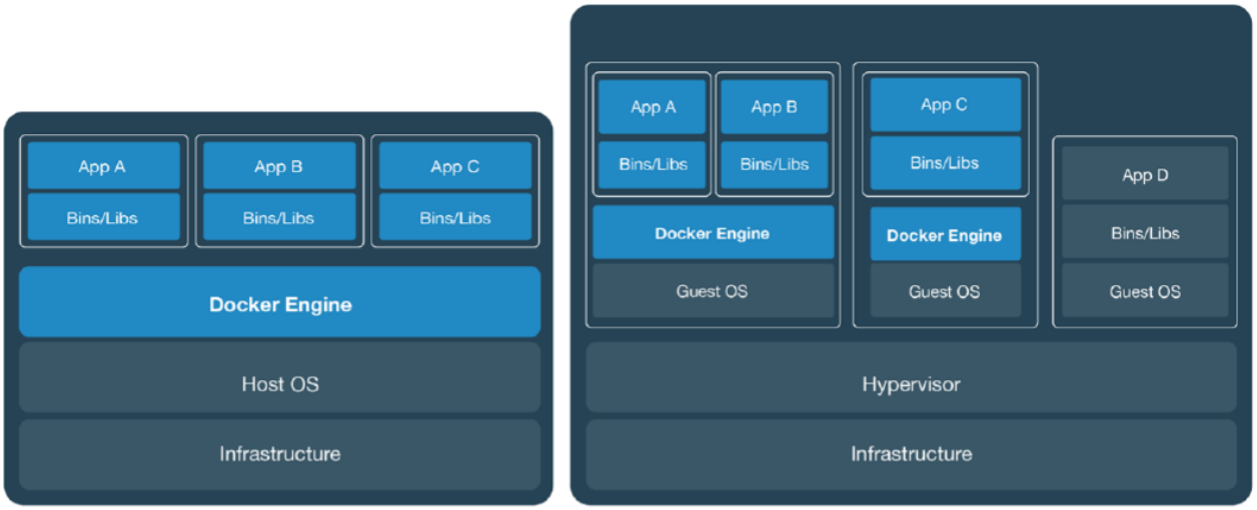
Docker Containers versus Virtual Machines:

Virtual Machines	Docker
Each VM runs its own OS	All containers share the same Kernel of the host
Boot up time is in minutes	Containers instantiate in seconds
VMs snapshots are used sparingly	Images are built incrementally on top of another like layers. Lots of images/snapshots
Not effective diffs. Not version controlled	Images can be diffed and can be version controlled. Dockerhub is like GITHUB
Cannot run more than couple of VMs on an average laptop	Can run many Docker containers in a laptop.
Only one VM can be started from one set of VMX and VMDK files	Multiple Docker containers can be started from one Docker image

When it comes to comparing the two, it could be said that Docker Containers have much more potential than Virtual Machines. It's evident as Docker Containers are able to share a single kernel and share application libraries. Containers present a lower system overhead than Virtual Machines and performance of the application inside a container is generally same or better as compared to the same application running within a Virtual Machine.

There is one key metric where Docker Containers are weaker than Virtual Machines, and that's "Isolation". Intel's VT-d and VT- x technologies have provided Virtual Machines with ring-1 hardware isolation of which, it takes full advantage. It helps Virtual Machines from breaking down and interfering with each other. Docker Containers yet don't have any hardware isolation, thus making them receptive to exploits.

Containers Vs VMs



As compared to virtual machines, containers can be faster and less resource heavy as long as the user is willing to stick to a single platform to provide the shared OS. A virtual machine could take up several minutes to create and launch whereas a container can be created and launched just in a few seconds. Applications contained in containers offer superior performance, compared to running the application within a virtual machine.

There is an estimation being done by Docker that application running in a container can go twice as fast as one in a virtual machine. Also, a single server can pack more than one containers as OS is not duplicated for each application.

Virtual Machines and Containers: better together

You can sometimes use a hybrid approach which uses both VM and Docker. There are also workloads which are best suited for physical hardware. If both are placed in a hybrid approach, it might lead to a better and efficient scenario. With this Hybrid setup, users can benefit from the advantages if they have workloads that fit the model.

Following are a few of them, that explain how they work together as a Hybrid:

1). Docker Containers and Virtual Machines by themselves are not sufficient to operate an application in production. So one should be considering how the Docker Containers are going to run in an enterprise data centre.

