IoT: Cloud Services

Containers

Motivation

- The Internet of Things, the highly connected network of smart devices, such as environmental sensors, medical trackers, home appliances, and industrial devices, is growing rapidly.
- By 2020, a predicted 20 billion devices will be connected, which is more than twice the number of PCs, smartphones, and tablets combined.
- Containers are a lightweight approach to virtualization that developers can apply to rapidly develop, test, deploy, and update IoT applications at scale.
- IoT applications target a wide variety of device platforms. During prototyping and early phases of development, IoT projects are often developed using generic microcontroller-based development boards or single-board computers (SBCs) like the Raspberry Pi.

From microservices to containers

- Agile software development is a broadly adopted methodology in enterprises today.
- "full-stack-responsibility" for the individual services.
  - infrastructures need to scale differently and the self-service model for projects is taking center stage.
- Containers are the foundation for this.
  - Along with proper DevOps and orchestration.

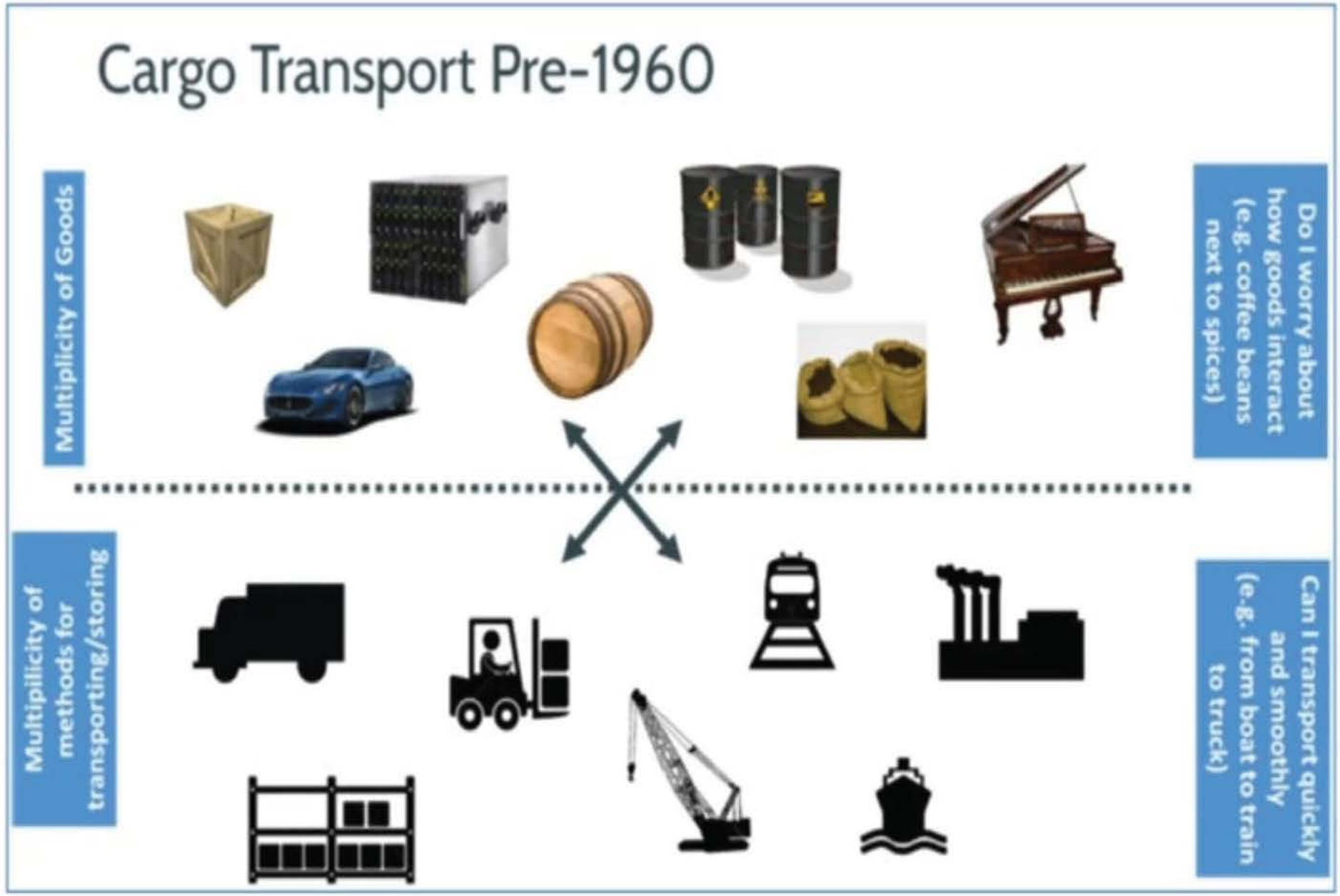
Containers - New Degree of Freedom

New concept of virtualization solution for PaaS and IaaS due to container's increased density, isolation, elasticity and rapid provisioning

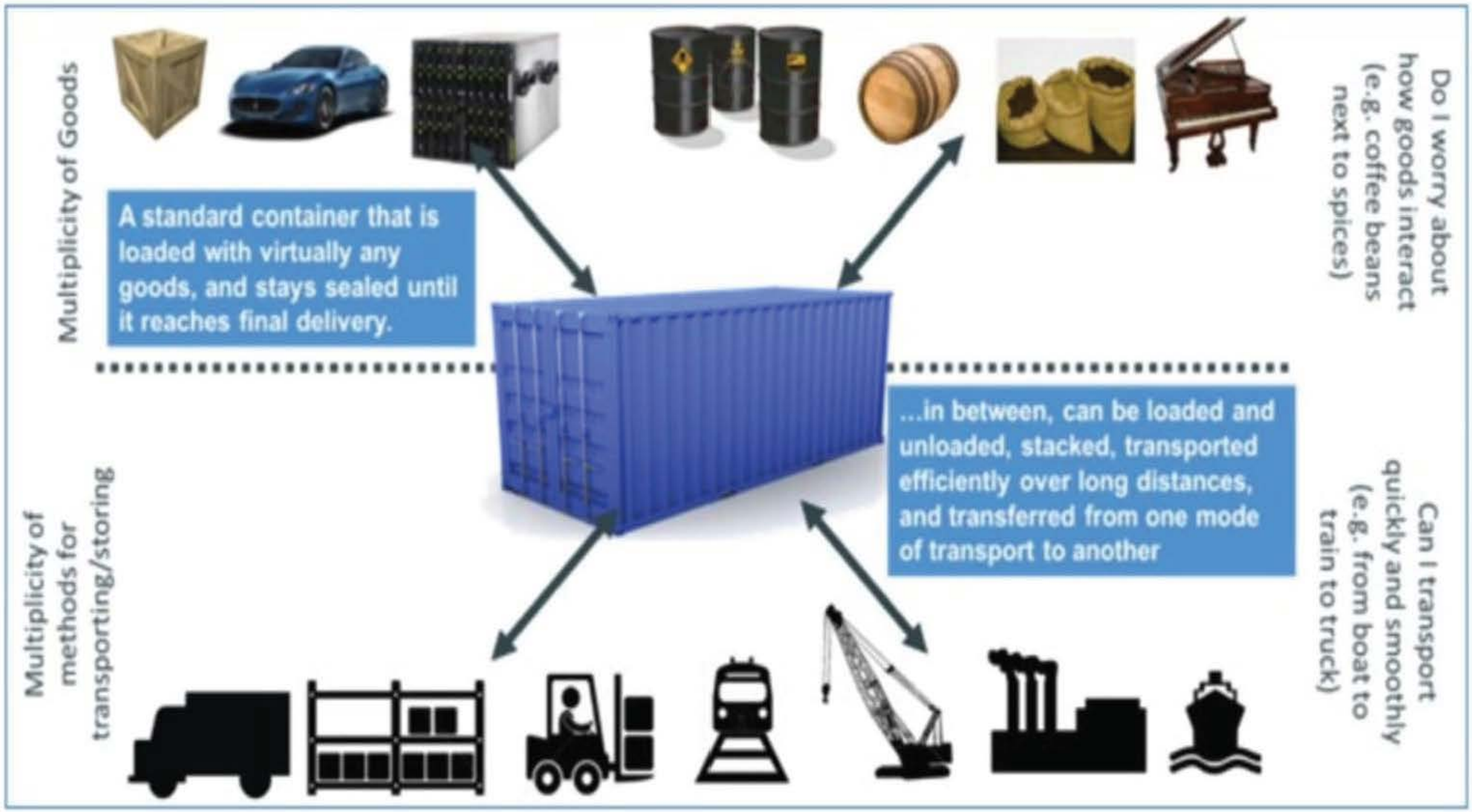
Containerization:

