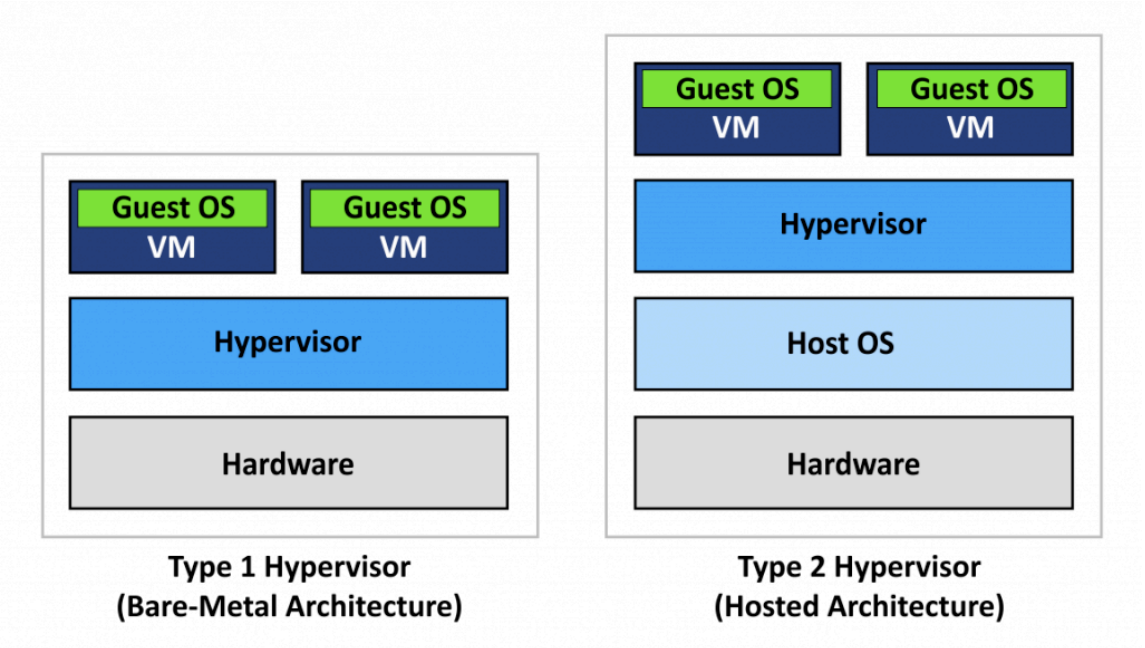
Docker 101

Sudharshan Govindan
Developer Advocate

sudharshan.govindan@in.ibm.com
@sudhargovindan

What is a Virtual Machine (VM)?

VMs were developed in 1974 by Popek and Goldberg as a set of conditions to support system virtualization. System virtual machines virtualize the hardware resources, including processor, memory, storage and peripheral devices. The Virtual Machine Monitor (VMM or Hypervisor) abstracts the virtual machine. You can run a native Hypervisor directly on the hardware or a hosted Hypervisor on a host Operating System (OS).



What is a Container?

A container creates "VM-like" isolation but managed at the **process level.**

Three Linux kernel components are used to isolate applications running in containers and limit access to resources on the host:

- **Linux namespaces**
- **Control groups (cgroups),** control the limits and monitoring of resources. The `ns` subsystem in cgroups integrates namespaces and control groups.
- **Seccomp**, developer in 2005, introduced in 2014, limits how processes use system calls, by defining a security profile for processes, whitelisting system calls, parameters and file description allowed to use.
- **SELinux contexts**, Secured Enhanced Linux, processes run as confined SELinux type that has limited access to host system resources, used to protect processes from each other and protect host from its running processes.

achieved by a set of "**namespaces**" (isolated view):

- PID –isolated view of process IDs
- USER- user and group IDs
- UTS - hostname and domain name
- NS - mount points
- NET - Network devices, stacks, ports
- IPC - inter-process communications, message queues

The key statement: **A container is a process(es) running in isolation**

What is an Image?

An image is a file-system bundle that contains all dependencies required to execute a process: files in the file system, installed packages, available resources, running processes and kernel modules.

VM vs Container

Before containers in an OS environment:

- Entanglement between OS and App
- Entanglement between runtime environment and OS
- Version dependency and conflicts between Apps
- Breaking updates
- Updates require full app stop and downtime
- Deployment and Maintenance of HA system is complex
- Slow startup time

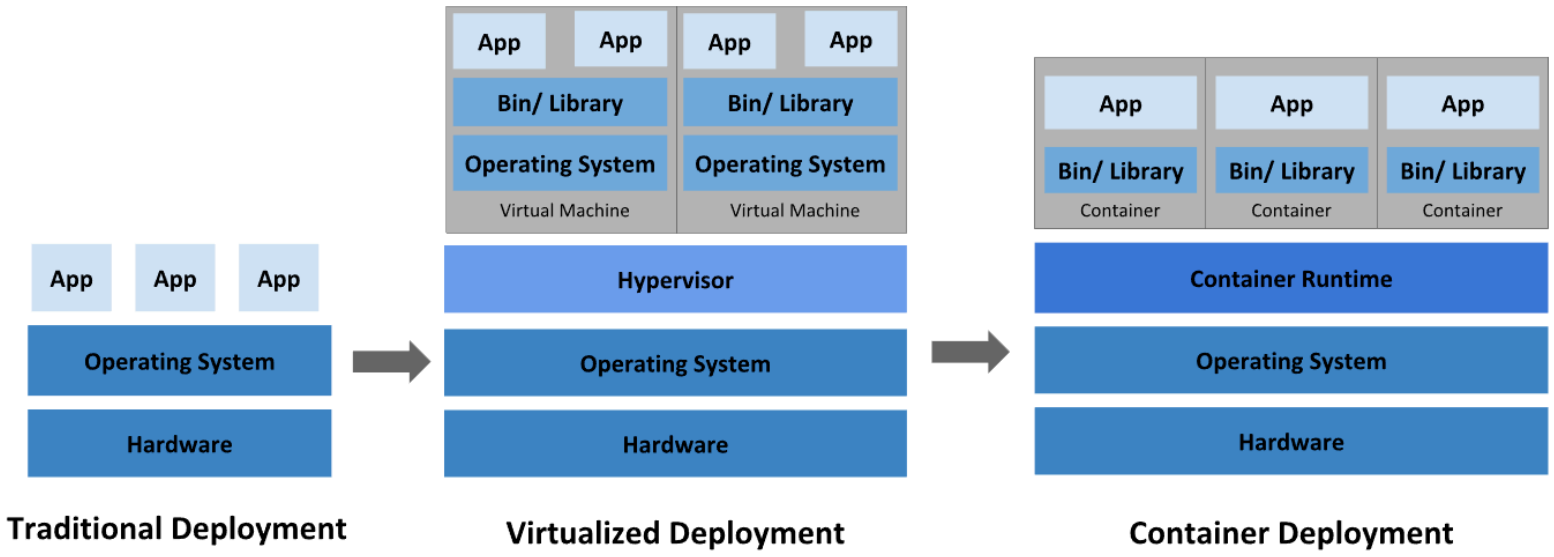
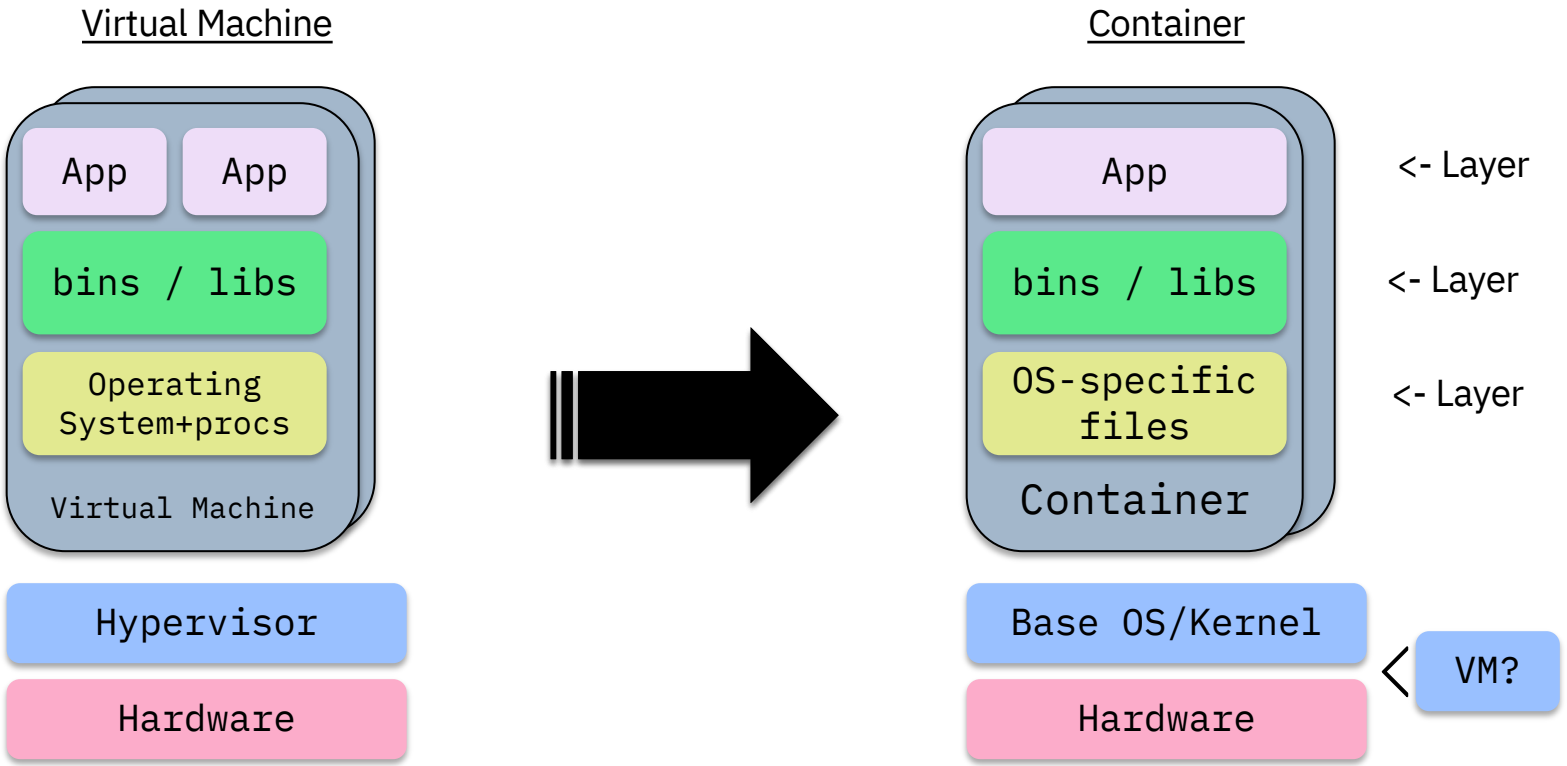


image source: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

VM vs Container: Notice the layers!



Each VM has its own OS

Containers share the same base Kernel

VMs, Containers and Container Orchestration

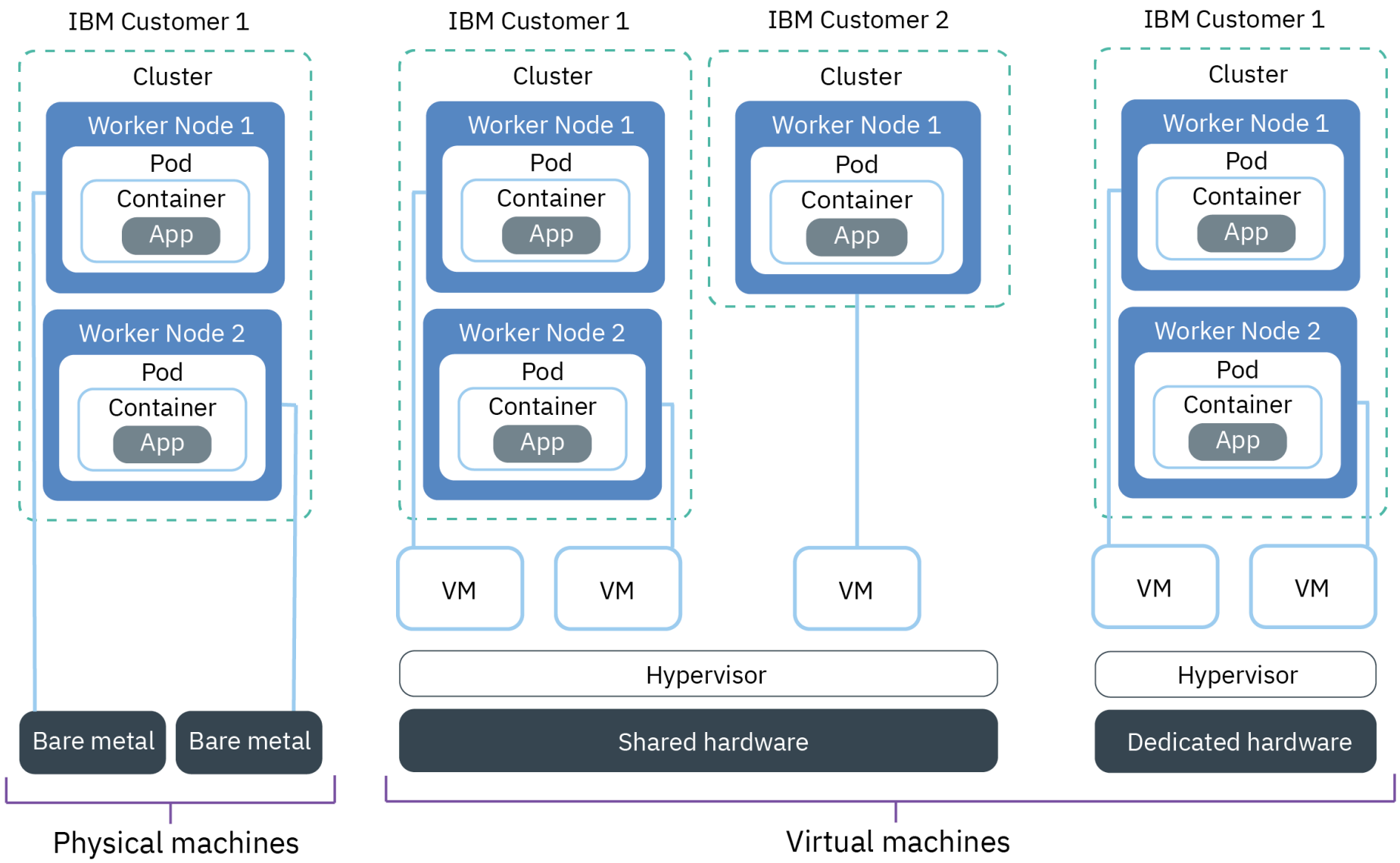
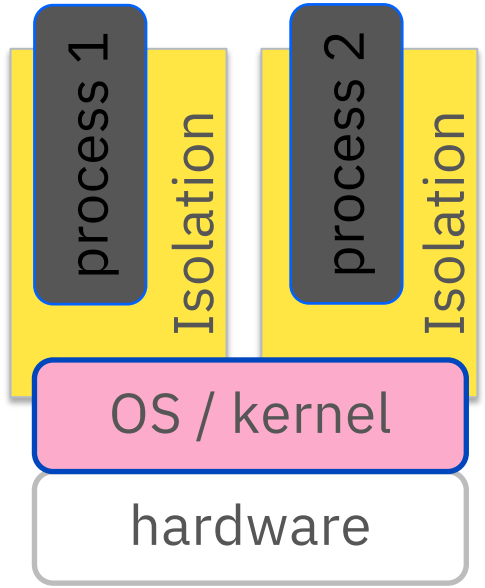


image source: https://cloud.ibm.com/docs/containers?topic=containers-planning_worker_nodes