2). Application probability and enabling the accordant provisioning of the application across infrastructure is provided by containers. But other operational requirements such as security,

performance and capacity management and various management tool integrations are still a challenge in front of Docker Containers, thus leaving everyone in a big puzzle.

3). Security isolation can be equally achieved by both Docker Containers and Virtual Machines.

4). Docker Containers can run inside Virtual Machines though they are positioned as two separate technologies and provide them with pros like proven isolation, security properties, mobility, dynamic virtual networking, software-defined storage and massive ecosystem.

Apples to apples comparison: On a physical host with a certain configuration and Virtual Machines with the same configuration running an identical running same number of docker Containers with the same performance on both?

Okay. Who's the winner?

Answer to this question so far cannot be ascertained but depending upon their configurations and constraints one could say that containers are overcoming virtual machines. Application design is the one standpoint suggesting which one of the two should be chosen. If application is designed to provide scalability and high availability then containers are the best choice else application can be placed in a virtual machine, though Docker containers have surely challenged virtualization market with containers. Well, keeping the debate aside, it is easy to say that containers in Virtual Machines are twice as robust as one without the other.

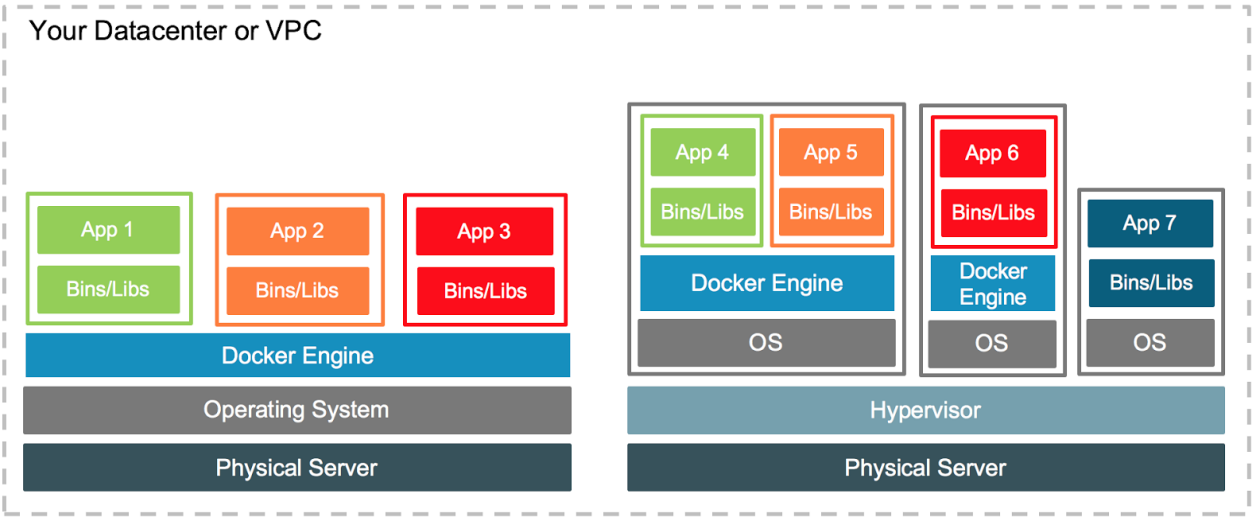
1.5 Similarity between Docker & VM

Docker containers might look very similar to Virtual Machines, especially for those who come from VMware Virtualization Background. Few of the interesting similarities between Docker & Virtual Machines are listed below:

Virtual Machines	Docker
Process in one VM can't see processes in other VMs	Process in one container can't see processes in other container
Each VM has its own root filesystem	Each container has its own root file system(Not Kernel)
Each VM gets its own virtual network adapter	Docker can get virtual network adapter. It can have separate IP and ports
VM is a running instance of physical files(.VMX and .VMDK)	Docker containers are running instances of Docker Image
Host OS can be different from guest OS	Host OS can be different from Container OS

1.6 Difference between Containers and Virtual Machines

Let us look at the component view of how virtual machines and containers are implemented.



Virtual Machine

If we closely look at how Virtual Machines are built over the physical hardware, there is a layer of Hypervisor which sits between physical hardware and operating systems. In a broader view, Hypervisor is used to virtualize the hardware which is then configured with the way a user wants it to.