- Use lightweight packages instead of full VMs
- Move from a single large monolithic app to composition of microservices
- Containerize different parts of an application
- Move parts of apps into different types of cloud infrastructure
- Simplify migration of applications between private, public and hybrid clouds

History of Cargo Transportation

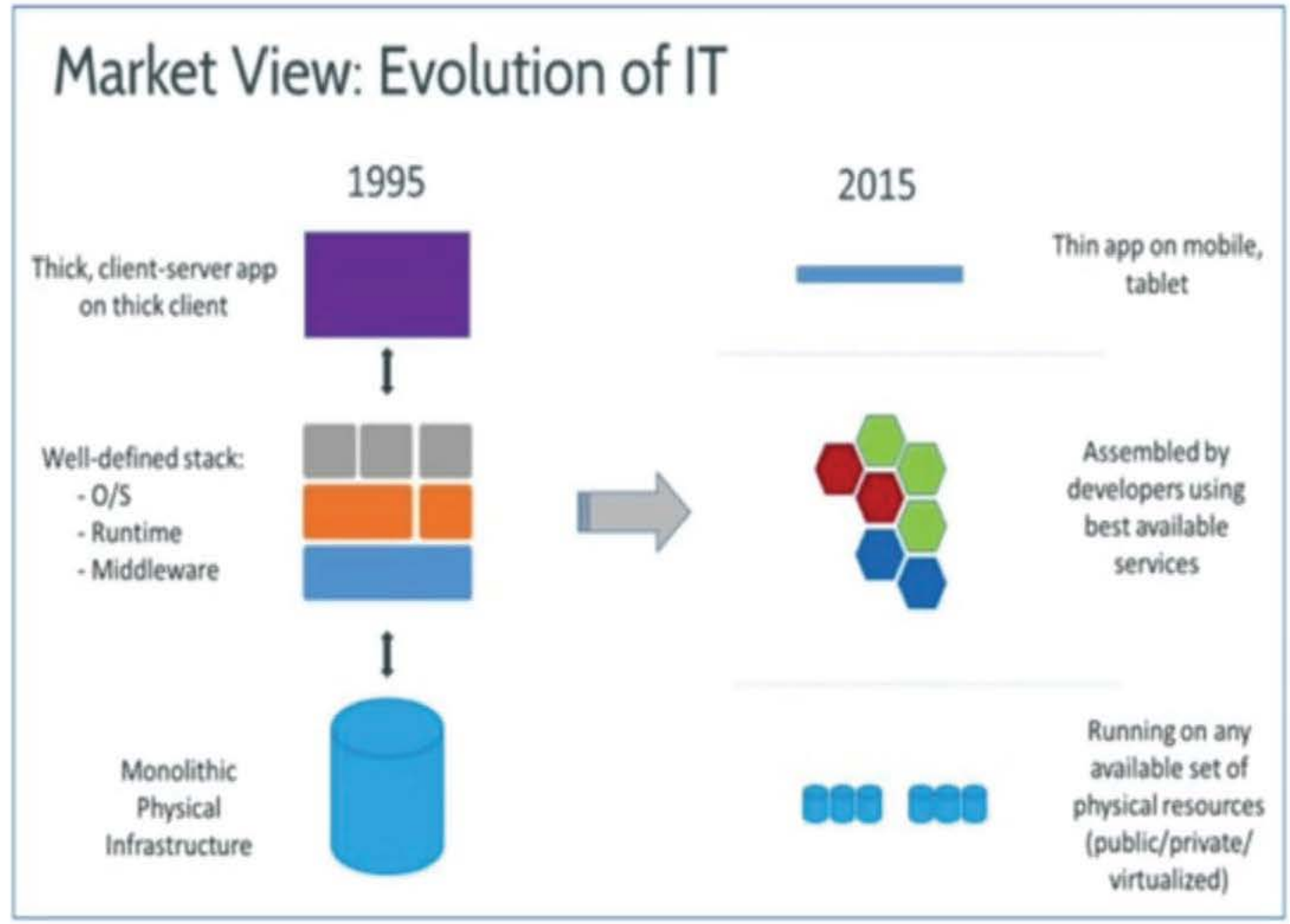


Solution to shipping Challenge

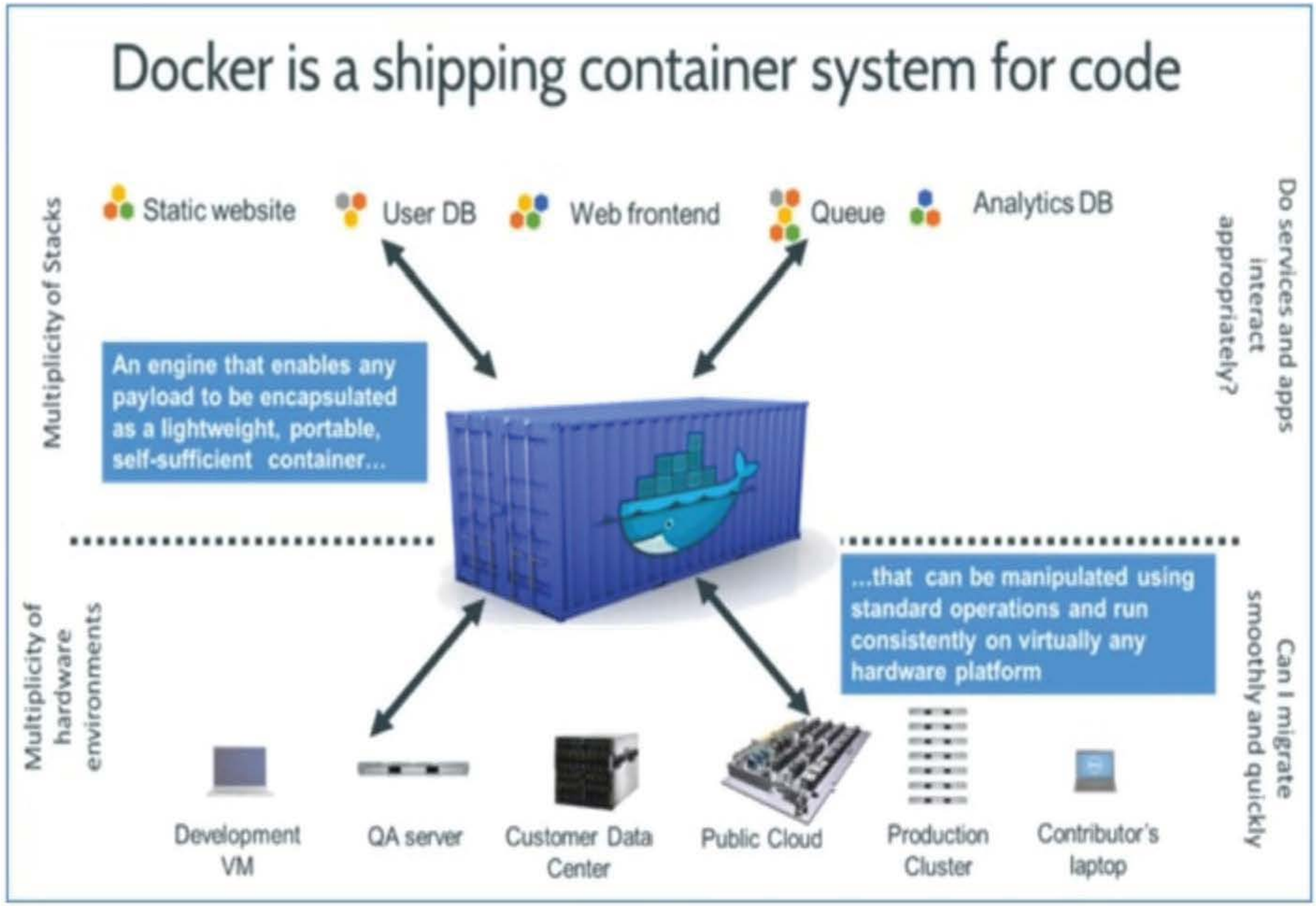


Analogy with Virtualization

- Historical way: $M mainframes
- Modern way: virtual machines
- Problem: performance overhead



The Solution to software shipping



IoT: Cloud Services

Containers or VMs

Why use Containers?

- Reduces build & deploy times
