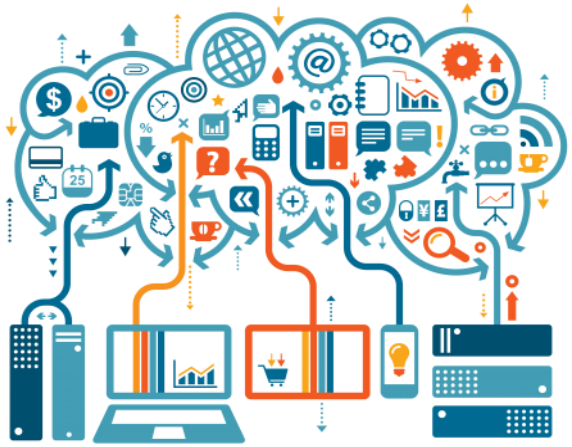
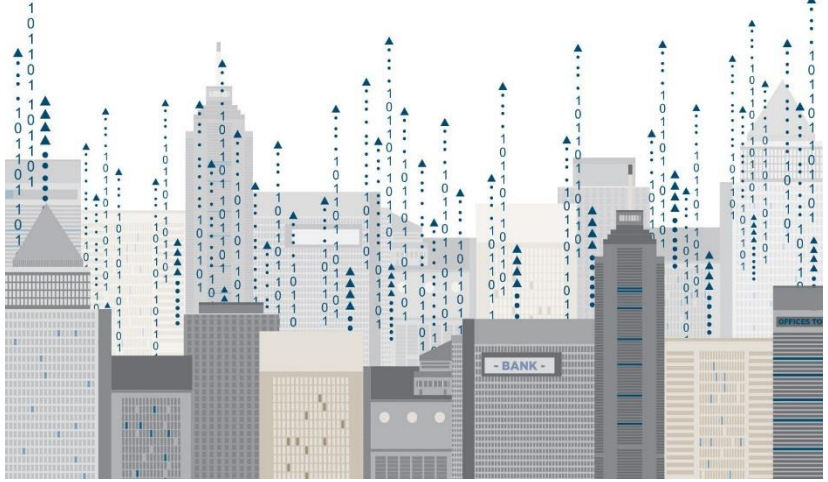