Virtual Machines our physical machine is divided into following parts :-

Example:

Say, you are using a system having 8GB RAM. If you create 2 VMs each with 4GB RAM, you are basically dividing your server into two components - each with 4GB RAM and would never be able to use that underlying 8GB RAM altogether again.

Virtual Machines	Docker
Each VM runs its own OS	All containers share the same Kernel of the host
Boot up time is in minutes	Containers instantiate in seconds
VMs snapshots are used sparingly	Images are built incrementally on top of another like layers. Lots of images/snapshots
Not effective diffs. Not version controlled	Images can be diffed and can be version controlled. Dockerhub is like GITHUB
Cannot run more than couple of VMs on an average laptop	Can run many Docker containers in a laptop.
Only one VM can be started from one set of VMX and VMDK files	Multiple Docker containers can be started from one Docker image

Containers

Unlike virtual machines where hypervisor divides physical hardware into parts, Containers are like normal operating system processes. Now the question is, if containers are like normal processes then how come there are isolated from other processes. This is where namespaces come into picture.

Namespaces is an advance concept in linux where each namespace has its own isolated resources without actual partitioning of the underlying hardware. Namesapces are used to virtualize the underlying operating system.

Since containers are nothing other than OS processes, This is the reason why lifting a container takes seconds and lifting a virtual machine takes minutes.

Conclusion

Prime differnce between Containers and Virtual Machine is the virtualization of *Operating System* and *Hardware* respectively. While using Virtual machine each virtual machine has its own underlying operating system whereas, While using containers, Each container runs on same underlying operating system instance, While underlying OS is the same containers can still have different OS environment in their respective namespace.

1.7 Top Reasons WHY TO & WHY NOT TO run Docker containers on Bare Metal System

Why to run containers on Bare Metal System?

Containers on bare-metal hosts get many of the advantages VMs offer but without the drawbacks of virtualization:

- Gain access to bare-metal hardware in apps without relying on pass-through techniques, because the app processes run on the same OS as the host server.
- Have optimal use of system resources. Although you can set limits on how much compute, storage and networking containers can use, they generally don't require these resources to be dedicated to a single container. A host can, therefore, distribute use of shared system resources as needed.
- Get bare-metal performance to apps, because there is no hardware emulation layer separating them from a host server.

In addition, by hosting containers on bare metal, you get the benefits that have traditionally been possible only with VMs:

- Gain the ability to deploy apps inside portable environments that can move easily between host servers.
- Get app isolation. Although containers arguably don't provide the same level of isolation as VMs, containers do enable admins to prevent apps from interacting with one another and to set strict limits on the privileges and resource accessibility associated with each container.

In short, run containers on bare metal to square the circle – do what seems impossible. Reap all the benefits of bare-metal servers' performance and hardware accessibility, and take advantage of the portability and isolation features seen with VMs.

WHY NOT TO RUN CONTAINERS ON BARE METAL SYSTEM

You probably are wondering why we don't run all containers on bare metal. Consider the following drawbacks of using bare-metal servers to host a container engine, rather than VMs:

- Physical server upgrades are difficult. To replace a bare-metal server, you must recreate the container environment from scratch on the new server. If the container environment were part of a VM image, you could simply move the image to the new host.
- Most clouds require VMs. There are some bare-metal cloud hosts out there, such as Rackspace's OnMetal offering and Oracle's Bare Metal Cloud Services. But bare-metal servers in cloud computing environments usually cost significantly more – if the cloud vendor even offers it. By and large, most public cloud providers only offer VMs. If you want to use their platforms to run containers, you'll have to deploy into VMs.
- Container platforms don't support all hardware and software configurations. These days, you can host almost any type of OS on a VM platform such as VMware or a kernel-based VM. And you can run that virtualization platform on almost any kind of OS or server.

Docker is more limited and can run only on Linux, certain Windows servers and IBM mainframes if hosted on bare metal. For example, bare-metal servers that run Windows Server 2012 – which Docker does not currently support – require a VM on top of the Windows host.

- Containers are OS-dependent. Linux containers run on Linux hosts; Windows containers run on Windows hosts. A bare-metal Windows server requires a Linux VM environment to host Docker containers for an app compiled for Linux. However, there are technological developments in this space (see sidebar).
- Bare-metal servers don't offer rollback features. Most virtualization platforms enable admins to take VM snapshots and roll back to that captured configuration status at a later time. Containers are ephemeral by nature, so there is nothing to roll back to. You might be able to use rollback features built into the host OS or file system, but those are often a less seamless experience. To take advantage of simple system rollback, host containers on a VM.