- Cost control and granularity
- Libraries dependency can be complicated
- Infrastructure configuration = spaghetti

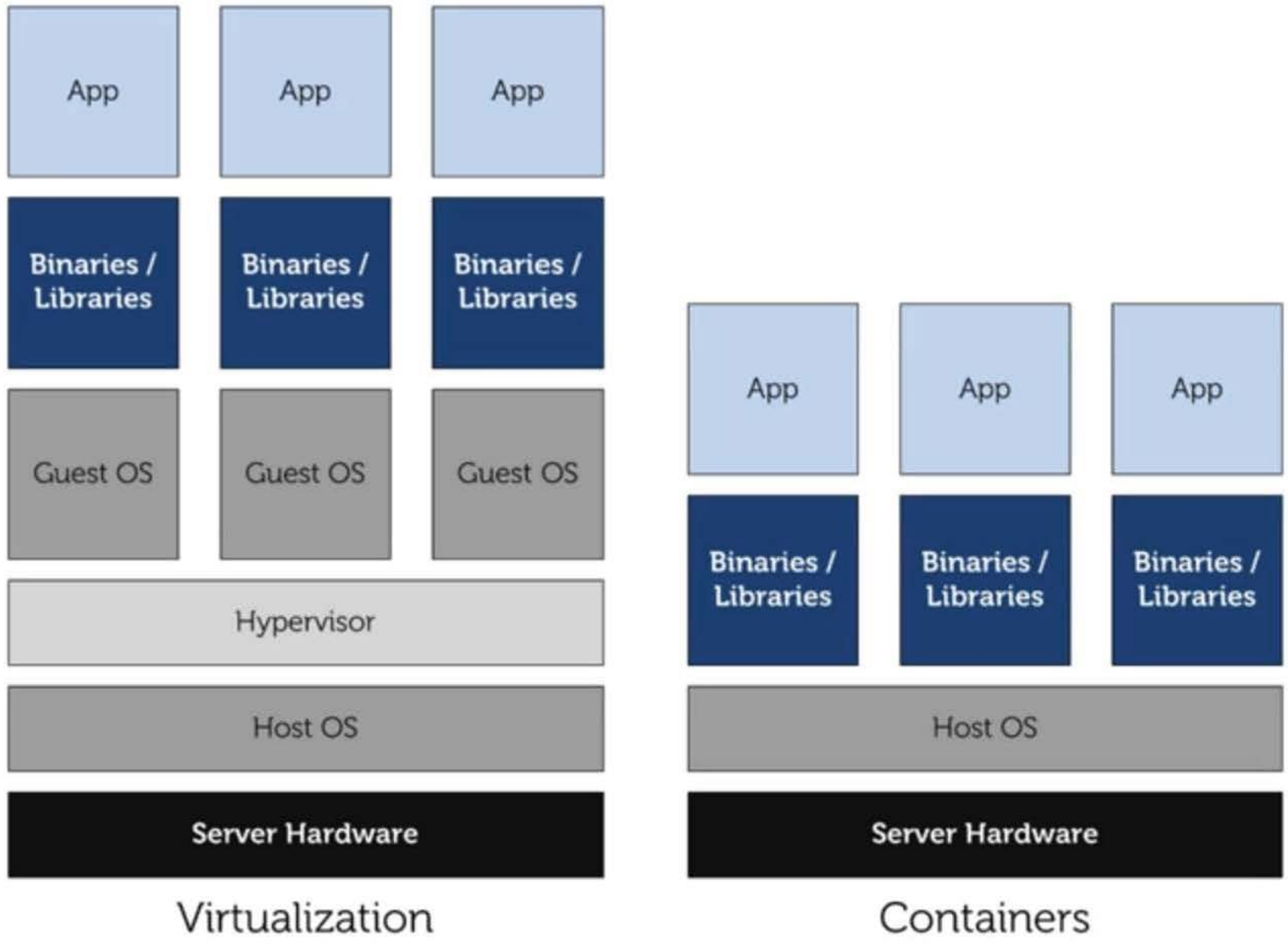
Containers vs VMs

Container technology simplifies cloud portability.

- Run same application in different environments

1. A container encapsulates applications and defines their interface with the surrounding system
2. In a virtual machine: a full OS install with the associated overhead of virtualized device drivers, etc.,
   - Containers use and share the OS and device drivers of the host.
3. Virtual machines have a full OS with its own memory management, device drivers, daemons, etc.
   - Containers share the host's OS and are therefore lighter weight.