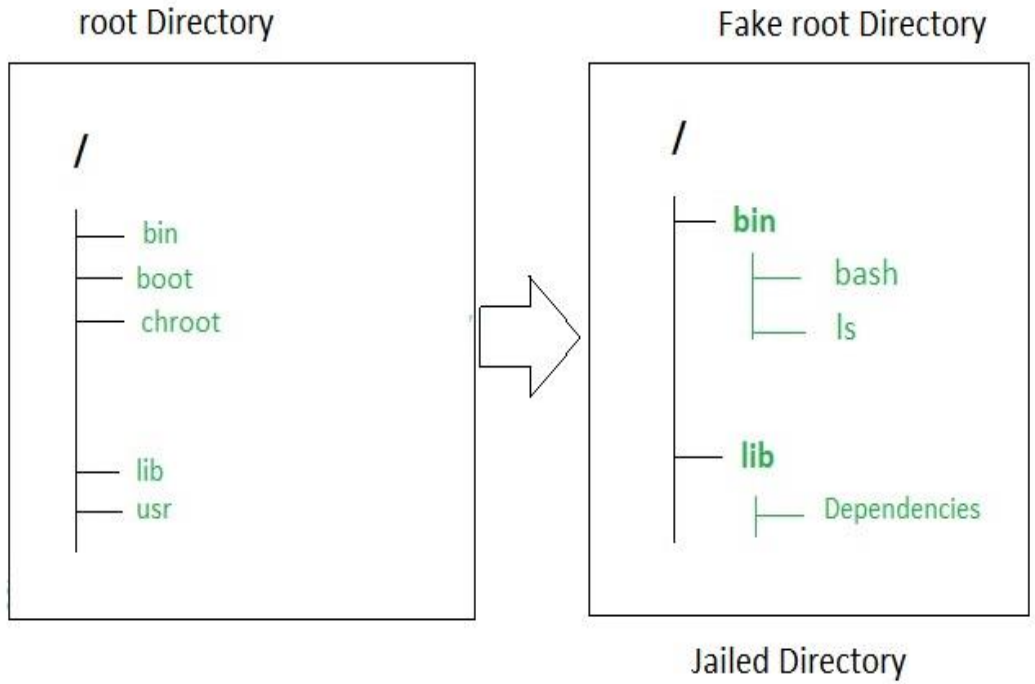
Big Data Analytics

Fall 2025

Lecture 4

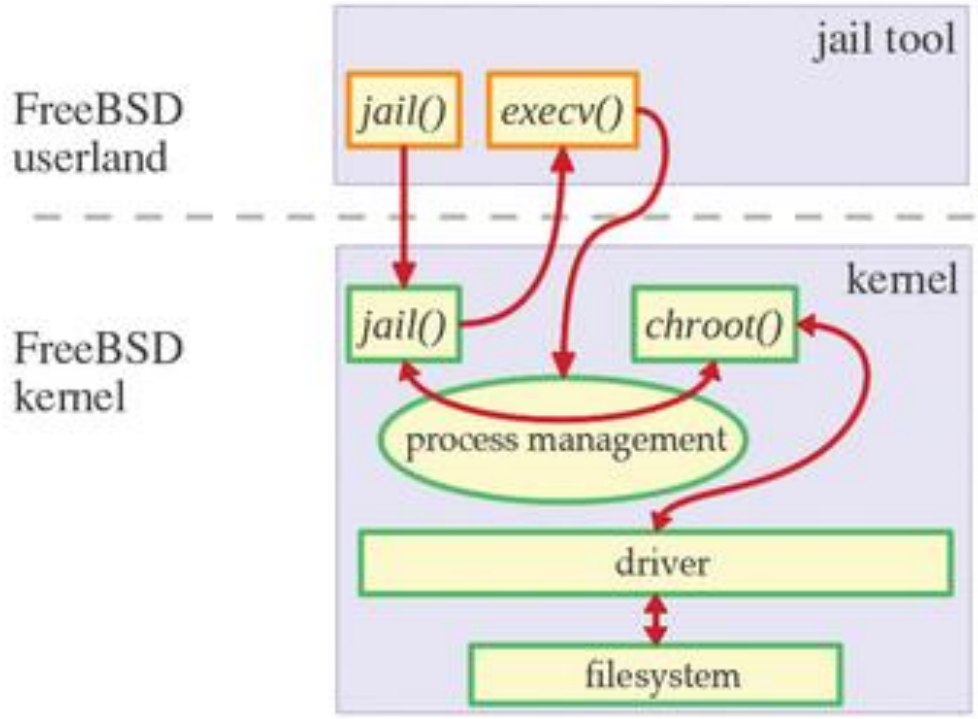
History of Linux Containers

- Chroot (1979): Change Root
  - Changes the root directory (/) for the current process
  - Process thinks (/mnt/test) is root :)
  - Hence, a simple process isolation.
  - Didn’t offer strong security
  - Only filesystem isolation