Here's a Bonus..

Linux containers on bare-metal Windows

Historically, there was not an easy, efficient way to run Linux containers on a Windows host. However, Linux Containers on Windows (LCOW) is a tool from Docker that automatically runs Linux containers on a Windows host.

Technically, LCOW needs the Windows host to set up a Hyper-V hypervisor, so the Linux containers actually don't run directly on bare metal inside the Windows environment. However, the hypervisor footprint is miniscule, and the hypervisor is fully managed. Consequently, the performance and management drawbacks associated with hosting containers inside a VM don't exist with LCOW, and the technology removes another historical downside of running containers on bare metal.

1.8 Difference between VM Networking Vs Container Networking

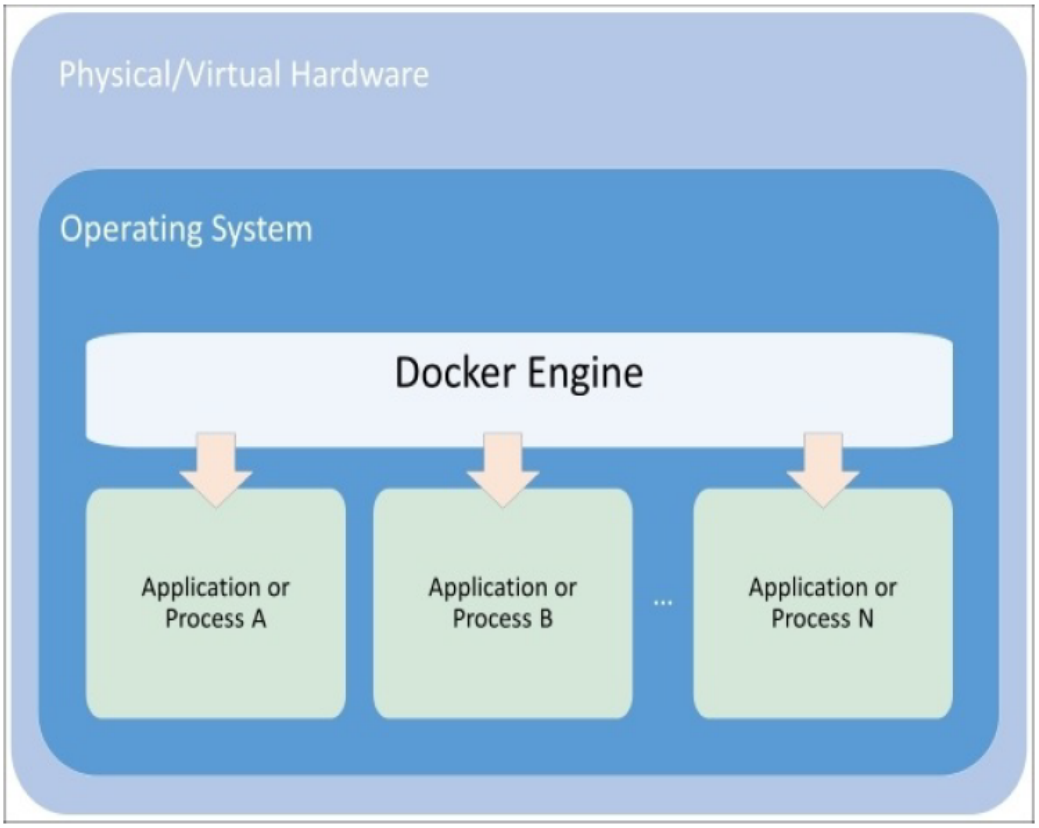
Feature	Container	VM
Isolation	Network isolation achieved using Network namespace.	Separate networking stack per VM
Service	Typically, Services gets separate IP and maps to multiple containers	Multiple services runs in a single VM
Service Discovery and Load balancing	Microservices done as Containers puts more emphasis on integrated Service discovery	Service Discovery and Load balancing typically done outside
Scale	As Container scale on a single host can run to hundreds, host networking has to be very scalable.	Host networking scale needs are not as high
Implementation	Docker Engine and Linux bridge	Hypervisor and Linux/OVS bridge