Containers vs VMs



Containers can and can't

Can

- Get shell (i.e. ssh)
- Own process space
- Own network interface
- Run as root
- Install packages
- Run services
- …

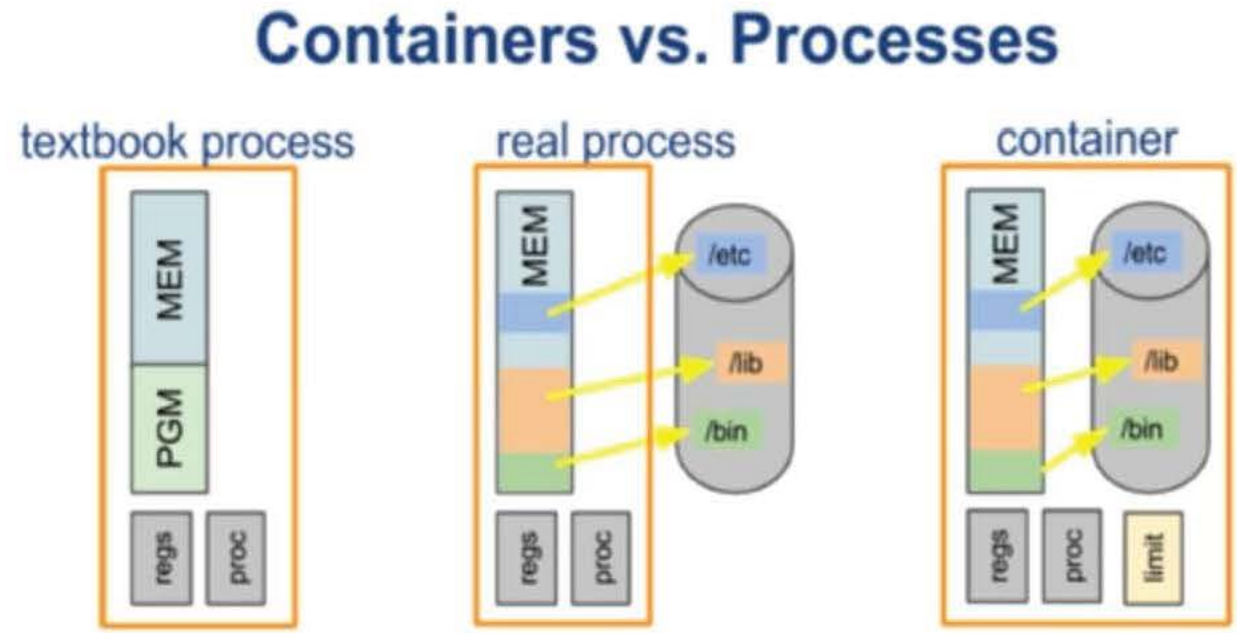
Can't

- Use the host kernel
- Boot a different OS
- Have its own modules
- Doesn't need *init* as PID 1

Containers vs Processes

Containers are processes with their full environment.

- A computer science textbook will define a process as having its own address space, program, CPU state, and process table entry.
- The program text is actually memory mapped from the filesystem into the process address space and often consists of dozens of shared libraries in addition to the program itself, thus all these files are really part of the process.