History of Linux Containers

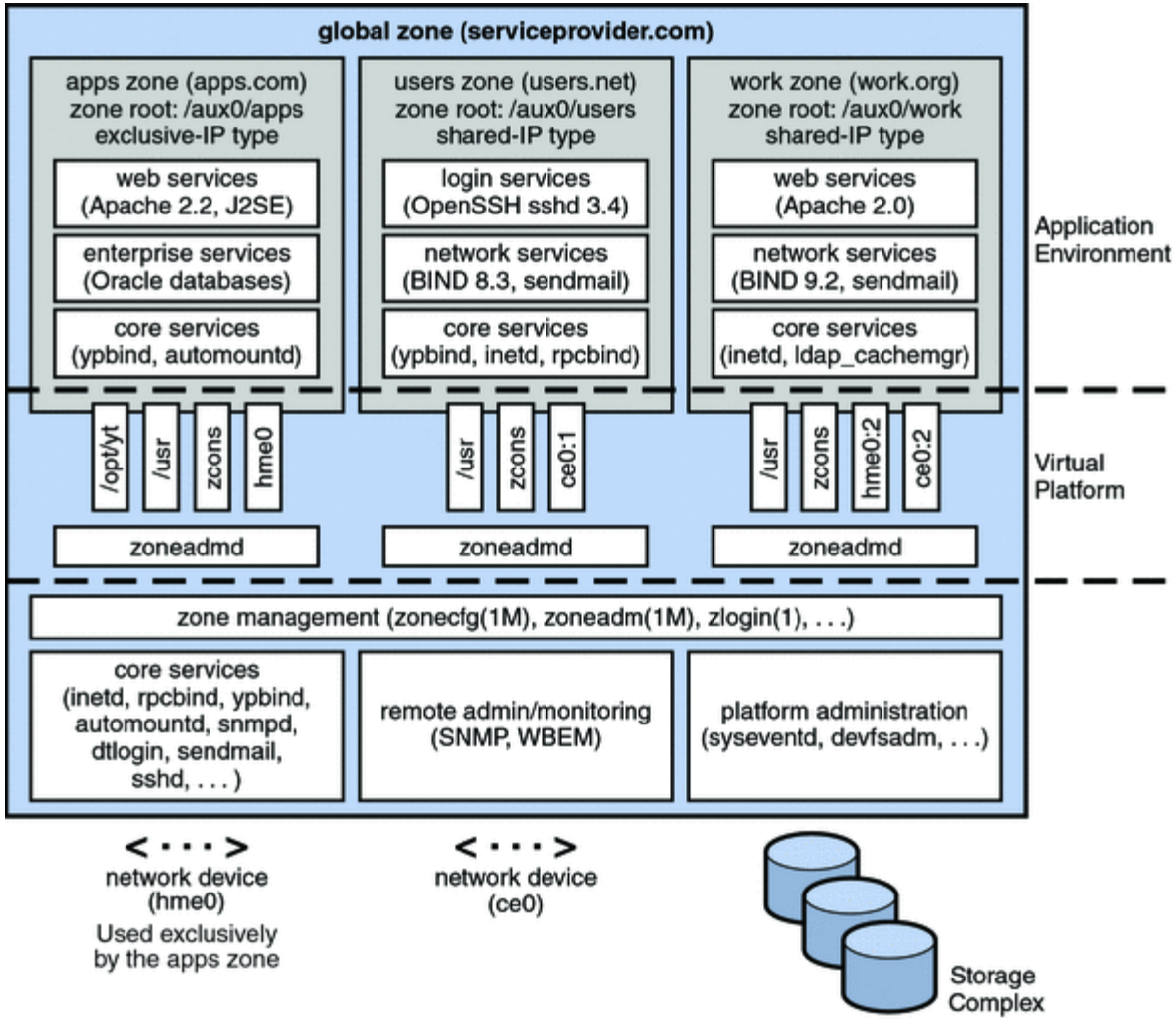
- Jails (2000):
  - Extended chroot by adding restrictions
  - File, process, user and networking (IP) isolation – so it's a jail :)
  - More secure environment
  - Limits what processes could do outside their jail.
  - Closer to containers