1.9 Understanding Docker Underlying Technology

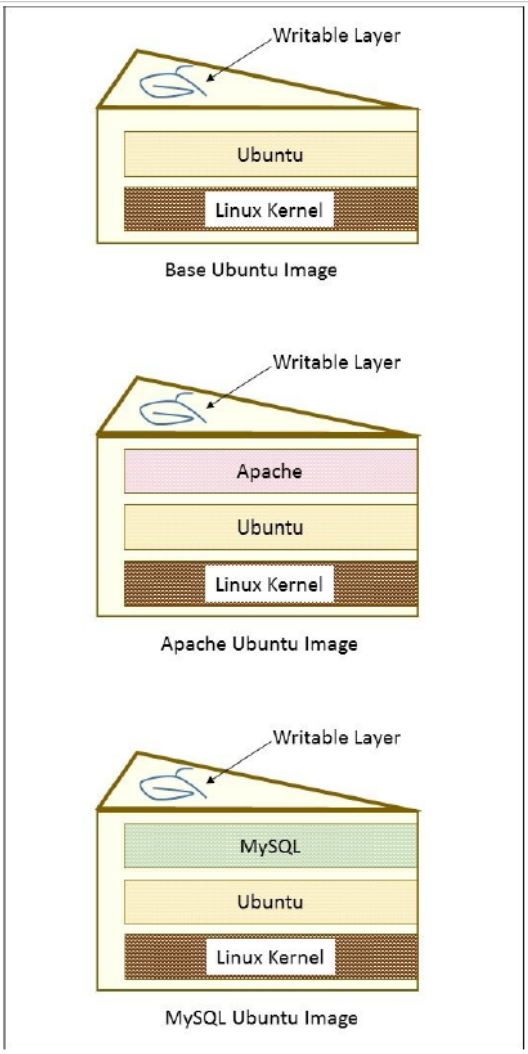
To understand Docker's underlying technology, let us first understand the core container technology.

At the core of container technology are cGroups and namespaces. Additionally, Docker uses union file systems(UFS) for added benefits to the container development process. Control groups (cGroups) work by allowing the host to share and also limit the resources each process or container can consume. This is important for both resource utilisation and security, as it prevents denial-of-service attacks on the host's hardware resources. Several containers can share CPU and memory while staying within the predefined constraints.

Namespaces offer another form of isolation in the way of processes. Processes are limited to see only the process ID in the same namespace. Namespaces from other system processes would not be accessible from a container process. For example, a network namespace would isolate access to the network interfaces and configuration, which allows the separation of network interfaces, routes, and firewall rules.



Union file systems are also a key advantage to using Docker containers. The easiest way to understand union file systems is to think of them like a layer cake with each layer baked independently. The Linux kernel is our base layer; then, we might add an OS like Red Hat Linux or Ubuntu. Next, we might add an application like Nginx or Apache. Every change creates a new layer. Finally, as you make changes and new layers are added, you'll always have a top layer (think frosting) that is a writable layer.