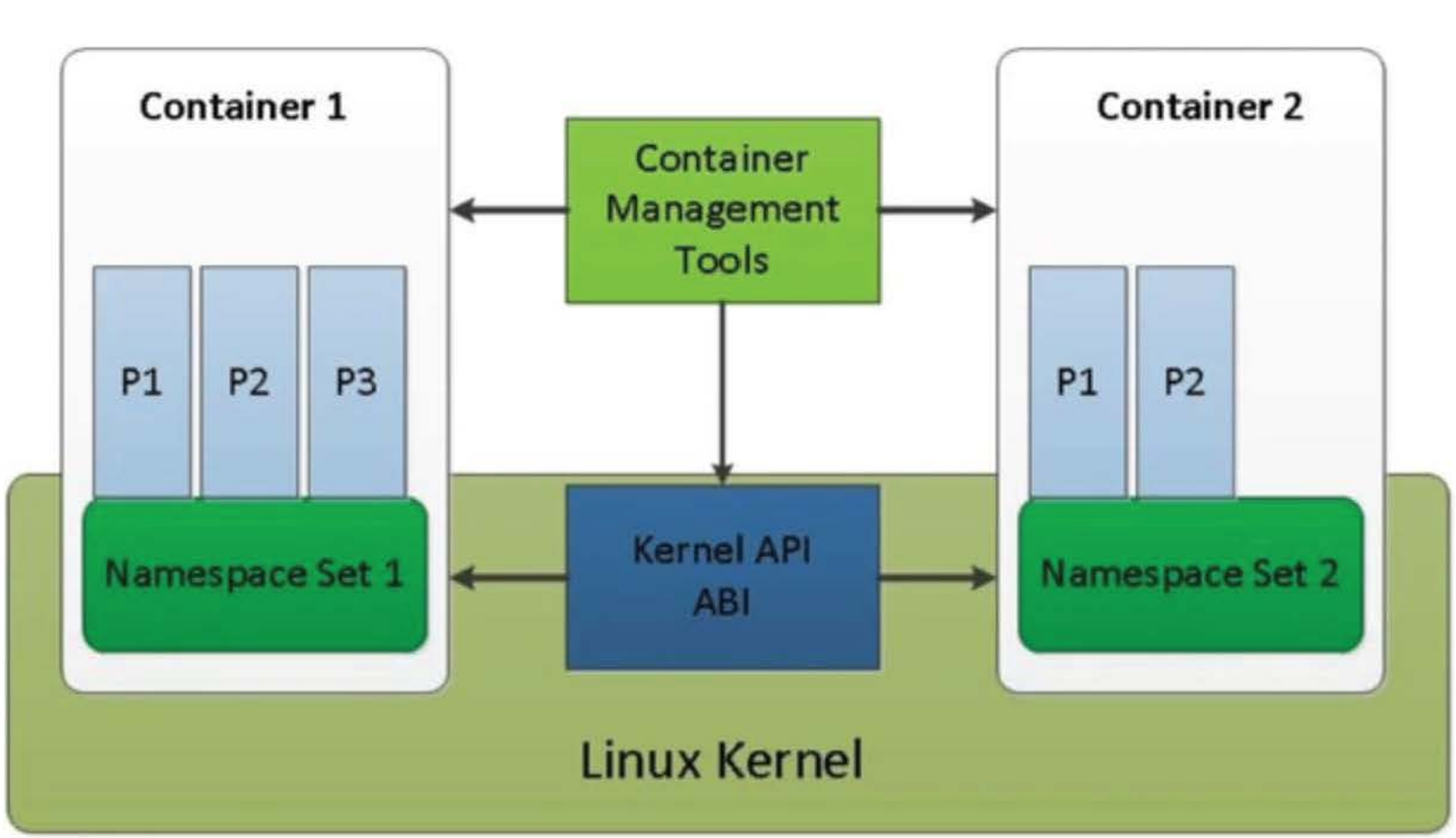


IoT: Cloud Services

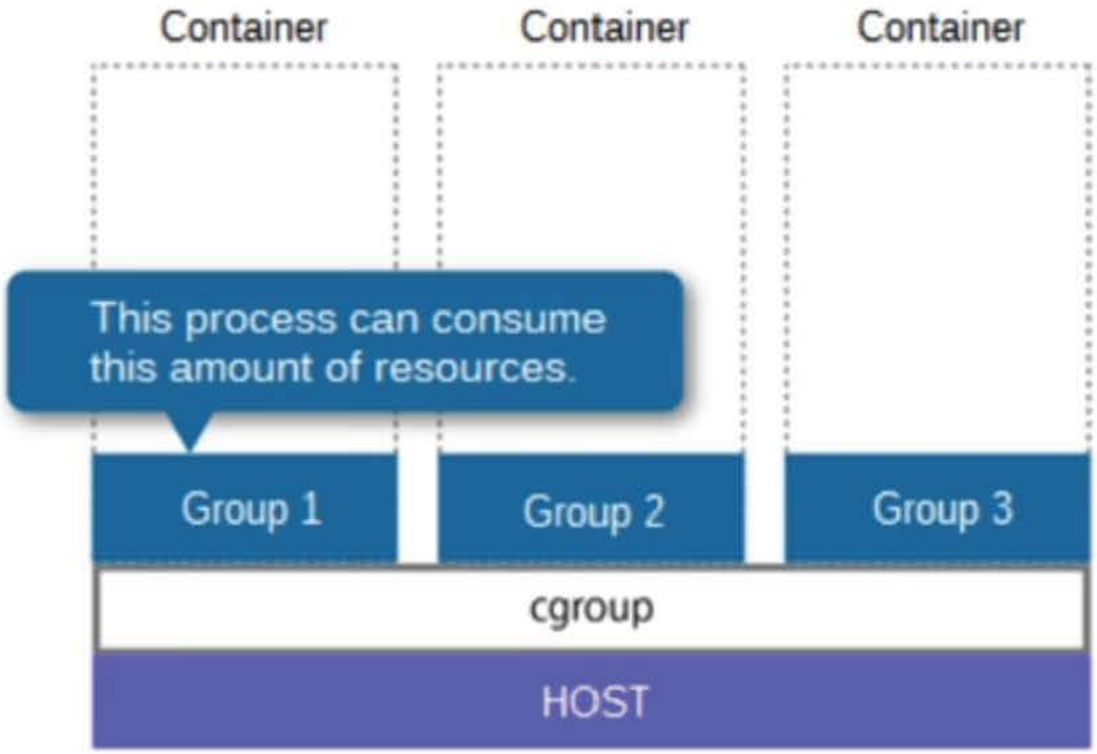
Containers - behind the hood

LinuX Containers (LXC)

- Operation System level virtualization method for Linux
- LXC (**L**inu**X** **C**ontainers) is an OS virtual for running isolated Linux systems (containers) on a single control host



Is containers a new thing?



Cgroups is a mechanism to control resources per hierarchical groups of processes

cgroups (abbreviated from **control groups**) is a Linux kernel feature to limit, account, and isolate resource usage (CPU, memory, disk I/O, etc.) of process groups.

- This work was started by engineers at Google in 2006 under the name "*process containers*".
- In late 2007, it was renamed to "Control Groups" due to the confusion caused by multiple meanings of the term "container" in the Linux kernel, and merged into kernel version 2.6.24.
- Since then, new features and controllers have been added.

Cgroups design goals

- Provide a unified interface to many different use cases, from controlling single processes to whole operating system-level virtualization
- Cgroups provides:
  - ***Resource limitation:*** groups can be set to not exceed a set memory limit — this also includes file system cache.
    - The original paper was presented at Linux Symposium and can be found at ***Containers:*** *Challenges with the memory resource controller and its performance.*
- ***Prioritization****:* some groups may get a larger share of CPU or disk I/O throughput.
- ***Accounting:*** to measure how much resources certain systems use for e.g. billing purposes.
- ***Control:*** freezing groups or check-pointing and restarting.

Namespaces

The purpose of each namespace is to wrap a particular global system resource in an abstraction that makes it appear to the processes within the namespace that they have their own isolated instance of the global resource.

Cgroups vs namespaces

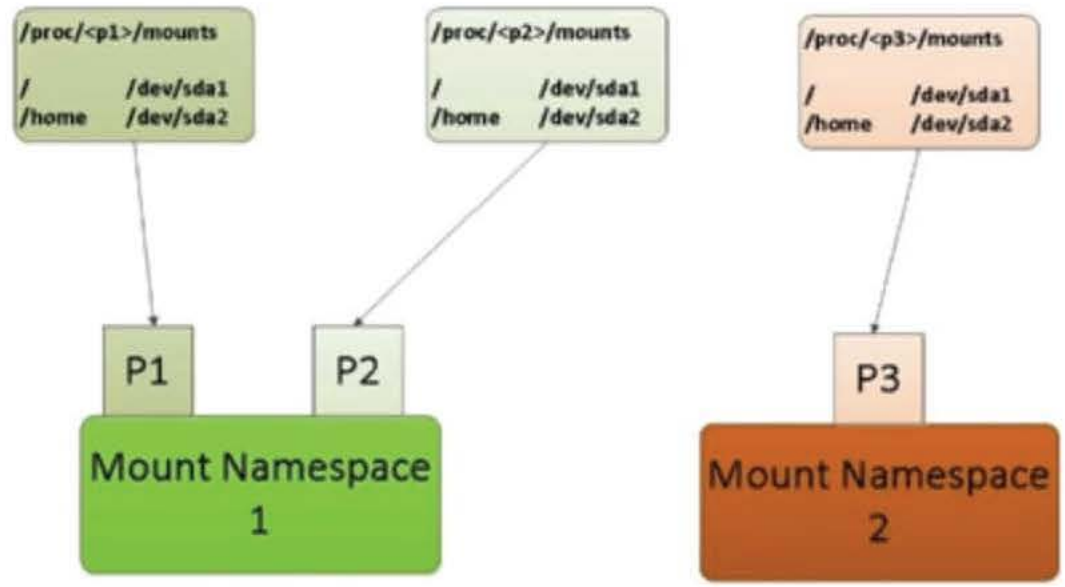
- **Cgroups**: limits how much you can use
- **Namespaces**: limits how much you can see (and use)

Namespaces

- mnt
- uts
- ipc
- net
- pid
- usr

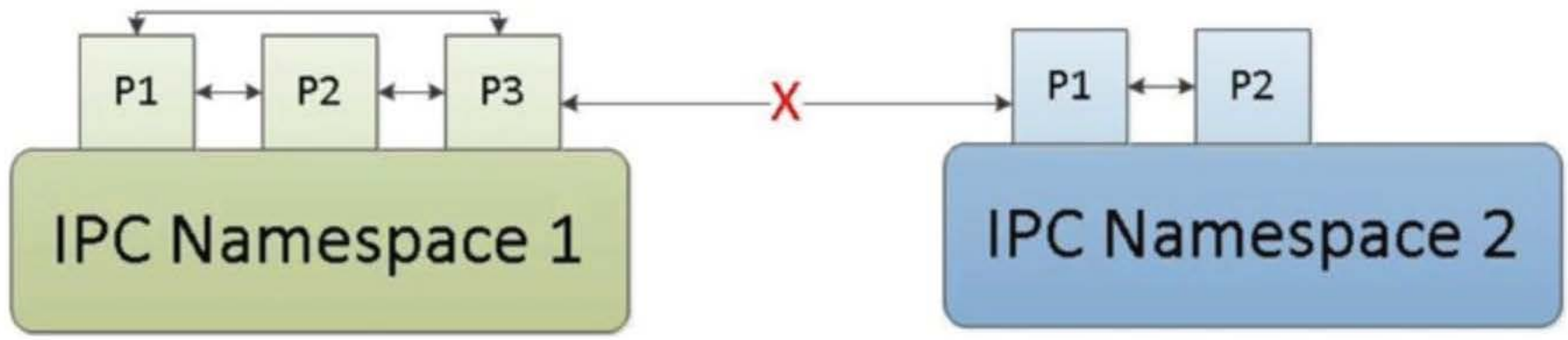
Mount namespace

- The “mount namespace” isolates the set of filesystem mount points seen by a group of processes
  - processes in a different mount namespaces can have different view of the filesystem hierarchy.
- Mounts can be private or shared
- The tasks running in each mount namespace doing mount/unmount will not affect the file system layout of the other mount namespace.