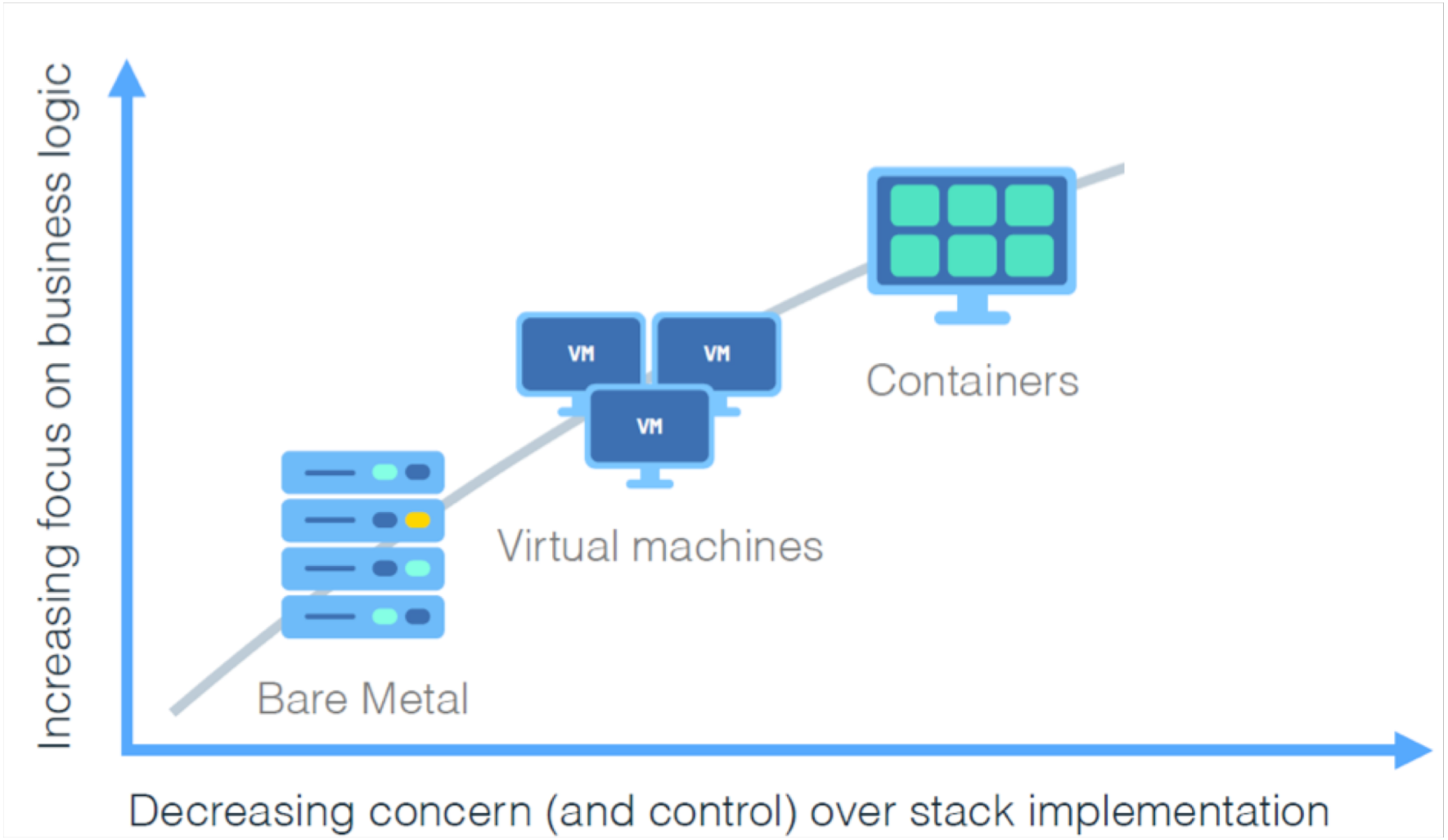
A Short History of Containers

Containers – not a new idea

- (1979) `chroot` command, changes apparent root for running process to isolate system processes
- (1982) `chroot` added to Unix v7
- (1990s) `jail` command created
- (2000) `jail` added to FreeBSD to isolate filesystems, users, networks etc.
- (2001) Linux VServer is a jail mechanism that partitions resources (file systems, network addresses, memory)
- (2004) Solaris Containers using Solaris Zones, application has a full user, processes, and file system, and access to the system hardware, but can only see within its own zone.
- (2005) Open VZ (Virtuzzo), OS-level virtualization for Linux using a patched Linux kernel for virtualization, isolation, resource management and checkpointing
- (2006) Google launches `Process Containers` for limiting resource usage, renamed `cgroups` in 2007
- (2008) `cgroups` merged into Linux kernel 2.6.24, becoming LinuX Containers (LXC)
- (2009) Cloud Foundry developed by VMWare called Project B29 (Warden, Garden)
- (2011) Cloud Foundry started Warden
- (2013) Docker released as open source
- (2014) LXC replaced by `libcontainer`, Google contributes container stack `LMCTFY` (Let Me Contain That For You) to Docker project as `libcontainer` in 2015



Evolution Of Containers



The challenge

Multiplicity of Stacks

Static website:
- Nginx
- OpenSSL
- Bootstrap 2
- ModSecurity

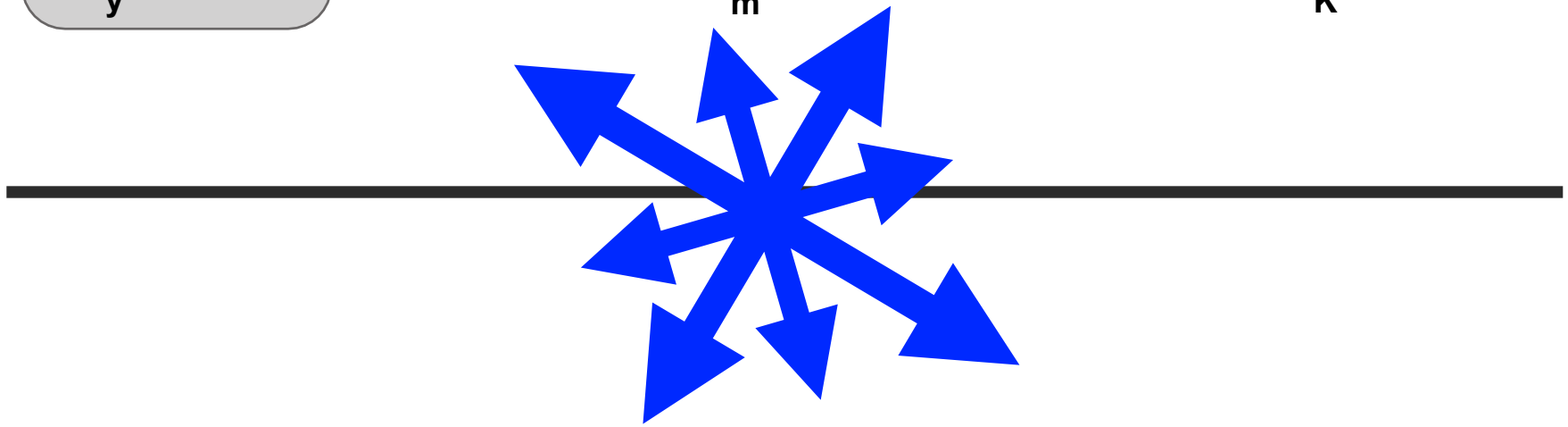
User DB:
- PostgreSQL
- pgv8
- v8

Web front end:
- Ruby
- Rails
- Sass
- Unicorn

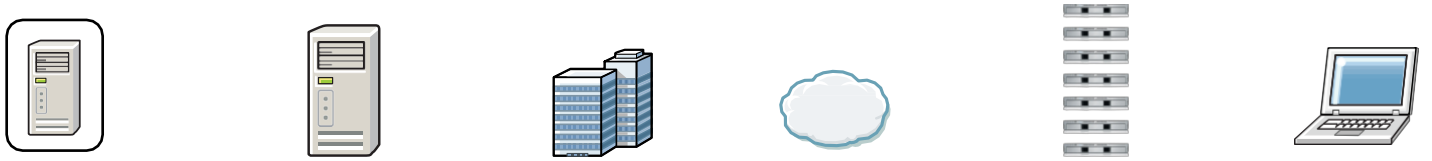
Queue:
- Redis
- Redis-sentinel

Analytics DB:
- Hadoop
- Hive
- Thrift
- OpenJDK

Do services and apps interact appropriately?



Multiplicity of hardware environments



Can I migrate smoothly and quickly?

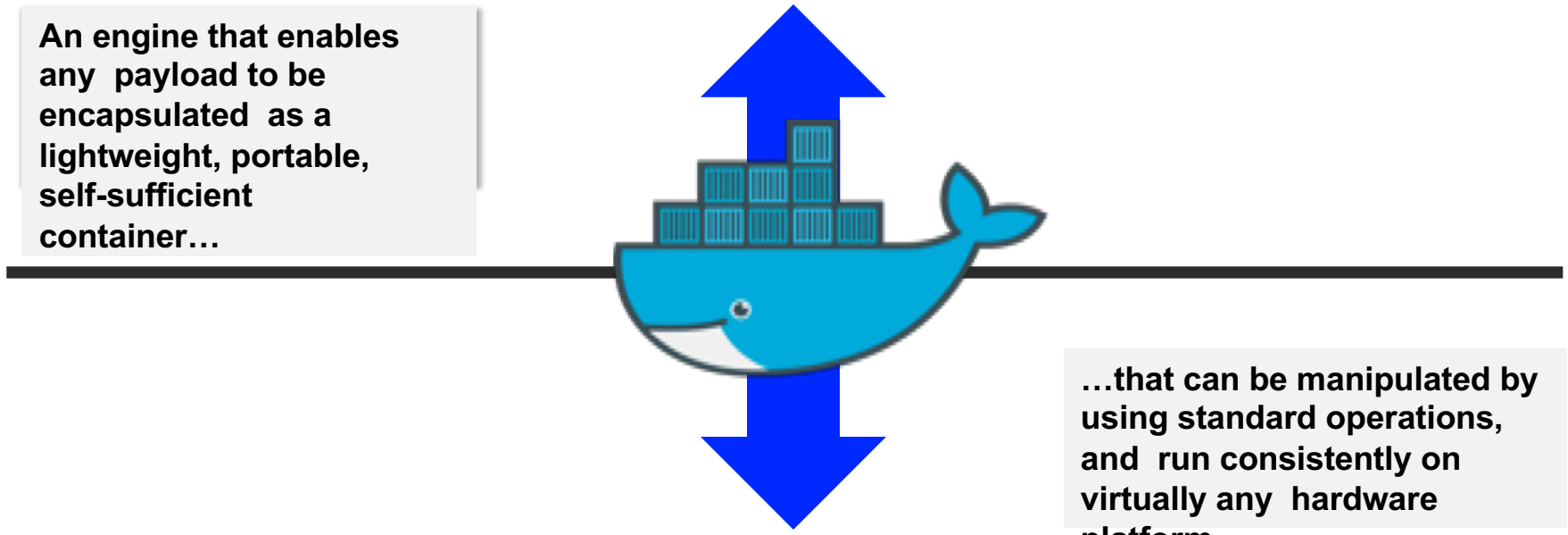
Docker: A shipping container for code

Multiplicity of Stacks

Static website | User DB | Web front end | Queue | Analytics DB

Do services and apps interact appropriately?

An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container…



…that can be manipulated by using standard operations, and run consistently on virtually any hardware platform.

Multiplicity of hardware environments

Can I migrate smoothly and quickly?


Developme nt VM


QA server


Customer Data Center


Public Cloud


Productio n Cluster


Contributor s laptop

Why Containers?