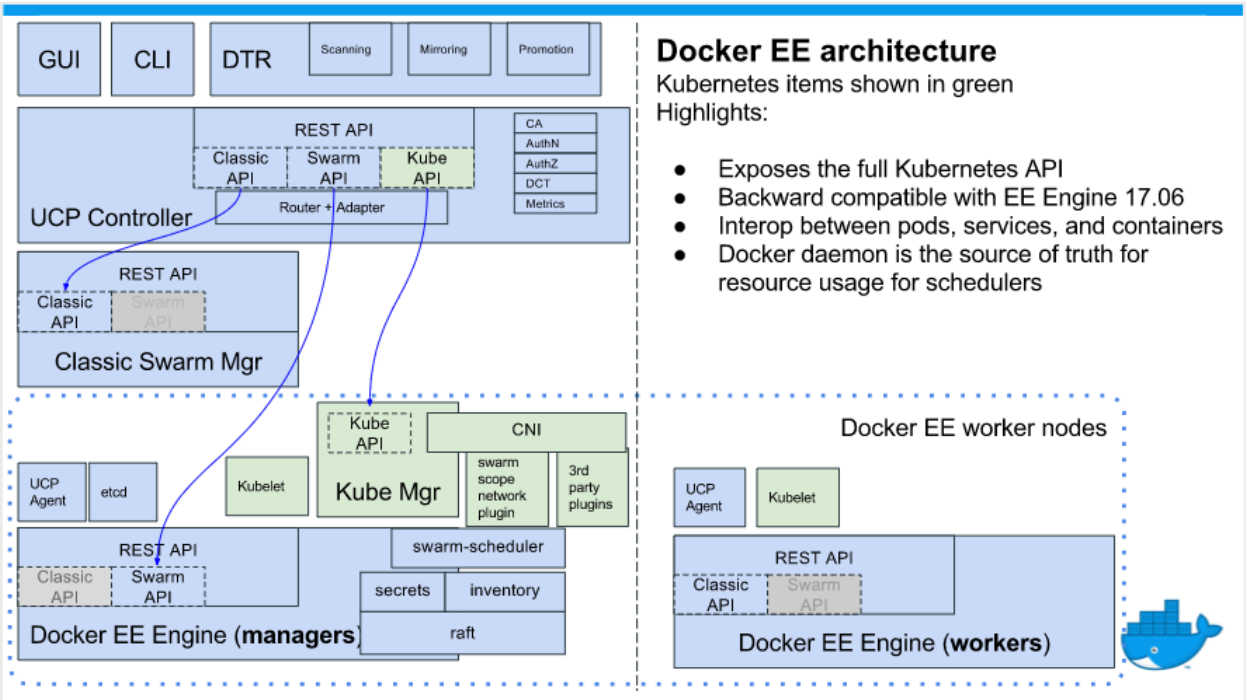
What makes this truly efficient is that Docker caches the layers the first time we build them. So, let's say that we have an image with Ubuntu and then add Apache and build the image. Next, we build MySQL with Ubuntu as the base. The second build will be much faster because the Ubuntu layer is already cached. Essentially, our chocolate and vanilla layers, from Figure 1.2, are already

baked. We simply need to bake the pistachio (MySQL) layer, assemble, and add the icing (writable layer).

1.10 Architecture of Docker Enterprise Edition 2.0

Introduction to Docker EE 2.0

- Docker EE is more than just a container orchestration solution.
- It is a full lifecycle management solution for the modernization of traditional applications and microservices across a broad set of infrastructure platforms.
- It is a Containers-as-a-Service(CaaS) platform for IT that manages and secures diverse applications across disparate infrastructure, both on-premises and in the cloud.
- Docker EE provides an integrated, tested and certified platform for apps running on enterprise Linux or Windows operating systems and Cloud providers.
- It is tightly integrated to the underlying infrastructure to provide a native, easy to install experience and an optimized Docker environment.



Docker EE 2.0 GA consists of 3 major components which together enable a full software supply chain, from image creation, to secure image storage, to secure image deployment.

Universal Control Plane 3.0.0 (application and cluster management):

Deploys applications from images, by managing orchestrators, like Kubernetes and Swarm. UCP is designed for high availability (HA). You can join multiple UCP manager nodes to the cluster, and if one manager node fails, another takes its place automatically without impact to the cluster.

Docker Trusted Registry 2.5.0:

The production-grade image storage solution from Docker.

EE Engine 17.06.2:

The commercially supported Docker engine for creating images and running them in Docker containers.