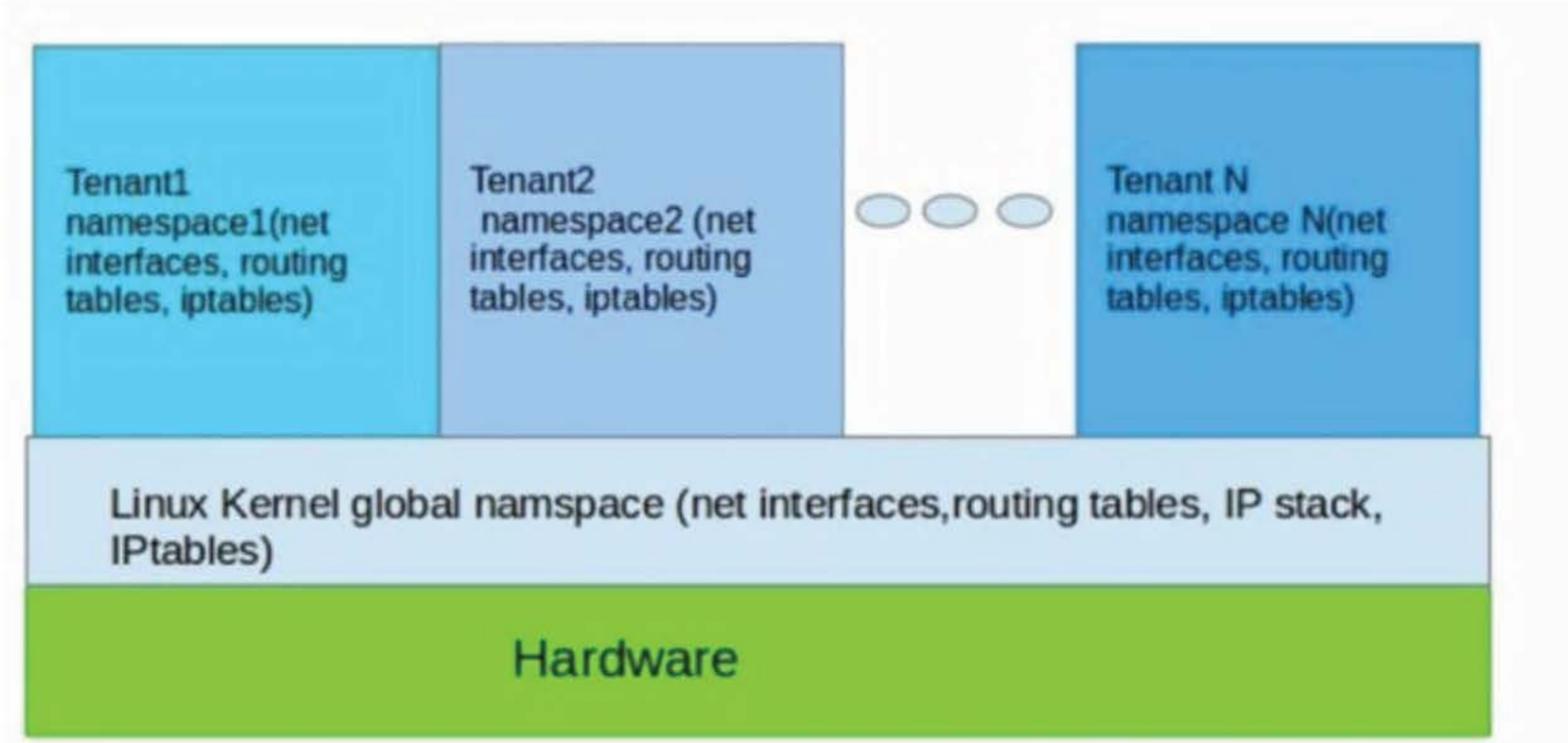
IPC Namespace

- IPC namespace isolates the inter-process communication resources.
- A process or a group of processes have their own:
  - IPC shared memory
  - IPC semaphore
  - IPC message queues