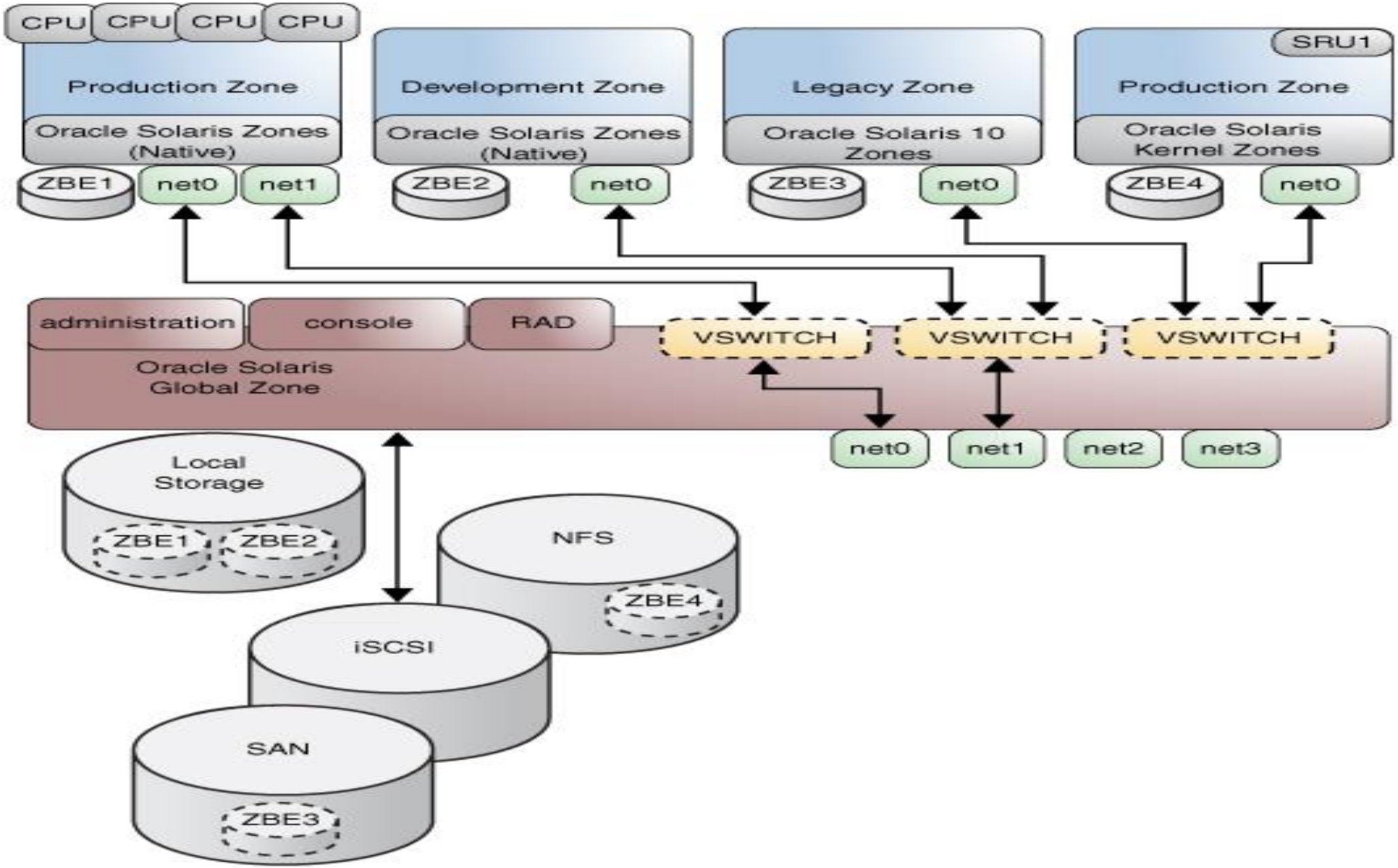
https://www.admin-magazine.com/index.php/Archive/2013/13/How-to-configure-and-use-jailed-processes-in-FreeBSD/%28offset%29/3

History of Linux Containers

- Solaris Containers (2004):
  - Zone: a virtualized OS env created within one instance of Solaris.
  - Separate environments on 1 instance
  - Abstract layer: Separates applications from physical attributes of machine
  - Closer to containers: isolated environments with dedicated resources.