Fast startup time - only takes milliseconds to:

- Create a new directory
- Lay-down the container's filesystem
- Setup the networks, mounts, ...
- Start the process

Better resource utilization

- Can fit far more containers

	CONTAINER BENEFITS	VIRTUAL MACHINE BENEFITS
Consistent Runtime Environment	✓	✓
Application Sandboxing	✓	✓
Small Size on Disk	✓	
Low Overhead	✓	

Advantages of Containers

- **Containers are portable**
  - Any platform with a container engine can run containers
- **Containers are easy to manage**
  - Container images are easy to share, download, and delete, especially with Docker registries
  - Container instances are easy to create and delete
  - Each container instance is easy and fast to start and stop
- **Containers provide "just enough" isolation**
  - More lightweight than virtual machines
  - Processes share the operating system kernel but are segregated
- **Containers use hardware more efficiently**
  - Greater density than virtual machines
    - Especially Docker containers, which can share layers
- **Containers are immutable**
  - Container images are versions
  - Containers cannot (should not) be patched

Containers vs Docker

Containers is the technology, Docker is the **tooling** around containers

Without Docker, containers would be **hard to use** (for most people)

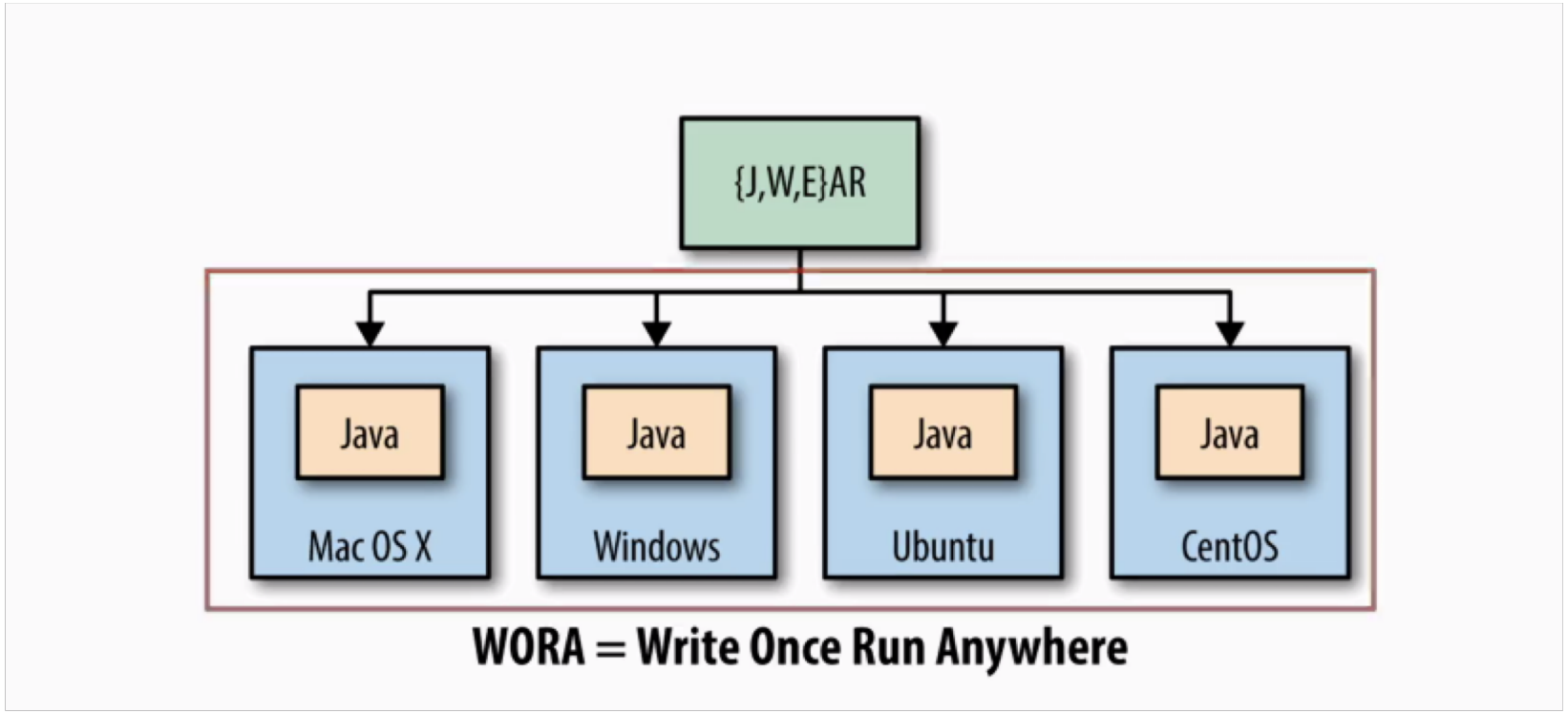
Docker **simplified** container technology

Added value: Lifecycle support, setup file system, etc

For extra confusion: **Docker Inc.** is a company, which is different than Docker the technology…

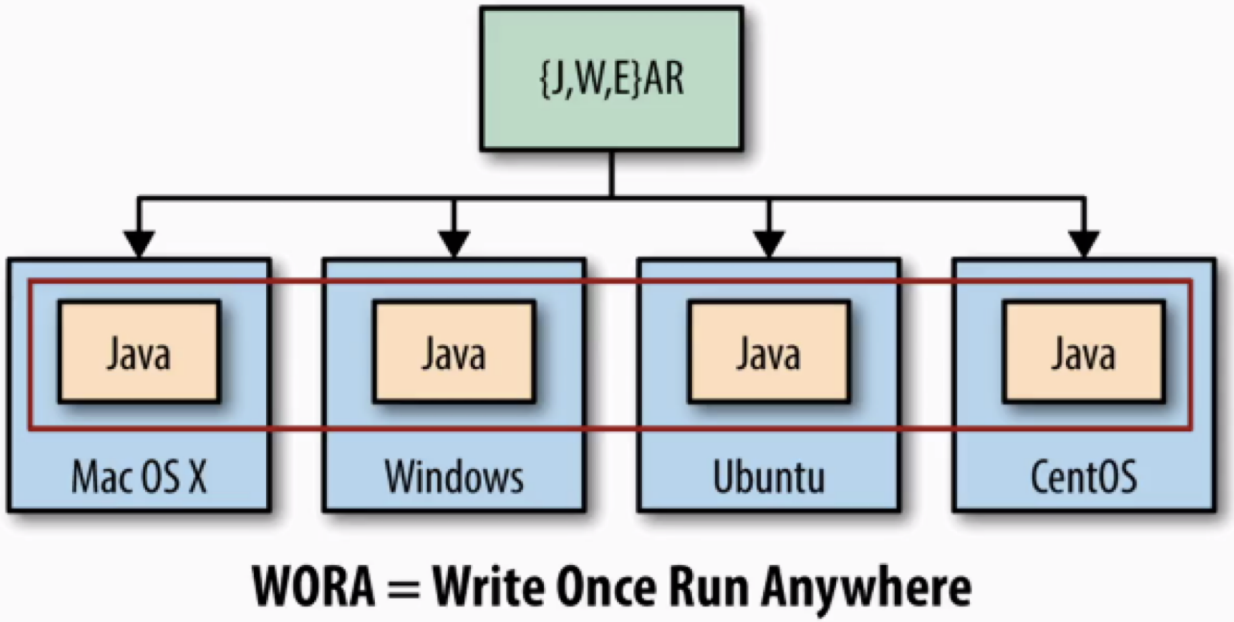
Docker’s ecosystem approach transformed the perception of containers
- Building application-centric containers
- Mechanism for sharing images (Docker Registry)
- Open-source enabled

Java Application



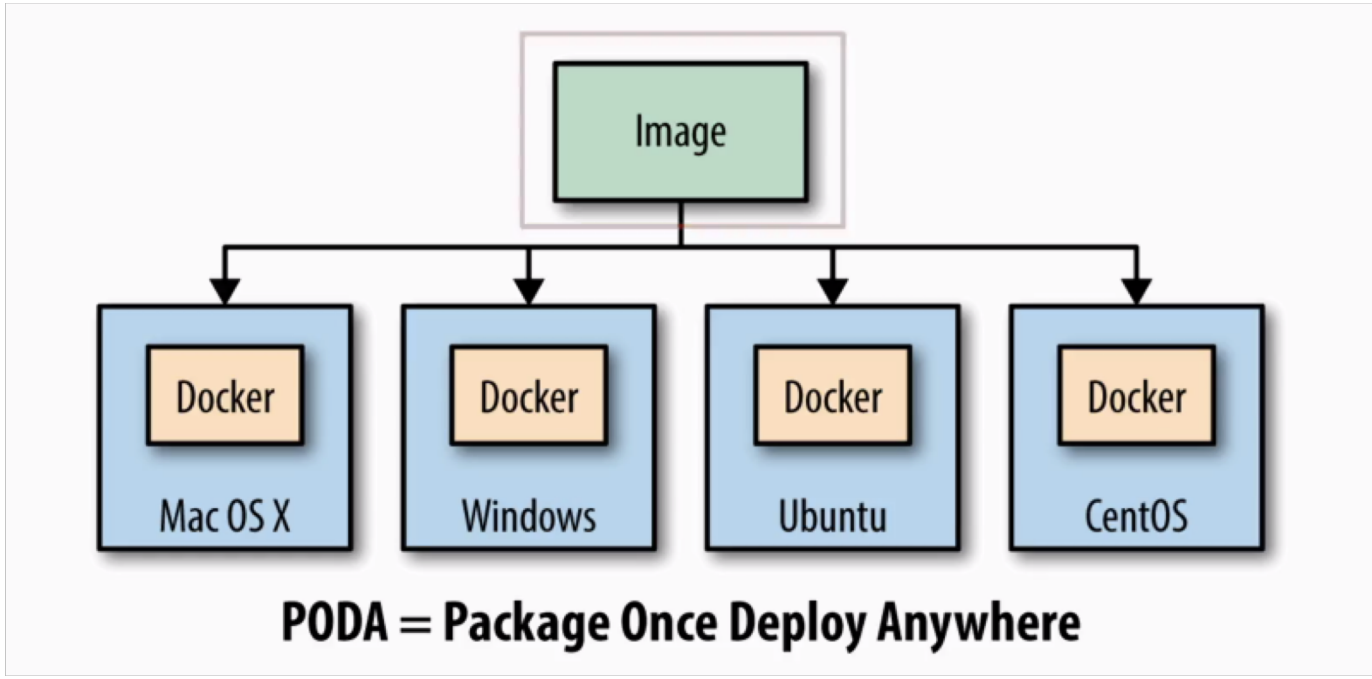
- Each OS has its Java Software
- With same version of Software

Dependencies



Java installed
(Proper version)

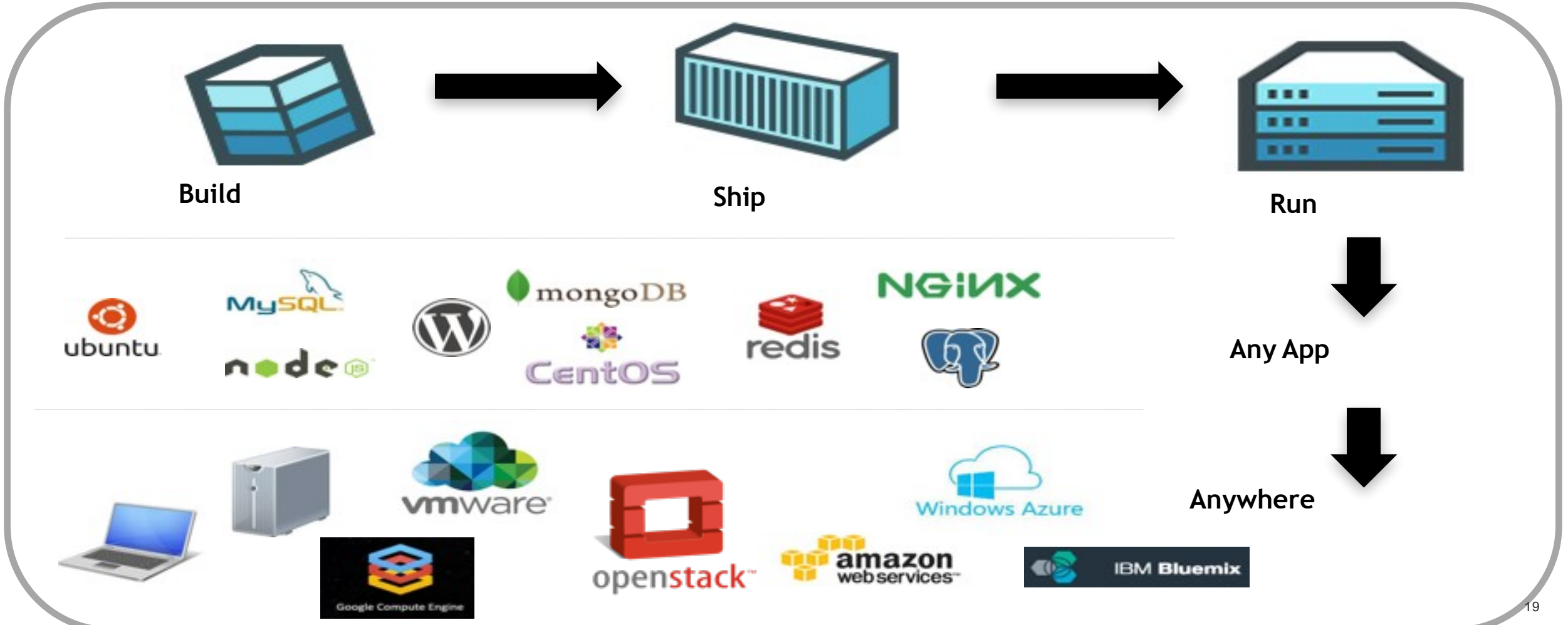
One Step Ahead : Docker

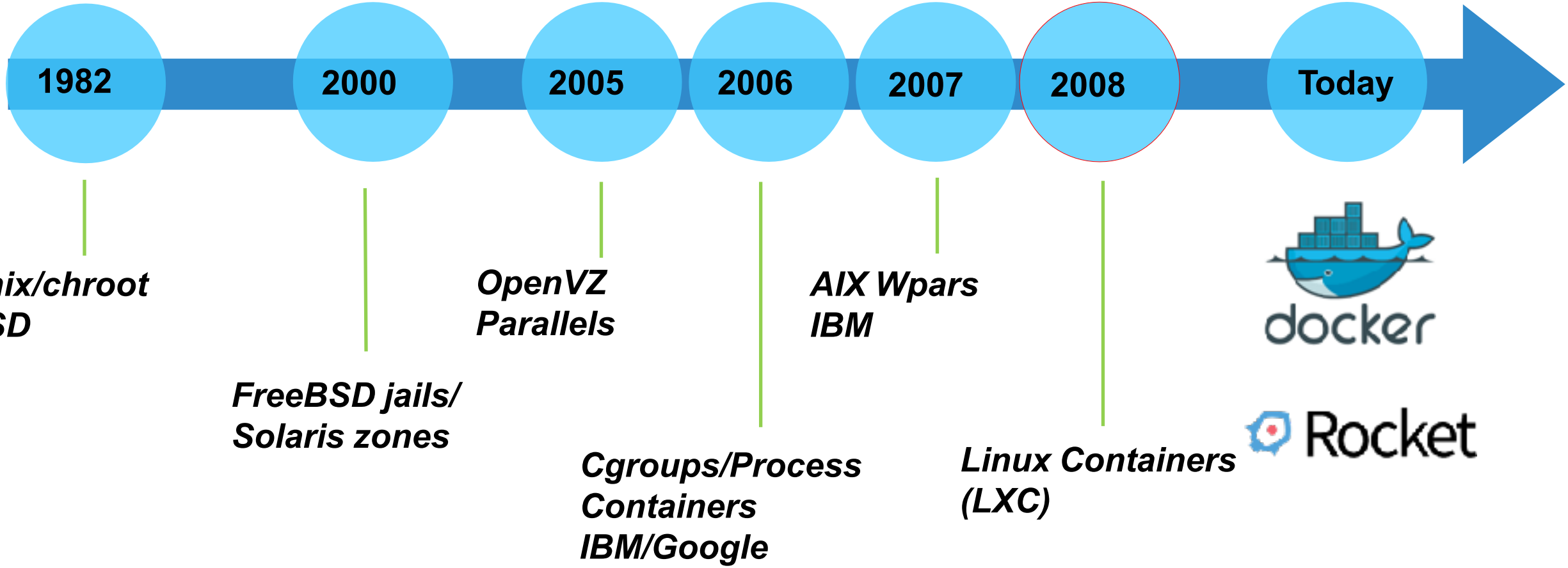


- No installation of software
- Docker Image using Docker Platform
- One step ahead

- Open software
  - Launched March 2013
  - 2.0+ billion downloads of Docker images
- Open contribution
  - 2000+ contributors
  - #2 most popular project
  - 185 community meet-up groups in 58 countries
- Open design
  - Contributors include IBM, Red Hat, Google, Microsoft, VMware, AWS, Rackspace, and others
- Open governance
  - 12 member governance advisory board selected by the community

Docker is an open platform for building distributed applications for developers and system administrators




1982
2000
2005
2006
2007
2008
Today
Unix/chroot
BSD
FreeBSD jails/
Solaris zones
OpenVZ
Parallels
Cgroups/Process
Containers
IBM/Google
AIX Wpars
IBM
Linux Containers
(LXC)
docker
Rocket

What is a Container Engine?