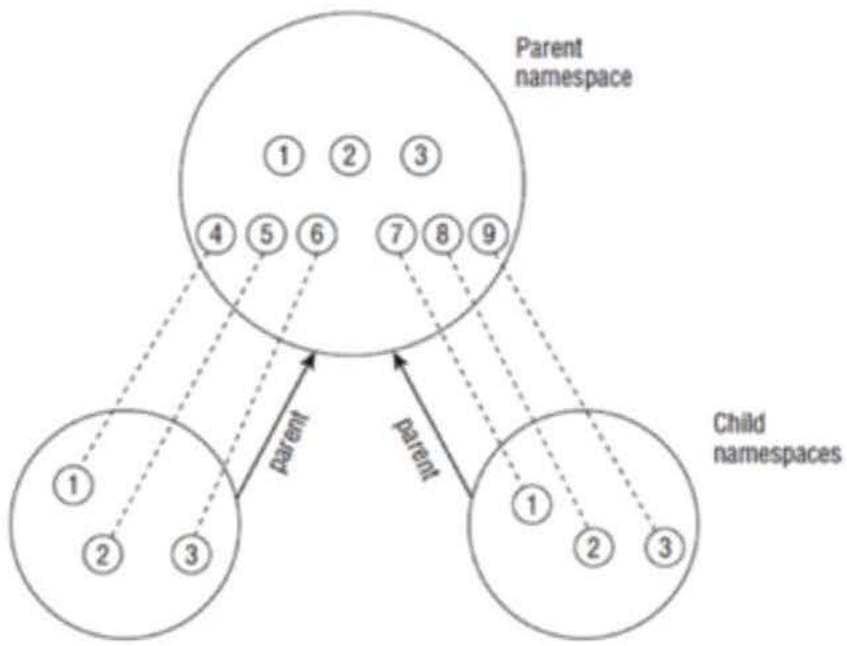
Network Namespace

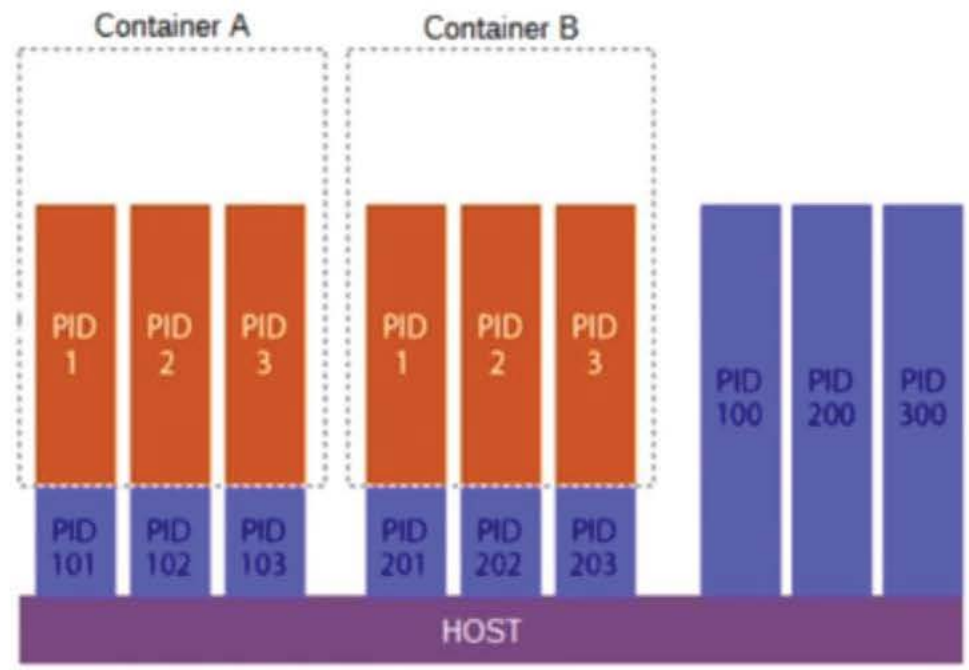
Net namespace isolates the networking related resources
Every net namespace has independent
    Network devices, IP addresses, firewall rules, routing table, sockets etc.



PID Namespace

- PID namespace isolates the Process ID, implemented as a hierarchy.
- PID namespace is a hierarchy comprised of “Parent” and “Child”.
  - The parent pid-ns can have many children pid-ns. Each Child pid-ns only has one parent pidns.





Other Namespaces

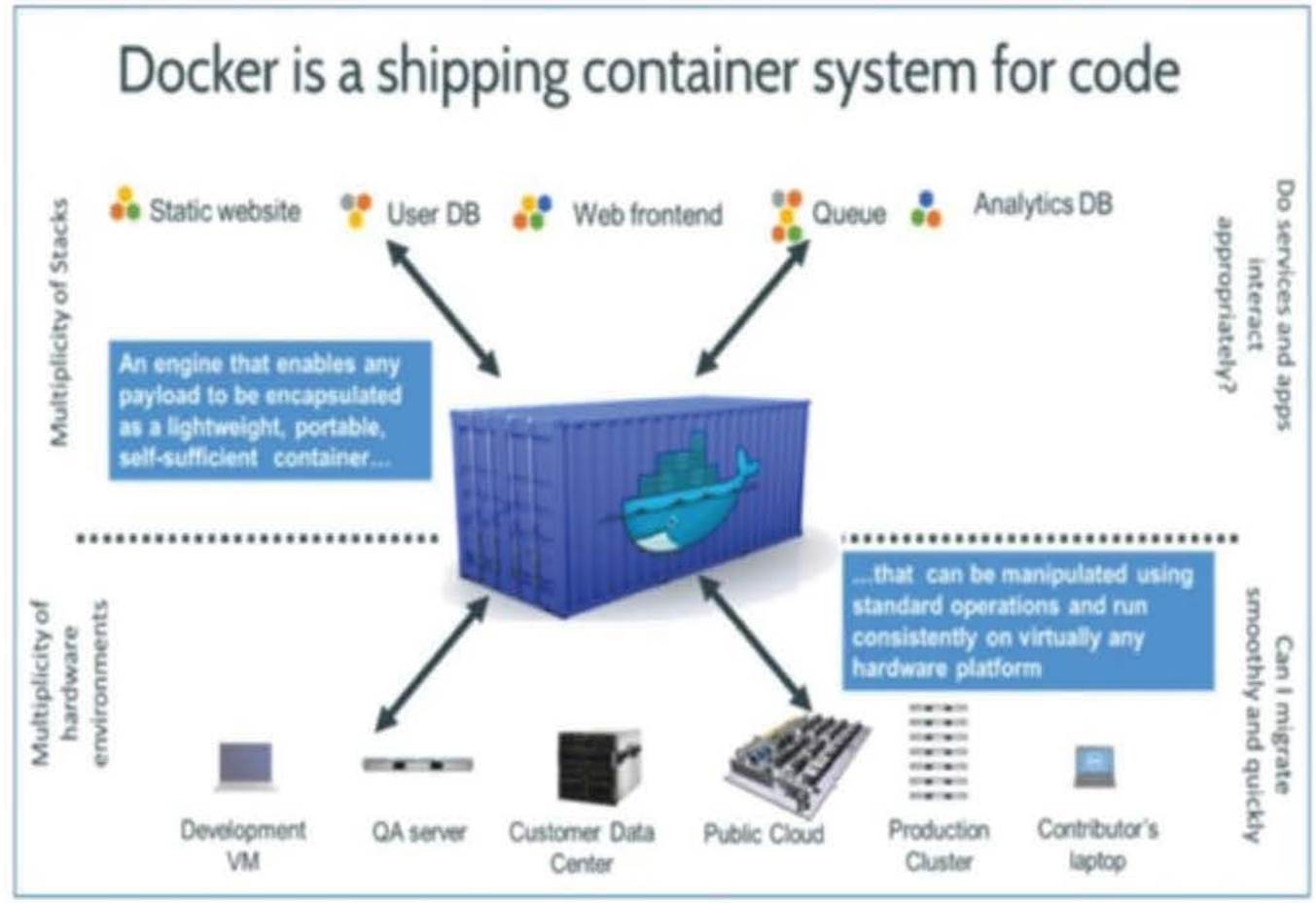
- UTS namespace
  - UTS = UNIX Time-sharing System
  - Each process has a separate copy of the hostname and the (now mostly unused) NIS domain name isolation!
  - In containers: useful for init and config scripts that tailor their actions based on the names
- User namespace
  - kuid/kgid: Original uid/gid, Global
  - uid/gid: user id in user namespace, will be translated to kuid/kgid finally
  - Only parent User NS has rights to set map

