Docker isn't a virtualization methodology. It relies on other tools that actually implement container-based virtualization or operating system level virtualization. For that, Docker was initially using LXC driver, then moved to libcontainer which is now renamed as runc. Docker primarily focuses on automating the deployment of applications inside application containers. Application containers are designed to package and run a single service, whereas system containers are designed to run multiple processes, like virtual machines. So, Docker is considered as a container management or application deployment tool on containerized systems.

In order to know how it is different from other virtualizations, let's go through virtualization and its types. Then, it would be easier to understand what's the difference there.

**Virtualization**

In its conceived form, it was considered a method of logically dividing mainframes to allow multiple applications to run simultaneously. However, the scenario drastically changed when companies and open source communities were able to provide a method of handling the privileged instructions in one way or another and allow for multiple operating systems to be run simultaneously on a single x86 based system.

**Hypervisor**

The hypervisor handles creating the virtual environment on which the guest virtual machines operate. It supervises the guest systems and makes sure that resources are allocated to the guests as necessary. The hypervisor sits in between the physical machine and virtual machines and provides virtualization services to the virtual machines. To realize it, it intercepts the guest operating system operations on the virtual machines and emulates the operation on the host machine's operating system.

The rapid development of virtualization technologies, primarily in cloud, has driven the use of virtualization further by allowing multiple virtual servers to be created on a single physical server with the help of hypervisors, such as Xen, VMware Player, KVM, etc., and incorporation of hardware support in commodity processors, such as Intel VT and AMD-V.

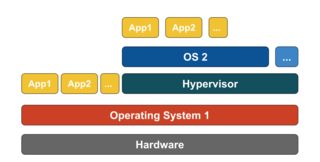
**Types of Virtualization**

The virtualization method can be categorized based on how it mimics hardware to a guest operating system and emulates a guest operating environment. Primarily, there are three types of virtualization:

* Emulation
* Paravirtualization
* Container-based virtualization

**Emulation**

Emulation, also known as full virtualization runs the virtual machine OS kernel entirely in software. The hypervisor used in this type is known as Type 2 hypervisor. It is installed on the top of the host operating system which is responsible for translating guest OS kernel code to software instructions. The translation is done entirely in software and requires no hardware involvement. Emulation makes it possible to run any non-modified operating system that supports the environment being emulated. The downside of this type of virtualization is an additional system resource overhead that leads to a decrease in performance compared to other types of virtualizations.

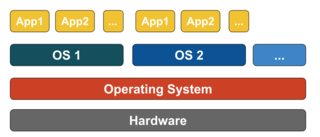
[](https://i.stack.imgur.com/FS8uzm.png)

Examples in this category include VMware Player, VirtualBox, QEMU, Bochs, Parallels, etc.

**Paravirtualization**

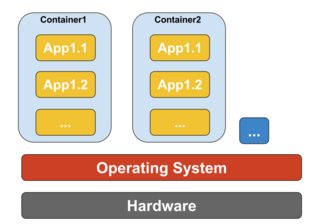
Paravirtualization, also known as Type 1 hypervisor, runs directly on the hardware, or “bare-metal”, and provides virtualization services directly to the virtual machines running on it. It helps the operating system, the virtualized hardware, and the real hardware to collaborate to achieve optimal performance. These hypervisors typically have a rather small footprint and do not, themselves, require extensive resources.

Examples in this category include Xen, KVM, etc.

[](https://i.stack.imgur.com/oPO6Um.png)

**Container-based Virtualization**

Container-based virtualization, also known as operating system-level virtualization, enables multiple isolated executions within a single operating system kernel. It has the best possible performance and density and features dynamic resource management. The isolated virtual execution environment provided by this type of virtualization is called a container and can be viewed as a traced group of processes.

[](https://i.stack.imgur.com/qTuWpm.png)

The concept of a container is made possible by the namespaces feature added to Linux kernel version 2.6.24. The container adds its ID to every process and adding new access control checks to every system call. It is accessed by the *clone()* system call that allows creating separate instances of previously-global namespaces.