The container runtime packages, instantiates, and runs containers. There are many container engines and many cloud providers have their own container engines to consume OCI compliant Container images. Having a standard Container Image Format allows interoperability between all these platforms.

Typically, the container engine is responsible:

- Handling user input
- Handling input over an API often from a Container Orchestrator
- Pulling the Container Images from the Registry Server
- Decompressing and expanding the container image on disk
- Preparing a container mount point
- Preparing metadata for container runtime to start container correctly (defaults from container image ex. Arch86, user input e.g. CMD or ENTRYPOINT)
- Calling the Container Runtime

The Container Runtime is the lower-level component used in Container Engine. The OCI Runtime Standard reference implementation is `runc`. When Docker was first created it relied on LXC as the container runtime, later replaced by `libcontainer` to start containers. When OCI was created, Docker donated `libcontainer` to OCI and it became `runc`. At the lowest level this provides a consistent way to start containers, no matter the container engine.

Container Runtime

In December 2014, CoreOS (now RedHat) released “rkt” as an alternative to Docker and initiated *app container* (appc) and *application container image* (ACI) as independent committee-steered specifications, which developed into the Open Container Initiative (OCI). As the number of runtime providers increases and varies between providers, for End of Vendor Support (EOVS) or End of Life (EOL) reasons or with increase of hybrid cloud architectures, interoperability and inter-exchangeability is critical.

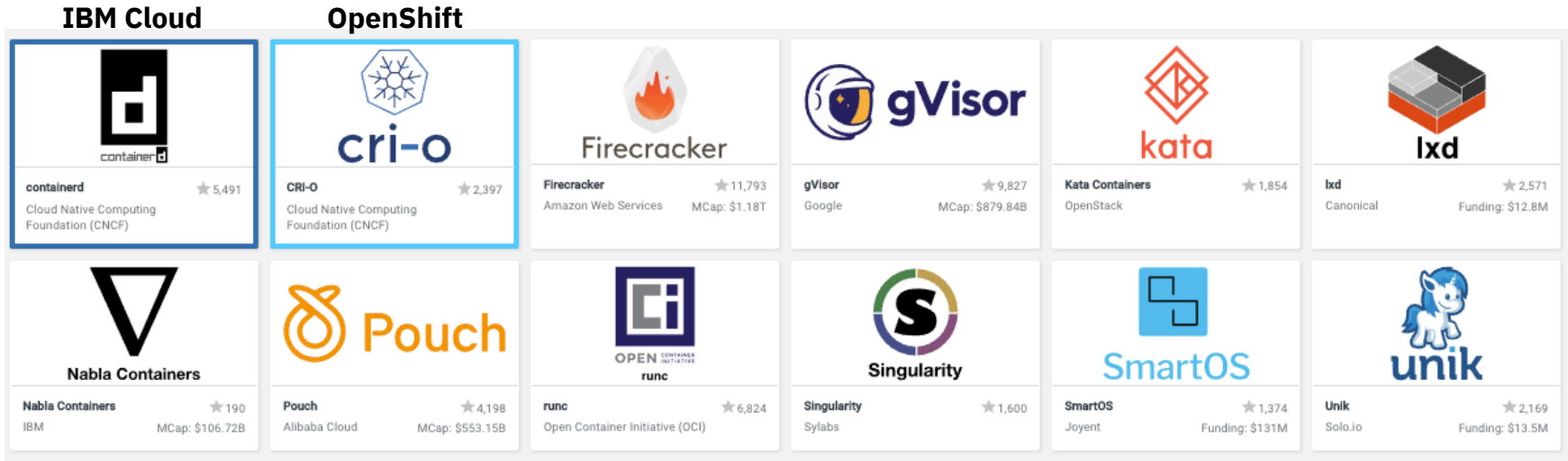
OPEN CONTAINER INITIATIVE

CLOUD NATIVE COMPUTING FOUNDATION

THE LINUX FOUNDATION

On June 22, 2015, the Open Container Initiative (OCI) was announced, formed under the Linux Foundation and launched by Docker, CoreOS and others. The OCI currently contains a Runtime Specification (runtime-spec) and an Image Specification (image-spec).

docker



Our First Container

$ docker run ubuntu echo Hello World

Hello World

What happened?

- Docker checked local repository for an image named `ubuntu`
- Docker downloaded an image `ubuntu` from https://registry.hub.docker.com/_/ubuntu
- Docker created a directory with an "ubuntu" filesystem (image)
- Docker created a new set of namespaces (lsns)
- Ran a new process: echo Hello World
  - Using the namespaces to isolate it from other processes
  - Using the new directory as the "root" of the filesystem (chroot)
- That's it!
  - Notice as a user I never installed "ubuntu"
- Run it again - notice how quickly it ran

```
$ ls /var/lib/docker/overlay2
backingFsBlockDev  l
```

```
$ docker run ubuntu echo Hello World
Unable to find image 'ubuntu:latest' locally
latest: Pulling from library/ubuntu
692c352adcf2: Pull complete
97058a342707: Pull complete
2821b8e766f4: Pull complete
4e643cc37772: Pull complete
Digest: sha256:55cd38b70425947db71112eb5dddfa3aa3e3ce307754a3df2269069c
Status: Downloaded newer image for ubuntu:latest
Hello World
[node1] (local) root@192.168.0.28 ~
$ ls /var/lib/docker/overlay2
499feefd10023e41919223712408e523a43c9784744082f101f65d5b6af0040c
4d77b1376351345961554b4d6638f56981930abfe4b2baf1100805b04c631fc3
4d77b1376351345961554b4d6638f56981930abfe4b2baf1100805b04c631fc3-init
91b44dc7c37872bc24bc4b0624949a08f6744ddb4dc9b04672afdfff201842ea
backingFsBlockDev
ddd1459cf51c30fc116a6d60bea2923812ede2129817be3718bf62137642a521
fe18976d26725e688dd41a5038677ea1dd487242d9cbef7edd11532faac03d94
```

https://github.com/tianon/docker-brew-ubuntu-core/blob/5510699c08fd4c68c7fd05783ceedf15751b6cd9/bionic/Dockerfile

```
FROM scratch
ADD ubuntu-bionic-core-cloudimg-amd64-root.tar.gz /
RUN [ -z "$(apt-get indextargets)" ]
RUN set -xe \
  && ...
RUN mkdir -p /run/systemd && echo 'docker' > /run/systemd/container
CMD ["/bin/bash"]
```

"ssh-ing" into a container

```
$ docker run -ti ubuntu bash
root@62deec4411da:/# pwd
/
root@62deec4411da:/# exit
$
```

- Now the process is "bash" instead of "echo"
- But its still just a process
- Look around, mess around, its totally isolated
  - rm /etc/passwd – no worries!
  - MAKE SURE YOU'RE IN A CONTAINER!

A look under the covers

```
$ docker run ubuntu ps -ef
UID          PID    PPID  C STIME TTY          TIME CMD
root           1       0  0 14:33 ?        00:00:00 ps -ef
```

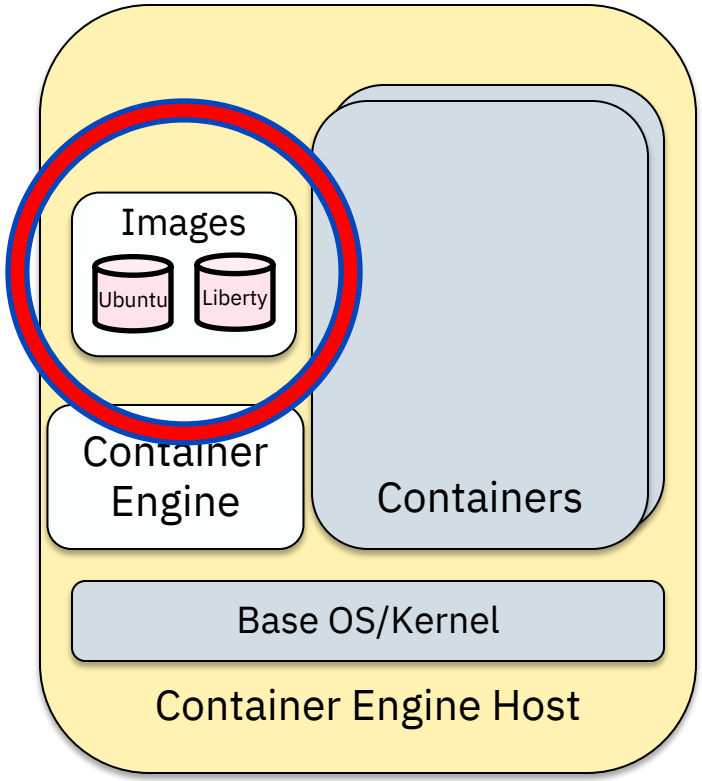
Things to notice with these examples:

- Each container only sees its own process(es)
- Each container only sees its own filesystem
- Running as "root"
- Running as PID 1