IoT: Cloud Services

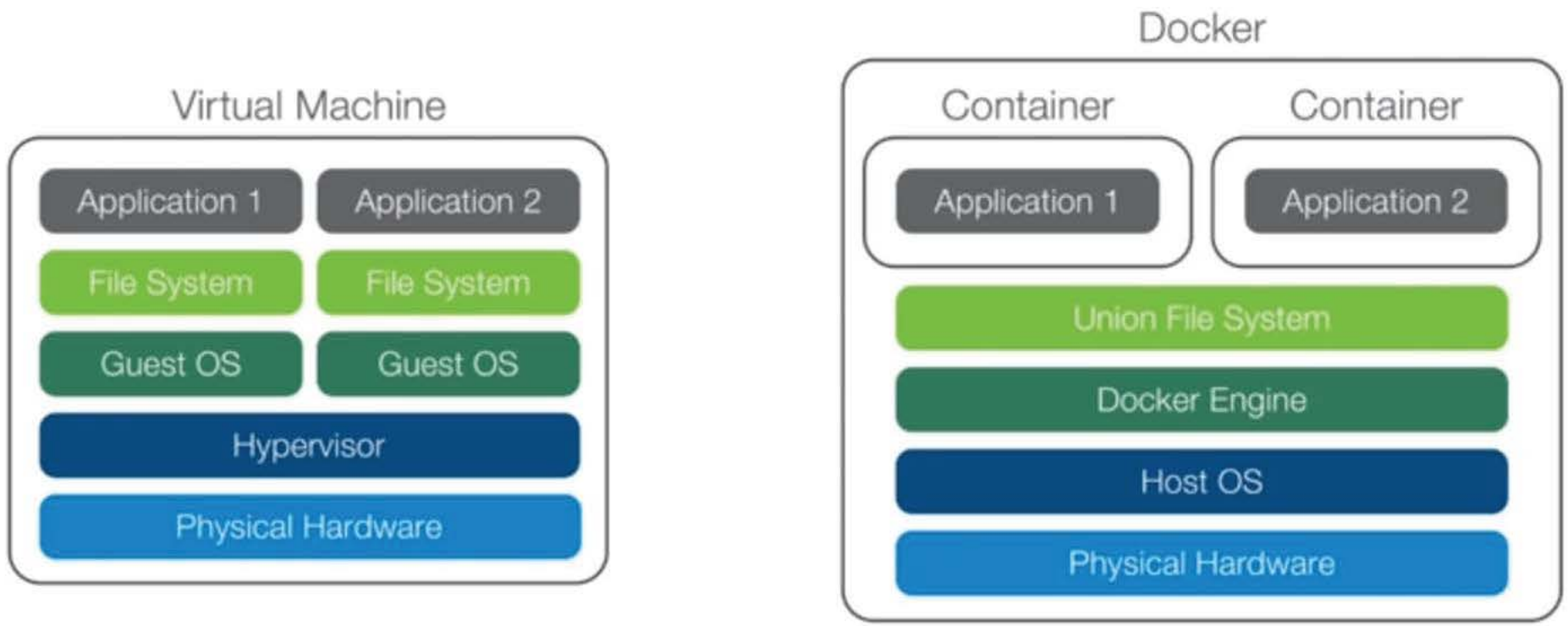
Docker

So what is Docker?

Docker is an open-source project that automates the deployment of applications inside software containers, by providing an additional layer of abstraction and automation of operating system–level virtualization on Linux.



VMs vs Containers vs Docker



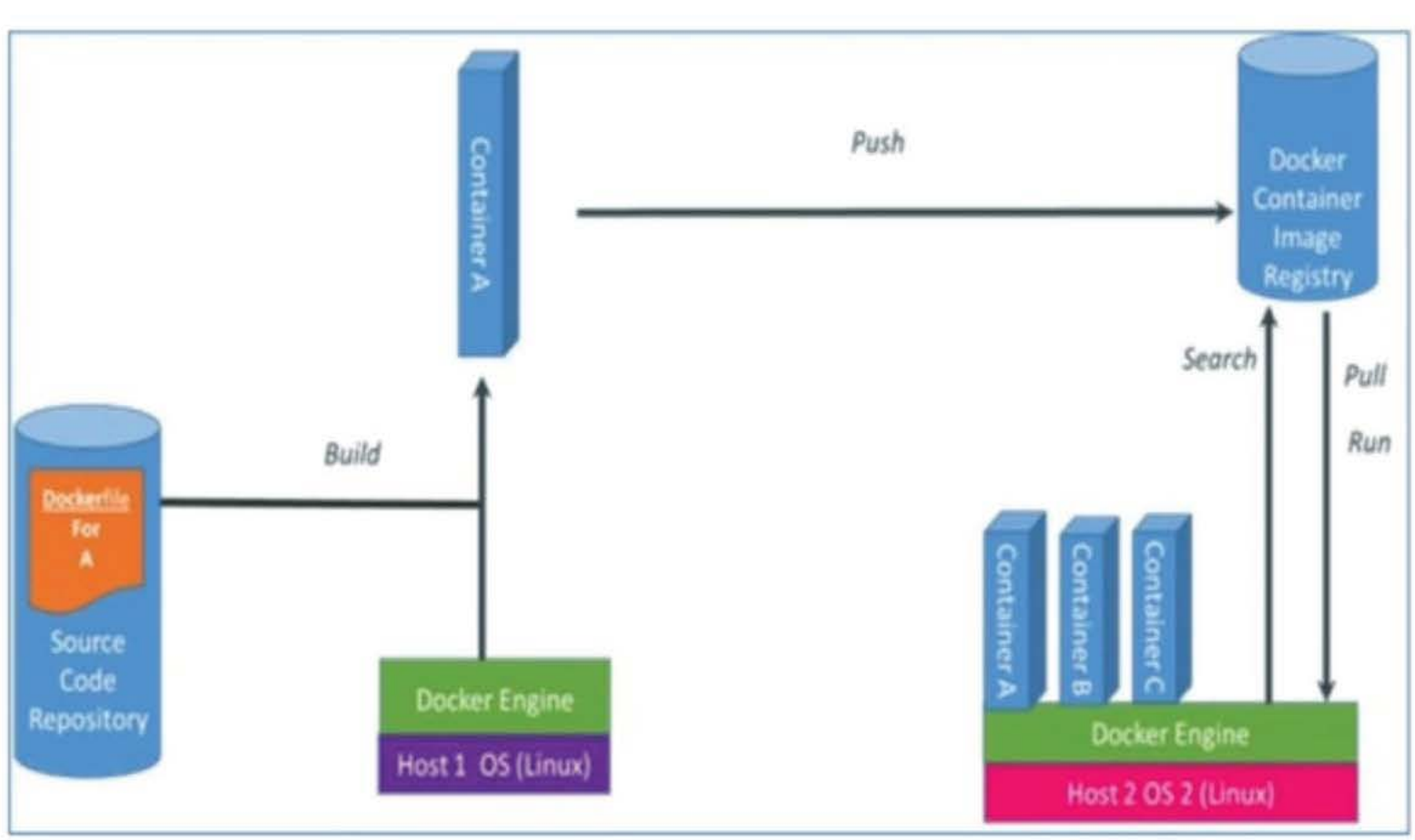
Why Docker?

Docker makes it easy to build, modify, publish, search, and run containers.

- With Docker, a container comprises both an application and all of its dependencies.
- Containers can either be created manually or, if a source code repository contains a DockerFile, automatically.
- Subsequent modifications to a baseline Docker image can be committed to a new container using the Docker Commit Function and then Pushed to a Central Registry.

How Docker Works

- Containers can be found in a Docker Registry (either public or private), using Docker Search.
- Containers can be pulled from the registry using Docker Pull and can be run, started, stopped, etc.



Docker use cases

- Development Environment
- Environments for Integration Tests
- Quick evaluation of software
- Microservices
- Multi-Tenancy
- Unified execution environment (dev test to prod (local, VM, cloud, ...)