Namespaces can be used in many different ways, but the most common approach is to create an isolated container that has no visibility or access to objects outside the container. Processes running inside the container appear to be running on a normal Linux system although they are sharing the underlying kernel with processes located in other namespaces, same for other kinds of objects. For instance, when using namespaces, the root user inside the container is not treated as root outside the container, adding additional security.

The Linux Control Groups (cgroups) subsystem, the next major component to enable container-based virtualization, is used to group processes and manage their aggregate resource consumption. It is commonly used to limit the memory and CPU consumption of containers. Since a containerized Linux system has only one kernel and the kernel has full visibility into the containers, there is only one level of resource allocation and scheduling.

Several management tools are available for Linux containers, including LXC, LXD, systemd-nspawn, lmctfy, Warden, Linux-VServer, OpenVZ, Docker, etc.

**Containers vs Virtual Machines**

Unlike a virtual machine, a container does not need to boot the operating system kernel, so containers can be created in less than a second. This feature makes container-based virtualization unique and desirable than other virtualization approaches.

Since container-based virtualization adds little or no overhead to the host machine, container-based virtualization has near-native performance

For container-based virtualization, no additional software is required, unlike other virtualizations.

All containers on a host machine share the scheduler of the host machine saving need of extra resources.

Container states (Docker or LXC images) are small in size compared to virtual machine images, so container images are easy to distribute.

Resource management in containers is achieved through cgroups. Cgroups does not allow containers to consume more resources than allocated to them. However, as of now, all resources of host machine are visible in virtual machines, but can't be used. This can be realized by running top or htop on containers and host machine at the same time. The output across all environments will look similar.

Update:

**How does Docker run containers in non-Linux systems?**

If containers are possible because of the features available in the Linux kernel, then the obvious question is how do non-Linux systems run containers. Both Docker for Mac and Windows use Linux VMs to run the containers. Docker Toolbox used to run containers in Virtual Box VMs. But, the latest Docker uses Hyper-V in Windows and Hypervisor.framework in Mac.

**VM**:

A virtual machine emulates a physical computing environment, but requests for CPU, memory, hard disk, network and other hardware resources are managed by a virtualization layer which translates these requests to the underlying physical hardware.

In this context the VM is called as the Guest while the environment it runs on is called the host.

**LXC**s:

Linux Containers (LXC) are operating system-level capabilities that make it possible to run multiple isolated Linux containers, on one control host (the LXC host). Linux Containers serve as a lightweight alternative to VMs as they don’t require the hypervisors viz. Virtualbox, KVM, Xen, etc.

Now unless you were drugged by Alan (Zach Galifianakis- from the Hangover series) and have been in Vegas for the last year, you will be pretty aware about the tremendous spurt of interest for Linux containers technology, and if I will be specific one container project which has created a buzz around the world in last few months is – Docker leading to some echoing opinions that cloud computing environments should abandon virtual machines (VMs) and replace them with containers due to their lower overhead and potentially better performance.

But the big question is, is it feasible?, will it be sensible?

a. LXCs are scoped to an instance of Linux. It might be different flavors of Linux (e.g. a Ubuntu container on a CentOS host but it’s still Linux.) Similarly, Windows-based containers are scoped to an instance of Windows now if we look at VMs they have a pretty broader scope and using the hypervisors you are not limited to operating systems Linux or Windows.

b. LXCs have low overheads and have better performance as compared to VMs. Tools viz. Docker which are built on the shoulders of LXC technology have provided developers with a platform to run their applications and at the same time have empowered operations people with a tool that will allow them to deploy the same container on production servers or data centers. It tries to make the experience between a developer running an application, booting and testing an application and an operations person deploying that application seamless, because this is where all the friction lies in and purpose of DevOps is to break down those silos.

So the best approach is the cloud infrastructure providers should advocate an appropriate use of the VMs and LXC, as they are each suited to handle specific workloads and scenarios.

Abandoning VMs is not practical as of now. So both VMs and LXCs have their own individual existence and importance.