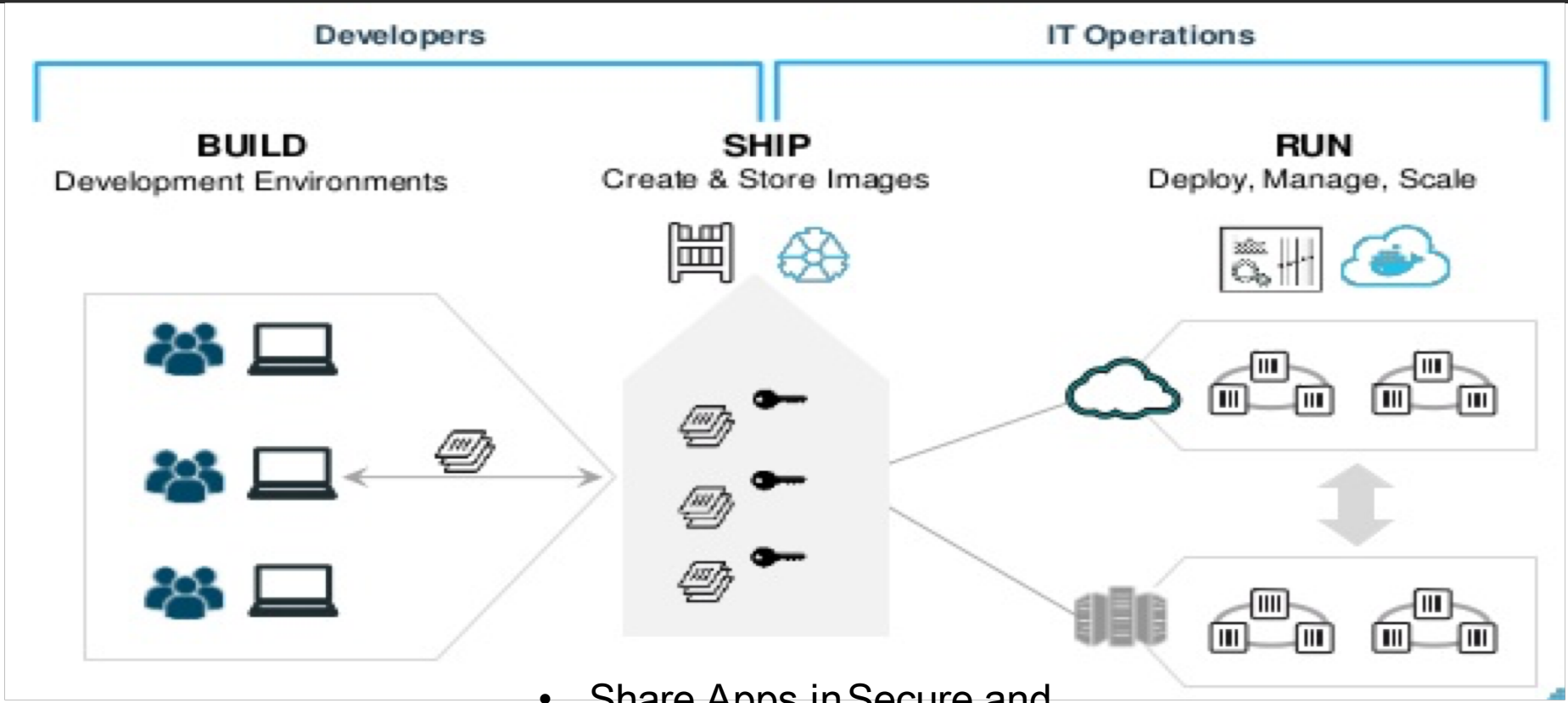
Container Images

- Tar file containing a container's filesystem + metadata

- For sharing and redistribution
  - Global/public registry for sharing



Phases : Build , Ship, Run

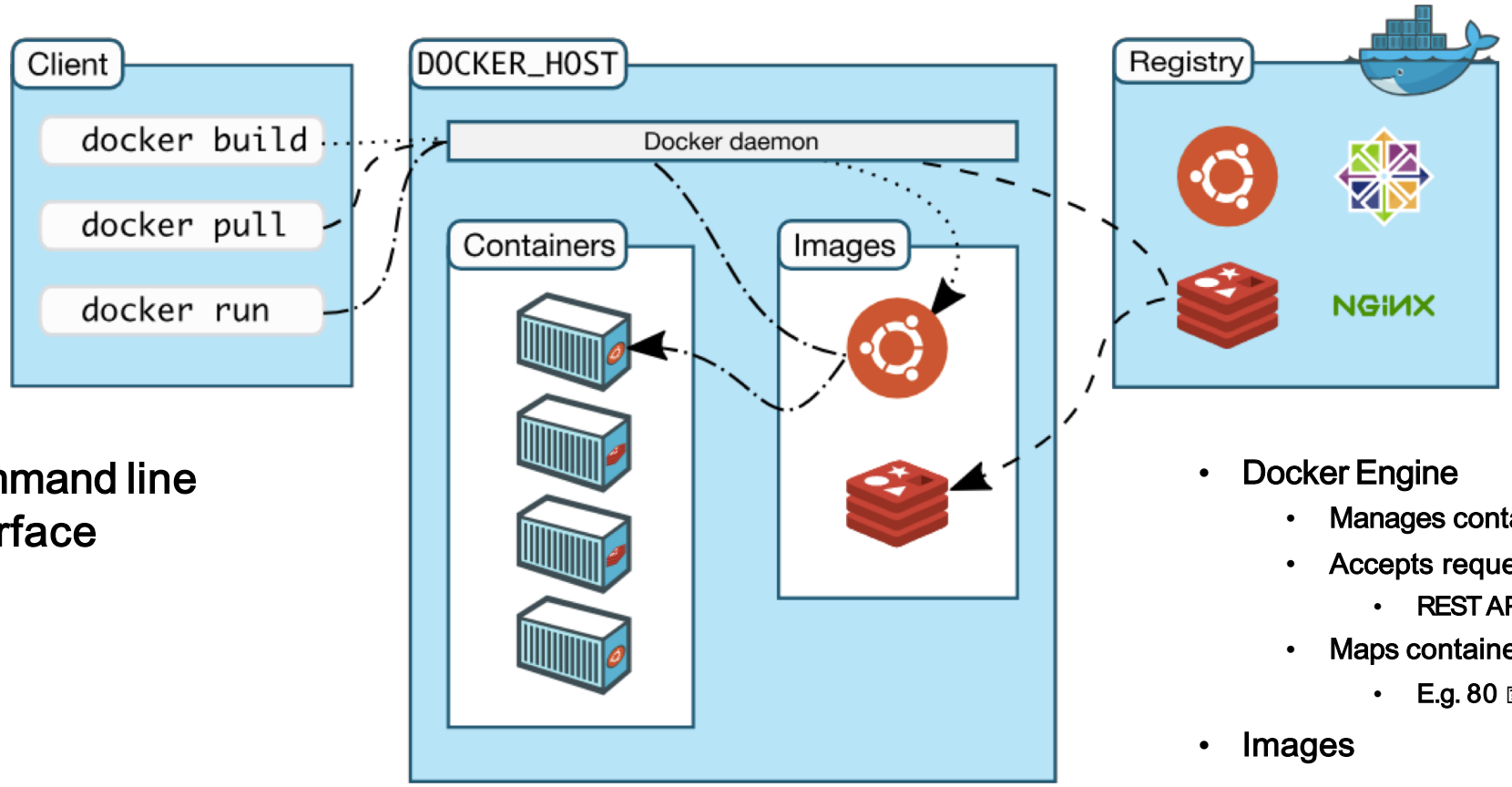


- Tools to create containerized application
- Package App – dependencies, infra as template – This is called Image

- Share Apps in Secure and collaboration manner
- Docker images stored, shared & managed in Docker Registry
- Docker Hub(Public) default registry for all images

- Deploy, manage and scale apps
- Container is runtime representation of image
- Lifecycle of container – run, start, scale, stop, move, delete

Docker Architecture



Command line Interface

Docker Engine

Images

- Docker Engine
  - Manages containers on a host
  - Accepts requests from clients
    - REST API
  - Maps container ports to host ports
    - E.g. 80 → 3582
- Images
- Docker Client
  - Drives daemon
  - Drives "builder" of Images
- Docker Registry
  - Image DB

Docker Registry

- DockerHub (https://hub.docker.com)
- Public registry of Docker Images
- The central place for sharing images with friends or coworkers!
- Also useful to find prebuilt images for web servers, databases, etc
- Enterprises will want to find a private registry to use (such as Artifactory)

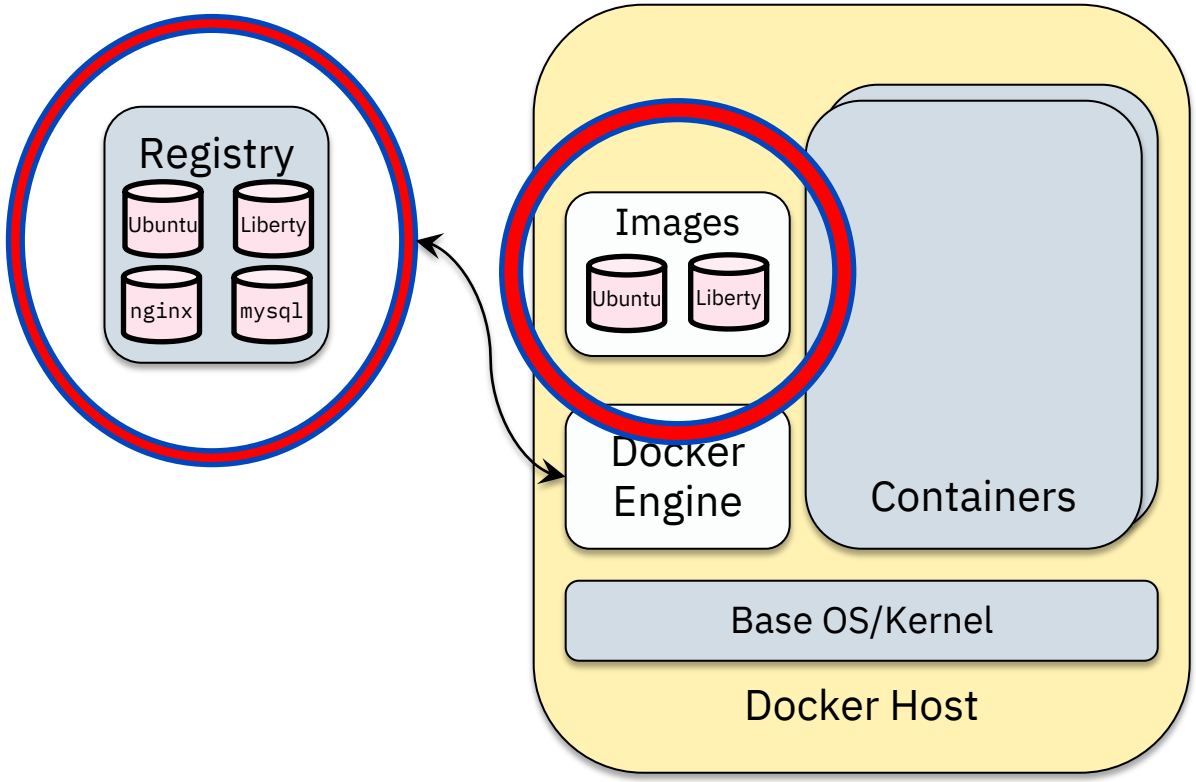


Image Registries

Red Hat container images provide the following benefits:

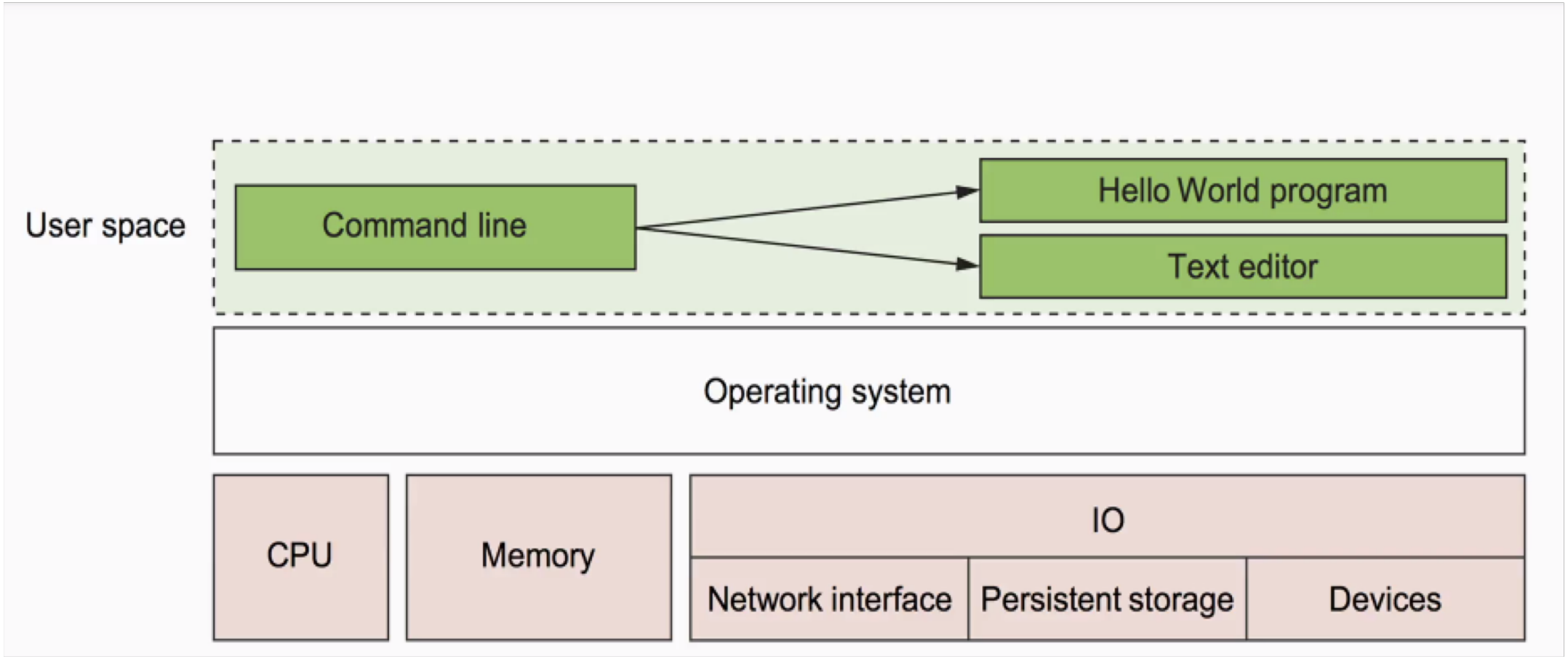
- *Trusted source*: All container images comprise sources known and trusted by Red Hat.
- *Original dependencies*: None of the container packages have been tampered with, and only include known libraries.
- *Vulnerability-free*: Container images are free of known vulnerabilities in the platform components or layers.
- *Runtime protection*: All applications in container images run as non-root users, minimizing the exposure surface to malicious or faulty applications.
- *Red Hat Enterprise Linux (RHEL) compatible*: Container images are compatible with all RHEL platforms, from bare metal to cloud.
- *Red Hat support*: Red Hat commercially supports the complete stack.

Container registries:

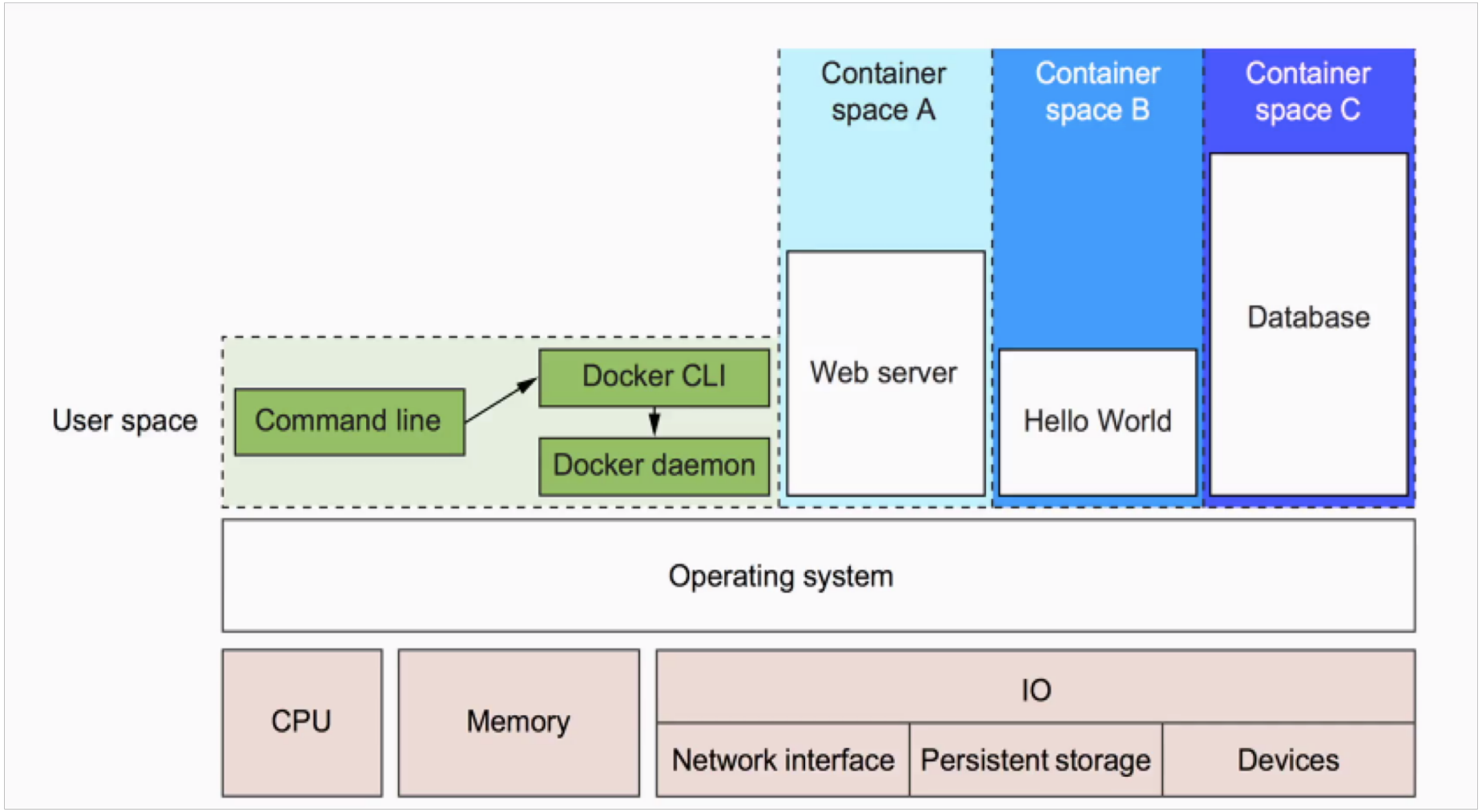
- https://hub.docker.com
- https://access.redhat.com/containers/
- https://quay.io (includes server-side image building, fine-grained access controls, and automatic scanning of images for known vulnerabilities)
- JFrog Artifactory
- Sonatype Nexus
- Bitnami

Many registries conform to the Docker Registry HTTP API v2 specification.