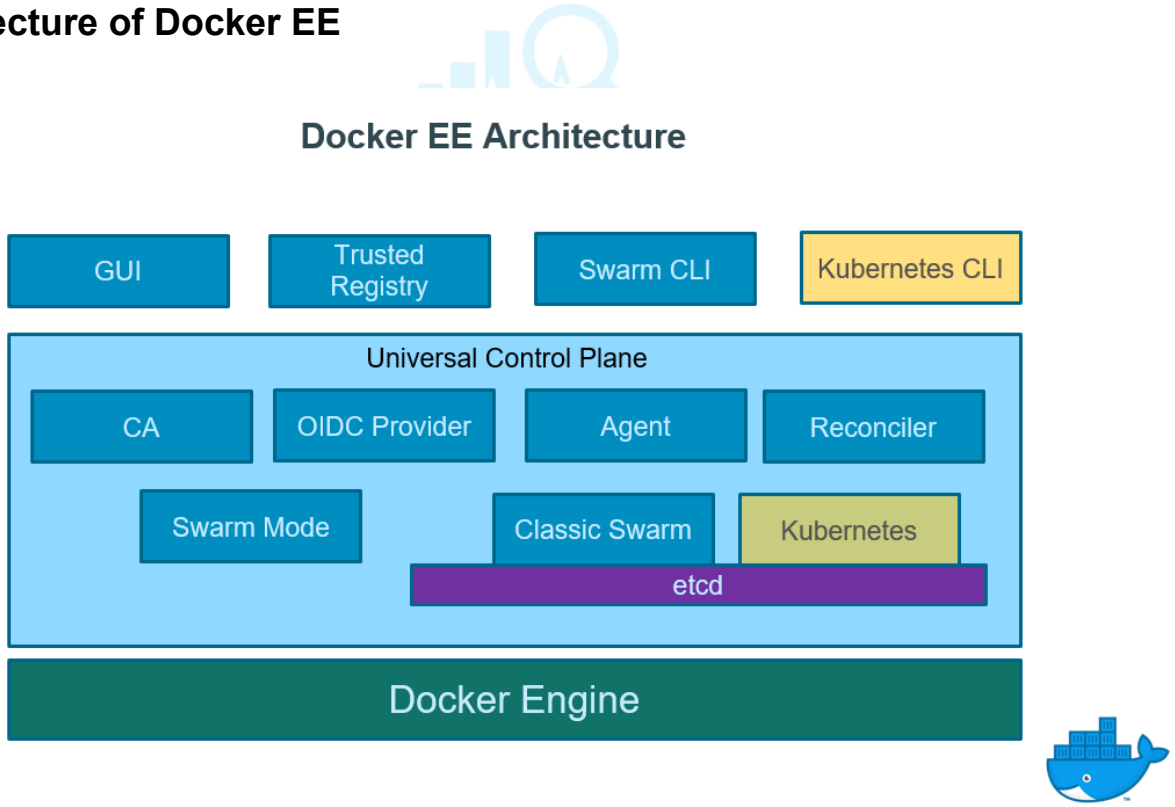
Does Docker EE support Kubernetes?

Yes, Kubernetes in Docker EE fully supports all Docker EE features, including role-based access control, LDAP/AD integration, scanning, signing enforcement, and security policies.

Kubernetes features on Docker EE include:

- Kubernetes orchestration full feature set
- CNCF Certified Kubernetes conformance
- Kubernetes app deployment by using web UI or CLI
- Compose stack deployment for Swarm and Kubernetes apps
- Role-based access control for Kubernetes workloads
- Pod-based autoscaling, to increase and decrease pod count based on CPU usage
- Blue-Green deployments, for load balancing to different app versions
- Ingress Controllers with Kubernetes L7 routing
- Interoperability between Swarm and Kubernetes workloads for networking and storage.

Architecture of Docker EE



At the base of the architecture, we have Docker Enterprise Engine. All the nodes in the cluster runs Docker Enterprise Edition as the base engine. Currently the stable release is 17.06 EE. It is a lightweight and powerful containerization technology combined with a work flow for building and containerizing your applications.

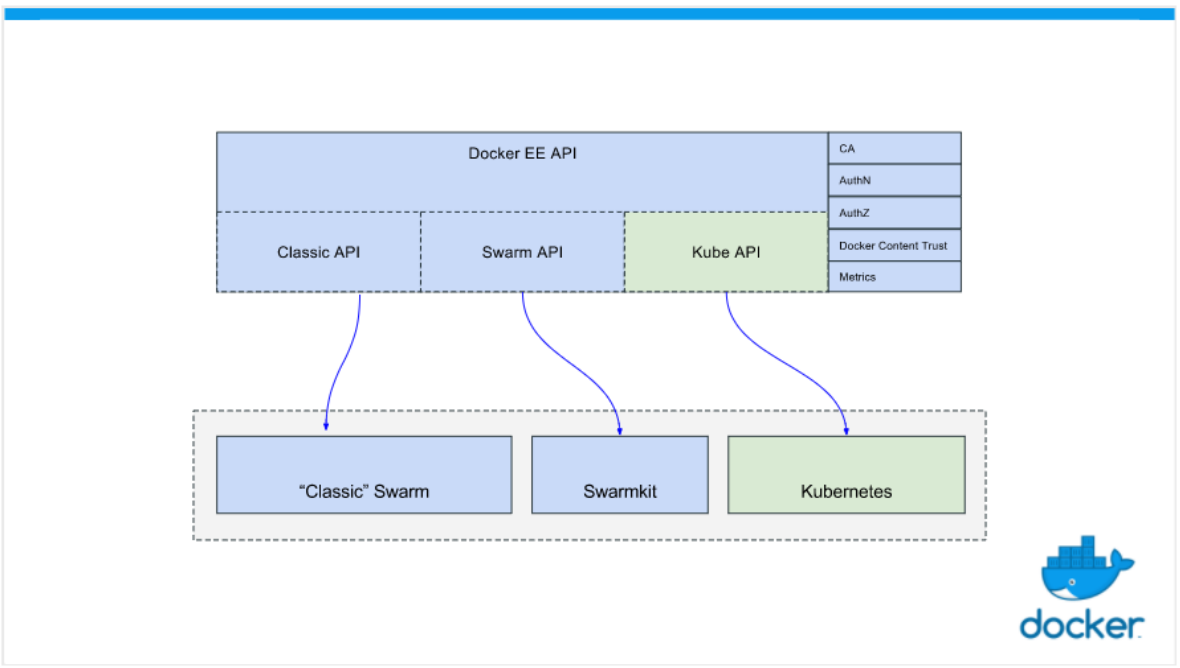
Sitting on top of the Docker Engine is what we call “Enterprise Edition Stack” where all of these components run as containers(shown below in the picture), except Swarm Mode. Swarm Mode is integrated into Docker Engine and you can turn it on/off based on your choice.Swarm Mode is enabled when Universal Control Plane (UCP) is up & running.