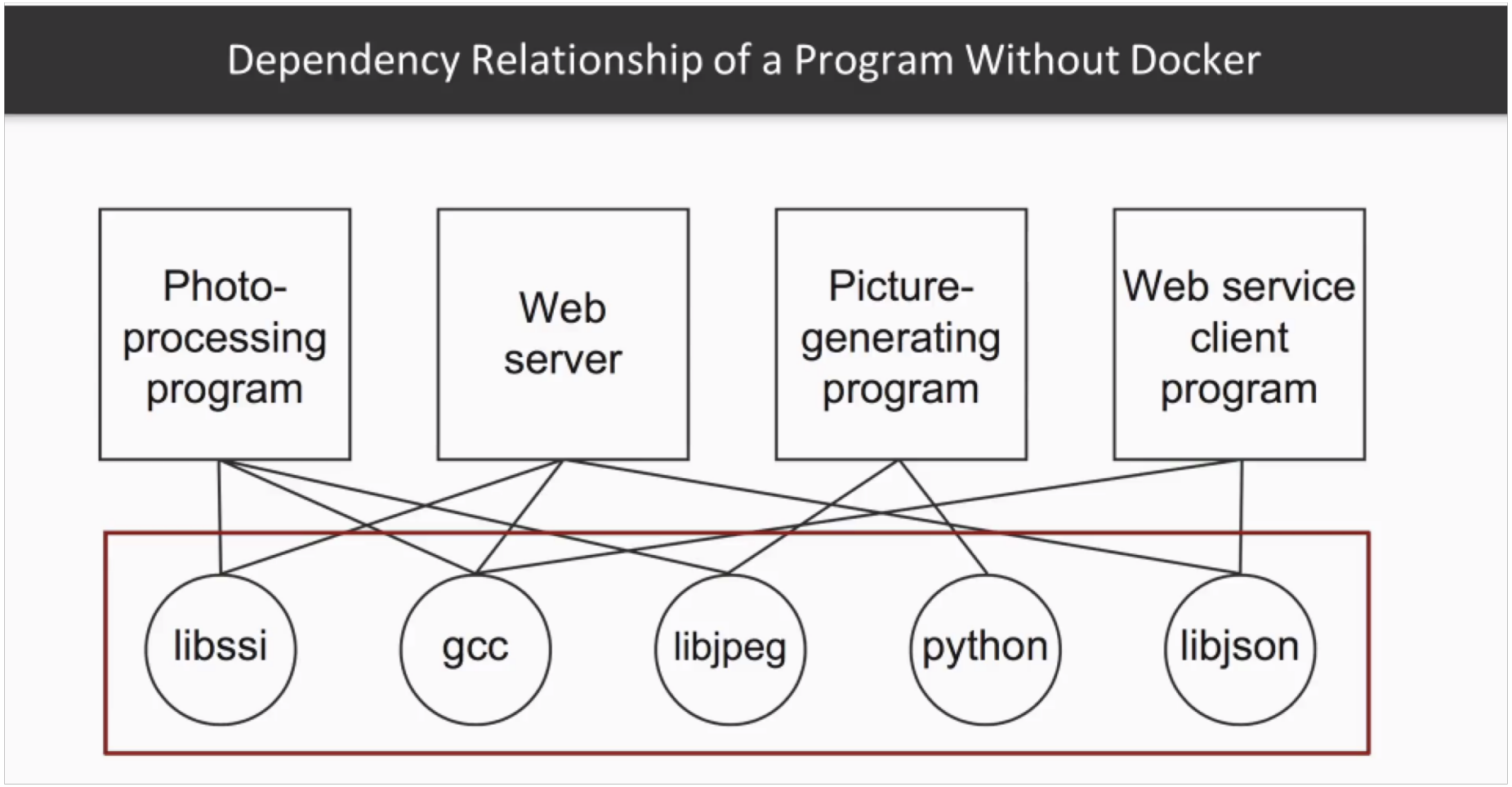
Isolation : Linux



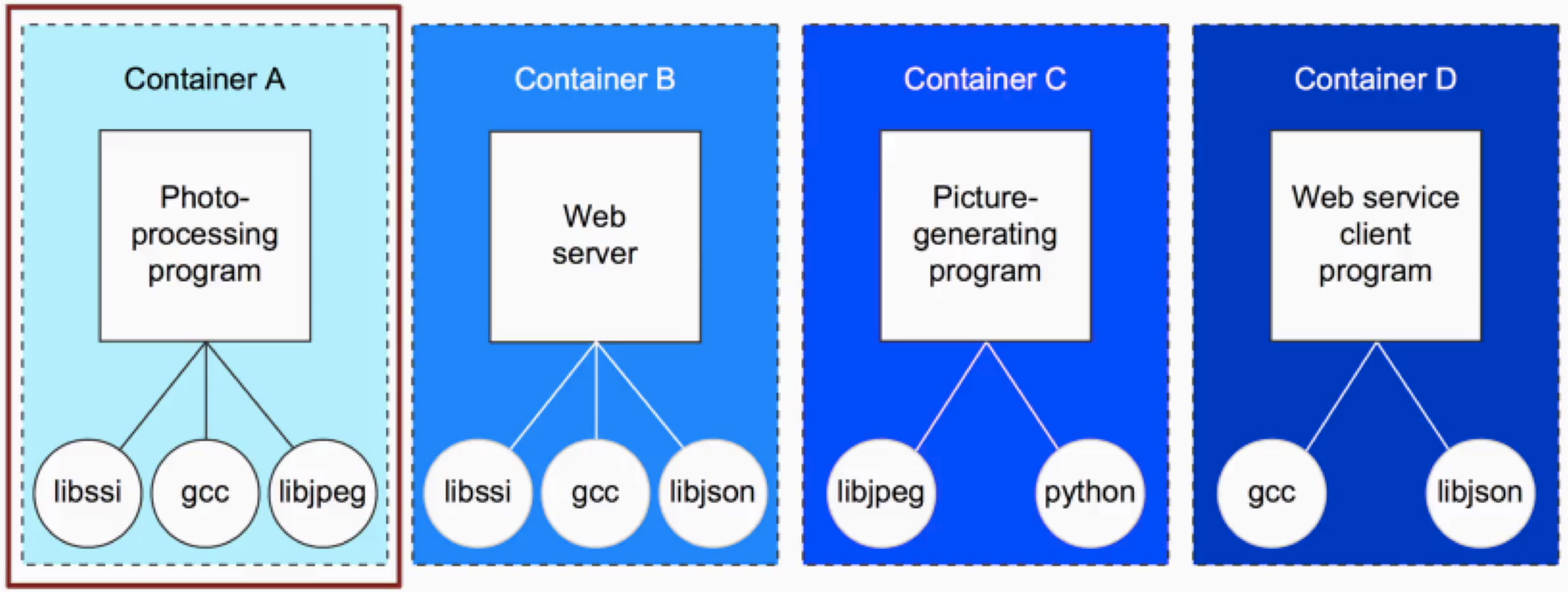
Docker Isolation：Process Level



Running programs without docker



Each App : With needed Dependencies



So : What is Docker?

1. At its core, Docker is tooling to manage containers

– Docker is not a technology, it's a tool or platform

– Simplified existing technology to enable it for the masses

2. Tooling to manage containers

– Containers are not new

– Docker just made them easy to use

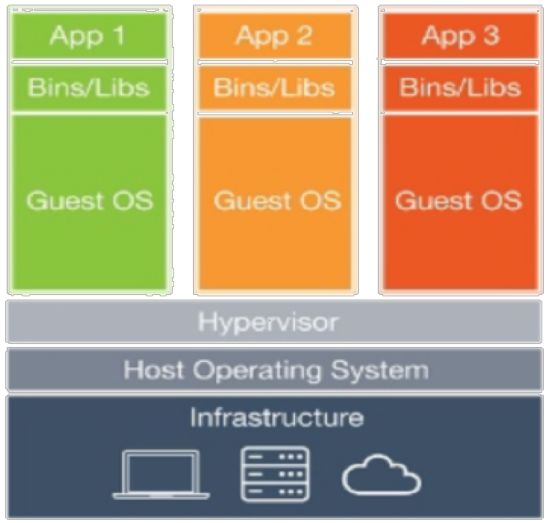
3. Docker creates and manages the lifecycle of containers

– Setup filesystem

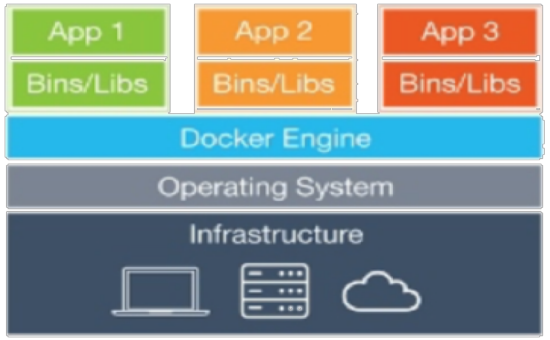
– CRUD container

  - Setup networks
  - Setup volumes / mounts
  - Create: start new process telling OS to run it in isolation

Docker Container Vs VMWare

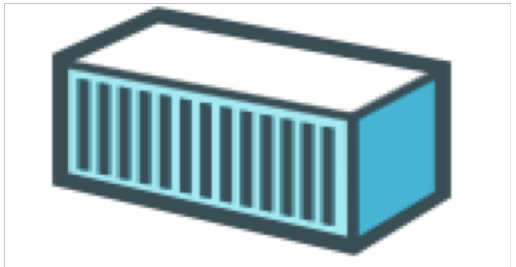


vmware

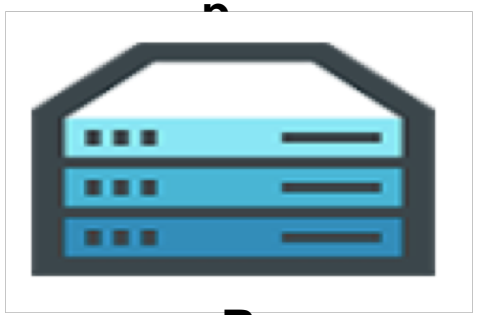


docker

Buil
d

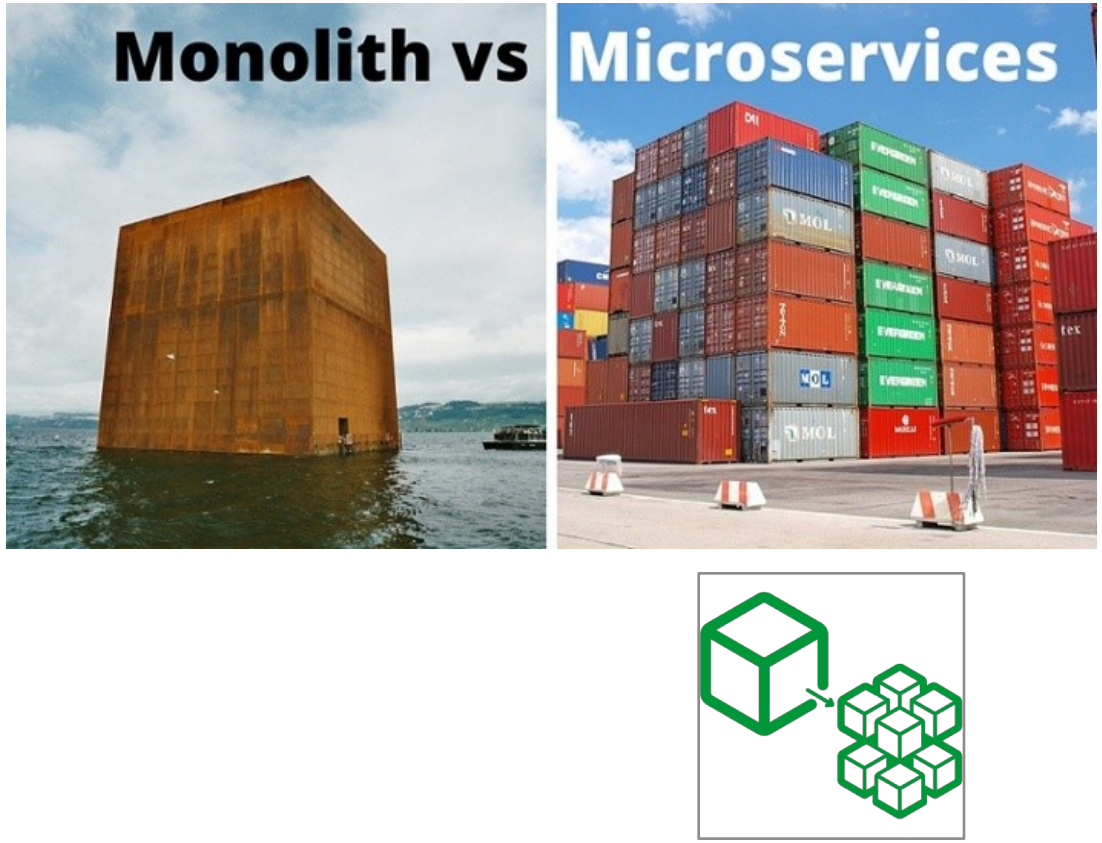


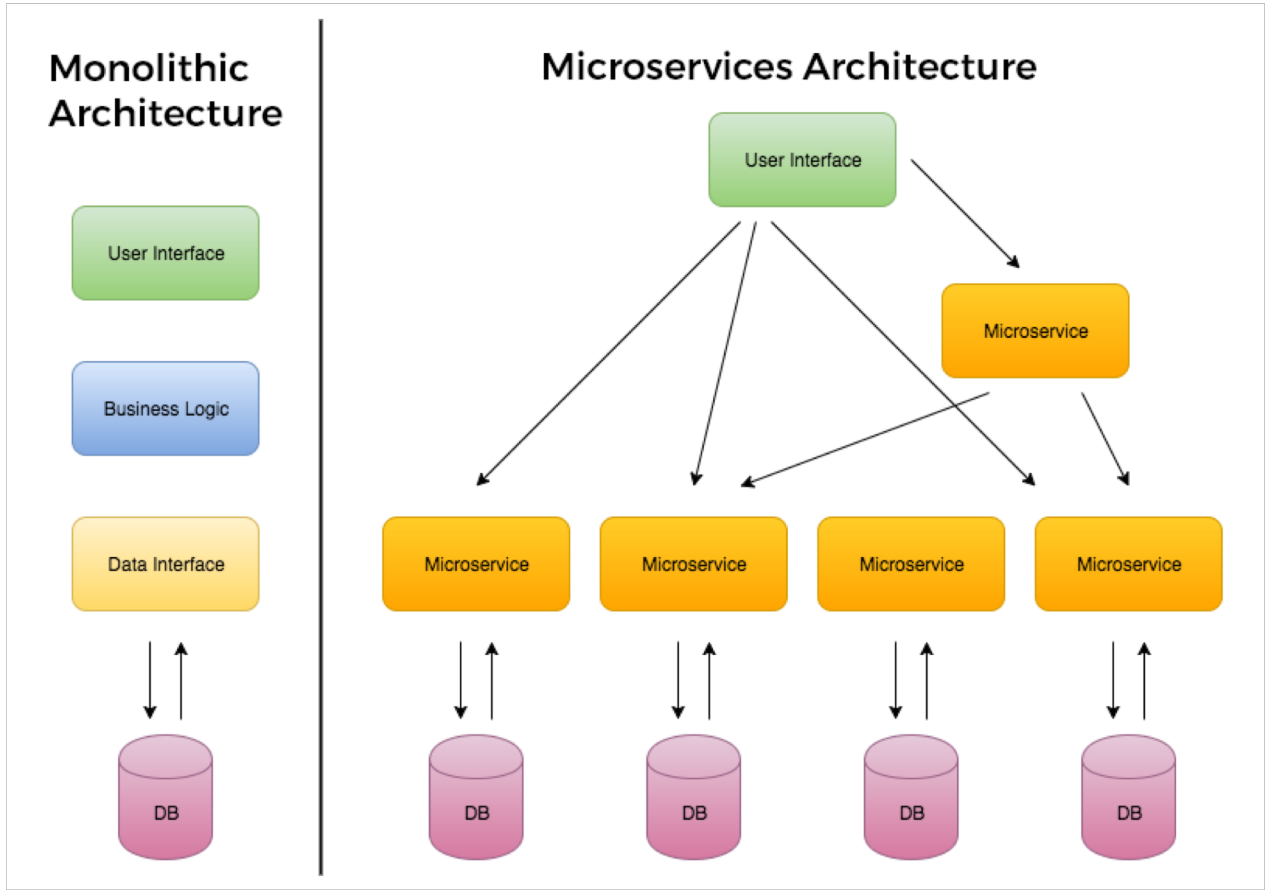
Shi
n



Ru
n

Microservices & Cloud Native Apps



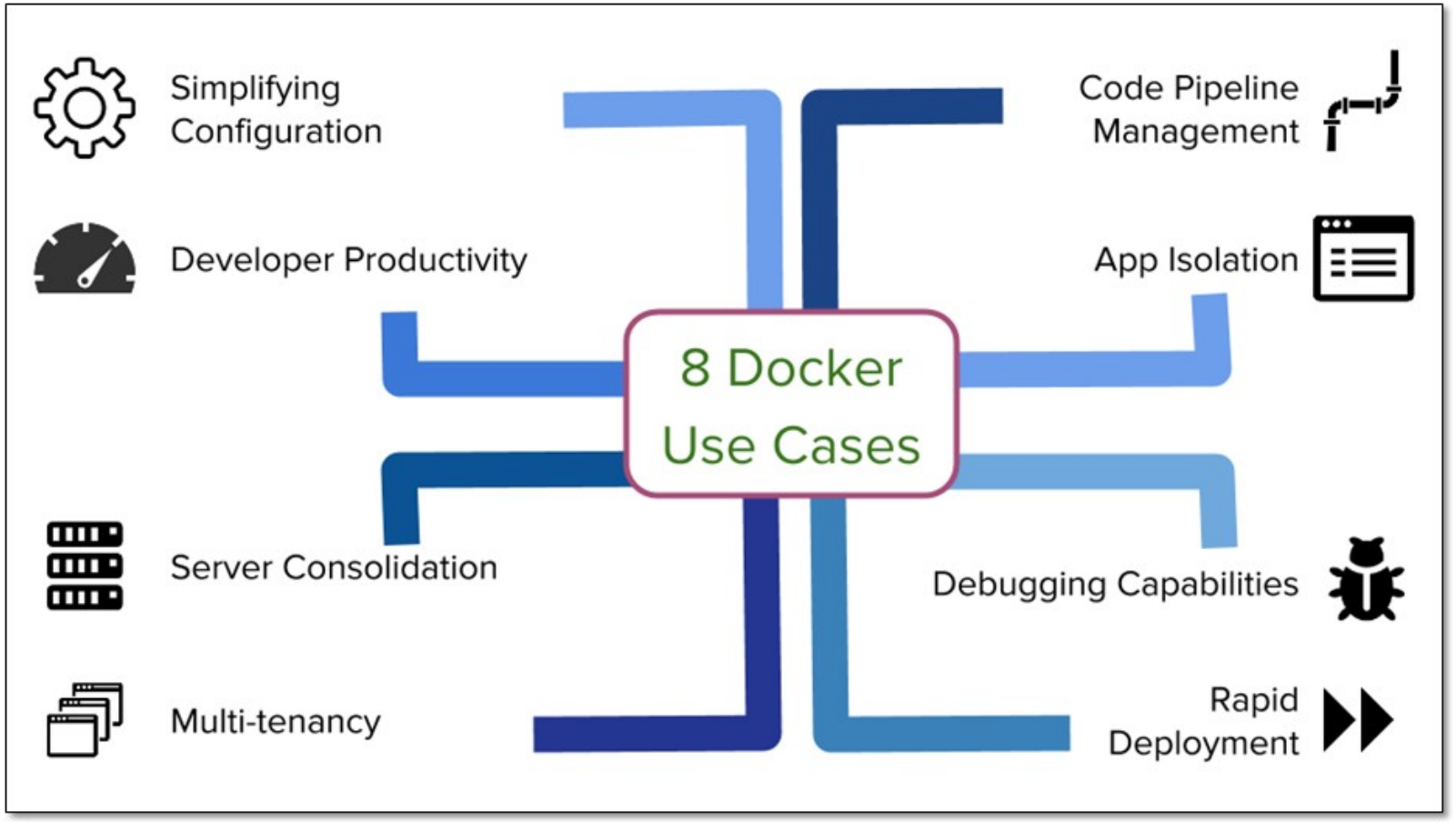


Docker – Use cases

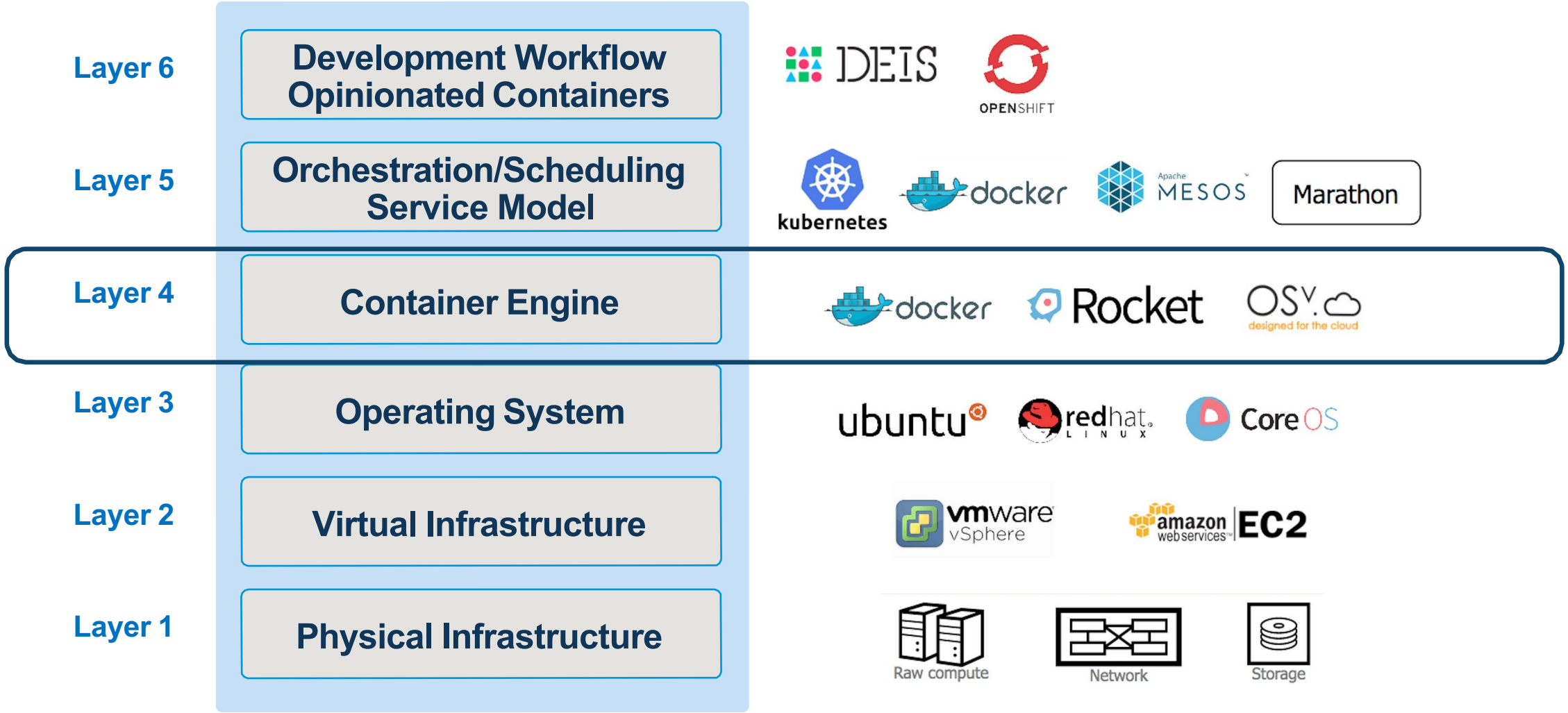
Server Apps like web servers, mail servers, databases, and proxies

Desktop software's, tools , email clients etc

Running these in a container and as a user with reduced previlgies will help protect your system from attack



Layers : Containers



Build your own image with a Dockerfile!

Step 1) Create Dockerfile to script how you want the image to be built

```
FROM java:8 # This might be an ubuntu or...
COPY *.jar app.jar
CMD java -jar app.jar
```

Step 2) **docker build** to build an image

Step 3) **docker push** to push to registry

Step 4) From another location, **docker pull** to download an image

Shared/Layered/Union Filesystems

Container images are *immutable* and *layered*, they are never changed, but rather composed of layers that add or override the contents of layers below. The container has a writeable layer to “make changes” on top of the read-only image layers, and is used to create working files, temporary files, and log files for instance. The container storage layer is exclusive to the running container.

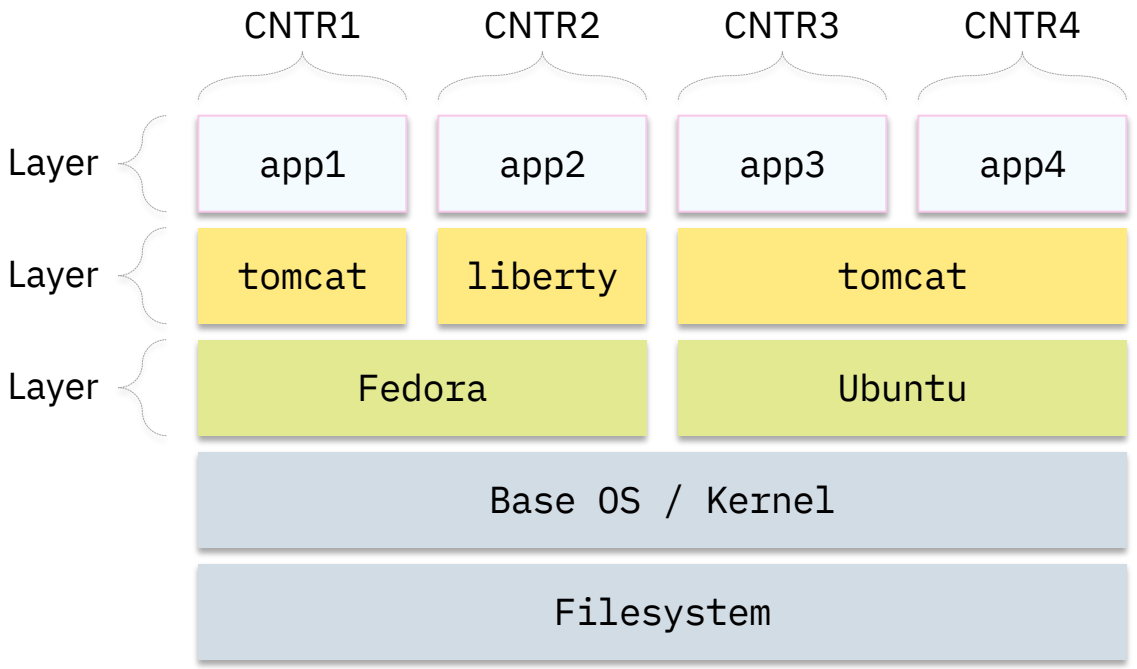
- Docker re-uses common layers between containers and images
- A single writeable layer is added on the top every time a new layer is created
- Layers are “smushed” with **union file system**
- Files are copied up when writes need to be made (copy-on-write)

Bottom Line

- More containers per host
- Faster downloads and uploads
- Faster container startups

ls /var/lib/docker/overlay2

```
0016ac03f0de110bd315ea3cd03546d4192ddd6a4a4c75ea1908c7edee69e9d3
0016ac03f0de110bd315ea3cd03546d4192ddd6a4a4c75ea1908c7edee69e9d3-init
16a28614760c68941fbd193fad753965943d35de3dfe5ebe059a1ba6d770fc10
37b3b057d9f04a0849dd74c3183604204e2bb745bd88d3b6c429a520fad8fb45
37d747c4f41f29b31664e357c5fde674025f9782c3336f29f4dcc2c85df15718
a5473075b9d58c609e45b0c226c2cf0495285a93898a2c8a478a2068c74630d1
l
```



Docker Volume

Volumes are the preferred mechanism for persisting data generated by and used by Docker containers, a new directory is created within Docker's storage directory on the host machine.

```
$> docker volume create hello
Hello
docker volume ls
DRIVER          VOLUME NAME
local           hello
$> docker volume inspect --format '{{ .Mountpoint }}' hello
/var/lib/docker/volumes/hello/_data
```

The Docker system paths are relative to the VM in which Docker is running.

You can connect to tty on Docker for Mac VM

```
screen ~/Library/Containers/com.docker.docker/Data/vms/0/tty
screen $(cat ~/Library/Containers/com.docker.docker/Data/vms/0/tty)
ls -al /var/lib/docker/volume
```

Or you can run a shell in the context of all namespaces on the Docker host (the process space of the mini VM running Docker4Mac) with `nsenter –pid-host`, and list the volumes,

```
docker run -it --privileged --pid=host debian nsenter -t 1 -m -u -n -i sh
/ # ls -al /var/lib/docker/volume
```

Docker Volume

Configure the container to use the volume,

```
$> docker run -d -v hello:/world busybox ls /world
```

The mount is created inside the container's /world directory.

By contrast, when you use a **bind mount**, a file or directory on the *host machine* is mounted into a container. The file or directory is referenced by its full or relative path on the host machine.