In case you’re new to UCP, Docker Universal Control Plane is a solution designed to deploy, manage and scale Dockerized applications and infrastructure. As a Docker native solution, full support for the Docker API ensures seamless transition of your app from development to test to production – no code changes required, support for the Docker tool-set and compatibility with the Docker ecosystem.

From infrastructure clustering and container scheduling with the embedded Docker Swarm Mode, Kubernetes to multi-container application definition with Docker Compose and image management with Trusted Registry, Universal Control Plane simplifies the process of managing infrastructure, deploying, managing applications and monitoring their activity, with a mix of graphical dashboards and Docker command line driven user interface

There are 3 orchestrators sitting on top of Docker Enterprise Engine – Docker Swarm Mode, Classic Swarm & Kubernetes. For both Classic Swarm and Kubernetes, there is the same underlying etcd instance which is highly available based on setup you have in your cluster.

Docker EE provides access to the full API sets of three popular orchestrators:



- Kubernetes: Full YAML object support
- SwarmKit: Service-centric, Compose file version 3
- “Classic” Swarm: Container-centric, Compose file version 2

Docker EE proxies the underlying API of each orchestrator, giving you access to all of the capabilities of each orchestrator, along with the benefits of Docker EE, like role-based access control and Docker Content Trust.

To manage the life cycle of an orchestrator, we have a component called Agents and reconciler which manages both the promotion and demotion flows as well as certification renewal rotation and deal with patching and upgrading.

Agent is the core component of UCP and is a globally-scheduled service. When you install UCP on a node, or join a node to a swarm that's being managed by UCP, the agent service starts running on that node. Once this service is running, it deploys containers with other UCP components, and it ensures they keep running. The UCP components that are deployed on a node depend on whether the node is a manager or a worker. In nutshell, it monitors the node and ensures the right UCP services are running.

Another important component is Reconciler. When a UCP agent detects that the node is not running the right UCP components, it starts the reconciler container to converge the node to its desired state. It is expected for the UCP reconciler container to remain in an excited state when the node is healthy.

In order to integrate with K8s and support authentication, Docker Team built an open ID connect provider. In case you're completely new to OIDC, OpenID Connect is a simple identity layer on top of the OAuth 2.0 protocol. It allows Clients to verify the identity of the End-User based on the authentication performed by an Authorization Server, as well as to obtain basic profile information about the End-User in an inter-operable and REST-like manner.

As Swarm Mode is designed, we have central certificate authority that manages all the certificates and cryptographic node identity of the cluster. All components uses mutual TLS communication and they are using nodes which are issue by certificate authority. We are using an unmodified version of Kubernetes.

Sitting on the top of the stack, there is GUI which uses Swarm APIs and Kubernetes APIs. Right next to GUI, we have Docker Trusted Registry which is an enterprise-ready on-premises service for storing, distributing and securing images. It gives enterprises the ability to ensure secure collaboration between developers and sysadmins to build, ship and run applications. Finally we have Docker & Kubernetes CLI capability integrated. Please note that this uses un-modified version of Kubernetes API.