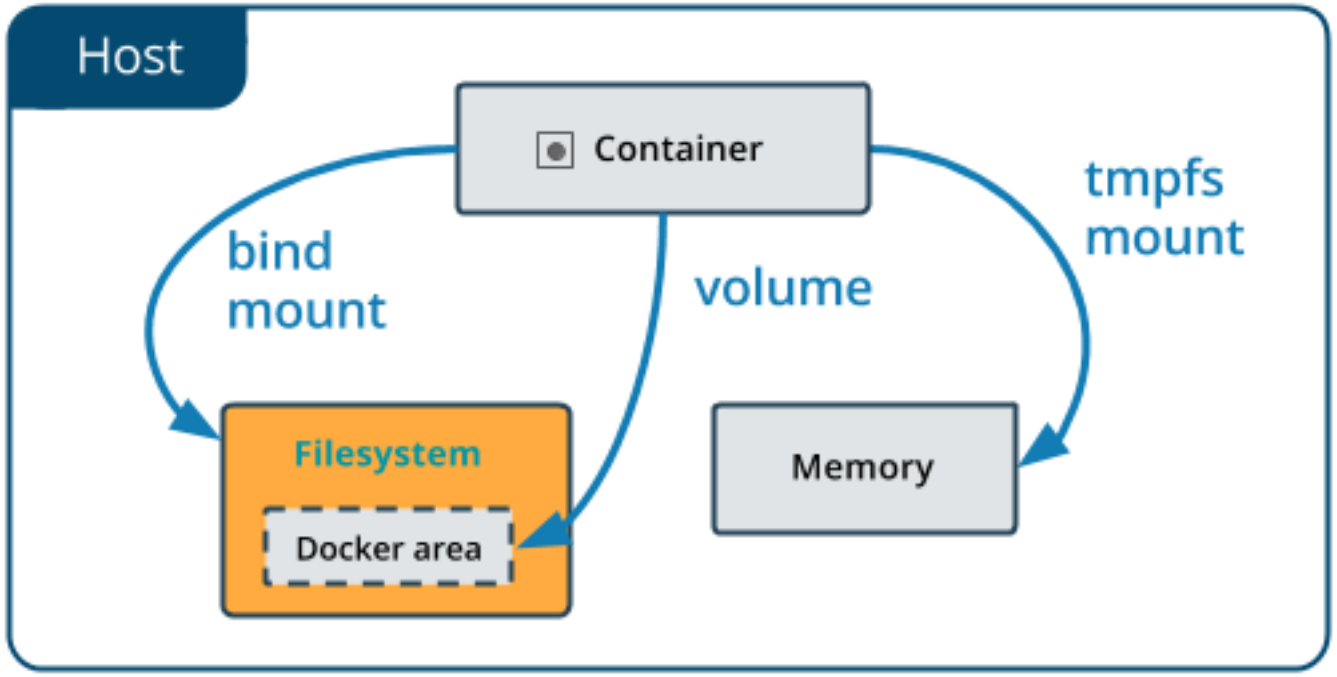


Image Optimization

- Use optimized Base Images, e.g. UBI
- Use Multi-Stage Builds, allows passing on artefacts
- Reduce number of layers
- Build custom base images
- Use production base images
- Avoid storing application data
- Design stateless containers without dependencies env and config
- Use .dockerignore when recursively using directories
- Add only required files
- Install required packages only
- Limit number of instructions or layers (RUN, COPY, ADD), e.g. use && conjunction
- Sort multi-line arguments
- Avoid using :latest
- Use exec form instead of shell form for ENTRYPOINT and CMD

Container Security

Container security includes securing the containers, the containerized application stack, the container pipeline (build-ship-run) and the infrastructure containers rely on and integrate with security tools and security policies.

4 Major areas of Docker security:

- Linux kernel security and its support for namespaces and cgroups
- Docker daemon attack surface; Running containers with Docker implies running the Docker daemon. See Podman.
- container configuration; in most cases, containers do not need full root privileges, and can run with a reduced capability set.
- Docker currently only supports the `hardening` security features of the kernel and how they interact with containers.

10 Key elements of container security:

1. **Multitenant host**: deploying multiple apps on single shared host: secure host kernel from containers and containers from each other. Drop privileges to least privilege possible; run as user not root; use Linux namespaces, SELinux (enforce mandatory access controls (MAC) for every user, application, process, and file), Cgroups, capabilities (lock down root in a container), and seccomp profiles to restrict available system calls; use lightweight operating system and optimized host;
2. **Container content**: trusted base images, e.g. Universal Base Images (UBI); container security monitoring and security scanning like OpenSCAP;
3. **Container registries**
4. **Building containers**: Source-to-Image (S2I); integrated Jenkins; integration RESTful APIs; SAST, DAST; vulnerability scanning; separate container layers;
5. **Deployment**: automated, policy-based deployment; Security Context Constraints (SCC) as Pod Security Policy and Container Security Policy.
6. **Container orchestration**: capacity; shared resources management like network and storage; container health, e.g. CloudForms; scaling; integrated OAuth server; multitenancy security;
7. **Network isolation**: network namespaces; pod-network, software defined network (SDN) and SDN plugins (ovs-subnet, ovs-multitenant, ovs-networkpolicy); SDN solutions like Calico; Network Policy;
8. **Storage**: PV with access modes; use annotations on PVs to add group IDs (gid); use SELinux to secure mounted volume; encrypt data-in-transit;
9. **API management for Microservices**: 3Scale, API Connect, Loopback, OpenAPIs;
10. **Federated clusters**: including federated secrets, federated namespaces and Ingress objects.

Source: https://www.redhat.com/cms/managed-files/cl-container-security-openshift-cloud-devops-tech-detail-f7530kc-201705-en.pdf

Podman

Podman is an alternative to the Docker client and is an open source tool for managing containers and container images and interacting with image registries using the Open Container Initiative (OCI) image format. Podman stores local images in local file-system, avoiding having to use a Docker daemon. Podman is compatible with Kubernetes.

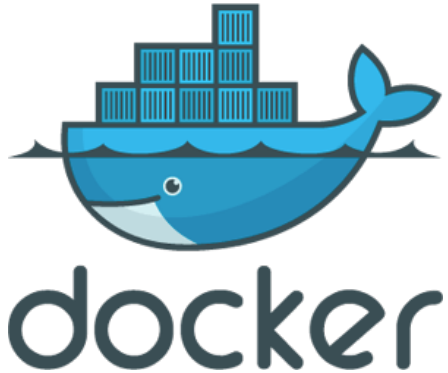
In a Traditional Deployment…

Are you testing these on ever commit?

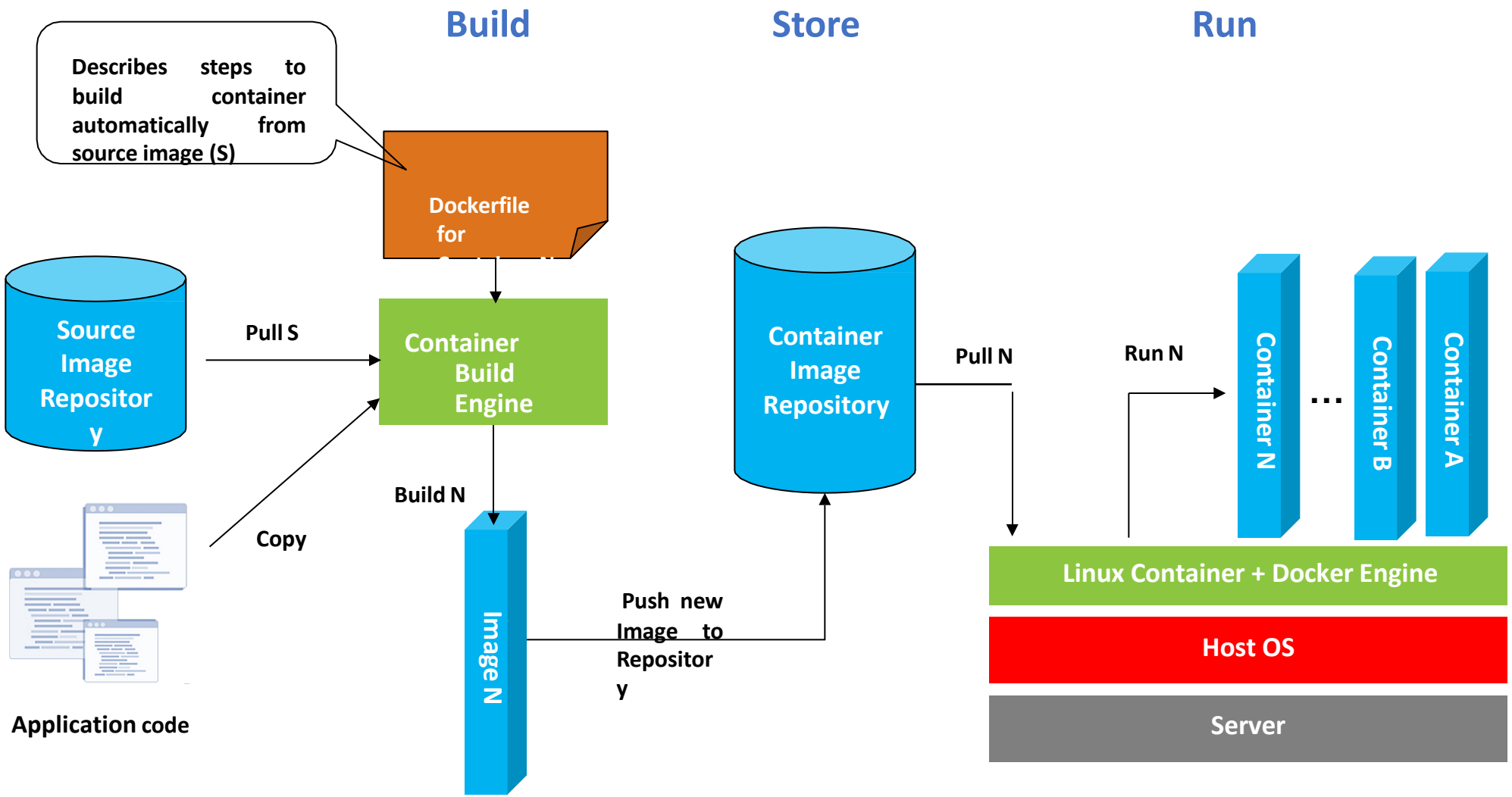
- Code (packages archive) ✓
- App server ✗
- Runtime versions ✗
- System libraries and versions ✗

Container = Code + Dependencies

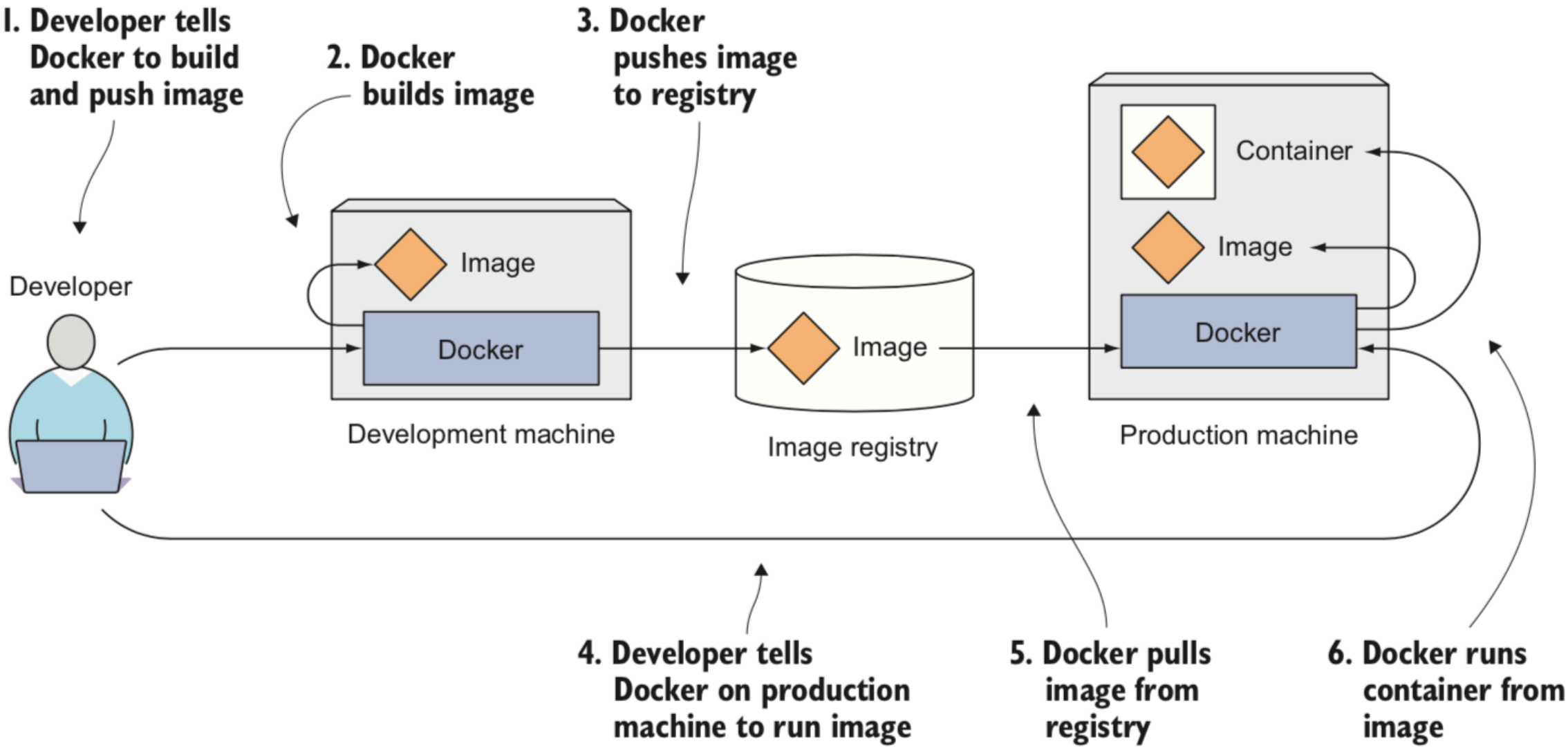
- Code (packages archive)
- App server
- Runtime versions
- System libraries and versions



Docker container lifecycle



Docker Flow


1. Developer tells Docker to build and push image
2. Docker builds image
3. Docker pushes image to registry
Developer
Image
Docker
Development machine
Image
Image registry
Container
Image
Docker
Production machine
4. Developer tells Docker on production machine to run image
5. Docker pulls image from registry
6. Docker runs container from image

Summary

Why Containers ? When compared to VMs:

- **Low hardware footprint**, better resource utilization (CPU, Memory, Disk – resources managed using namespaces and cgroups)
- Faster start-up times, no OS install or restart
- **Quick deployment**
- Better efficiency, elasticity, and reusability of the hosted applications
- Better portability of platform and containers
- **Environment isolation**, changes to host OS do not affect the container
- **Multiple environment deployment**
- **Reusability**
- Easier tooling / scripting
- Ideal for microservices

Docker value-add:

- User Experience
- Image layers
- Easily share images - DockerHub

- Docker Desktop - https://www.docker.com/products/docker-desktop
- Type the following command in your terminal:
  docker run -dp 80:80 docker/getting-started
- Open your browser to http://localhost/tutorial/
- Sign up for a Free Docker account at https://hub.docker.com/