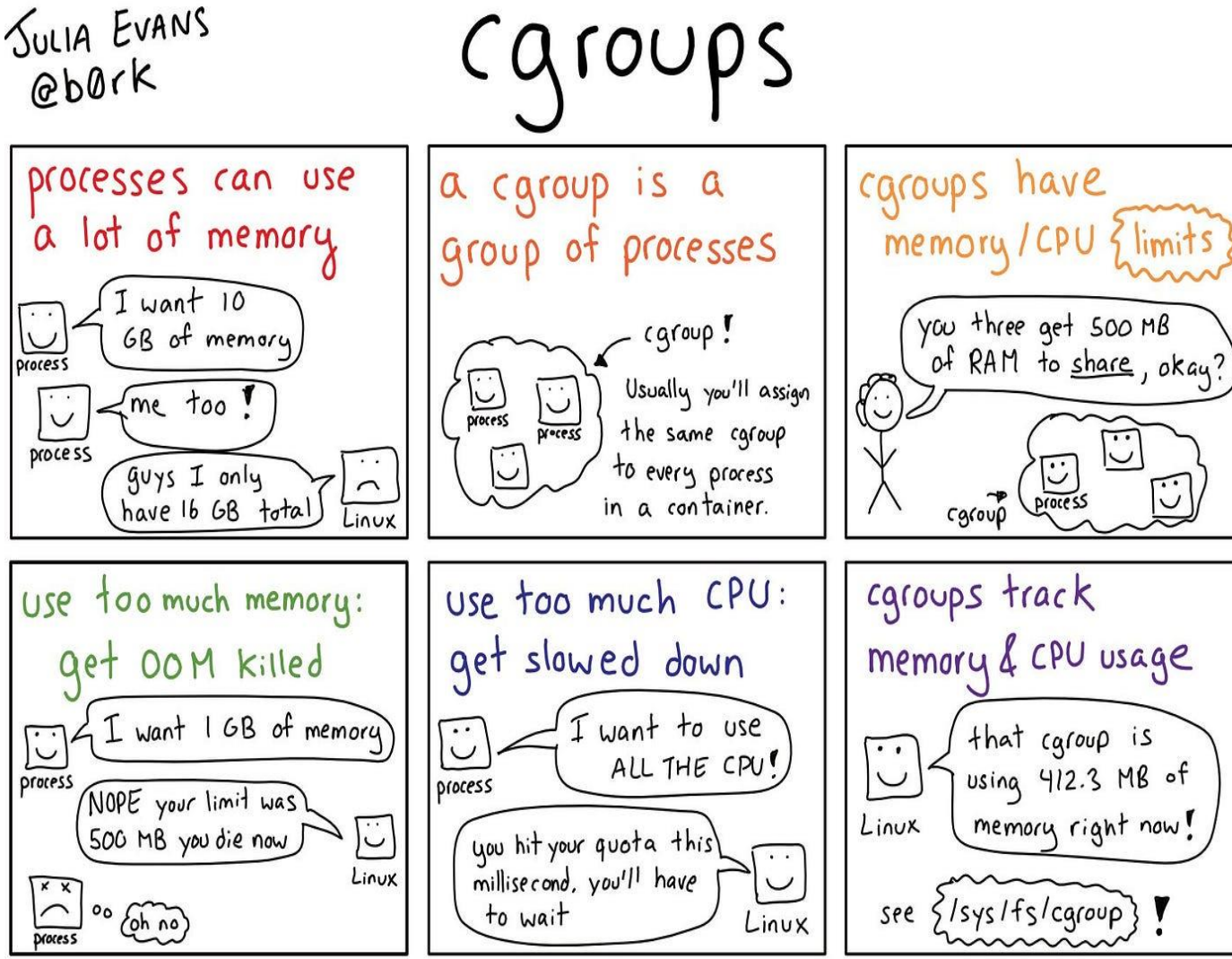
https://en.wikipedia.org/wiki/Solaris_Containers



https://en.wikipedia.org/wiki/Solaris_Containers

History of Linux Containers

- **Control Groups (cgroups) (2007):**
  - Introduced in Linux kernel by Google.
  - Allows partitioning of resources such as CPU, memory, I/O, bandwidth among processes
  - Easy: resource allocation, prioritization, control (freeze, checkpoint), accounting
  - Building block for Docker, LXC, Kubernetes



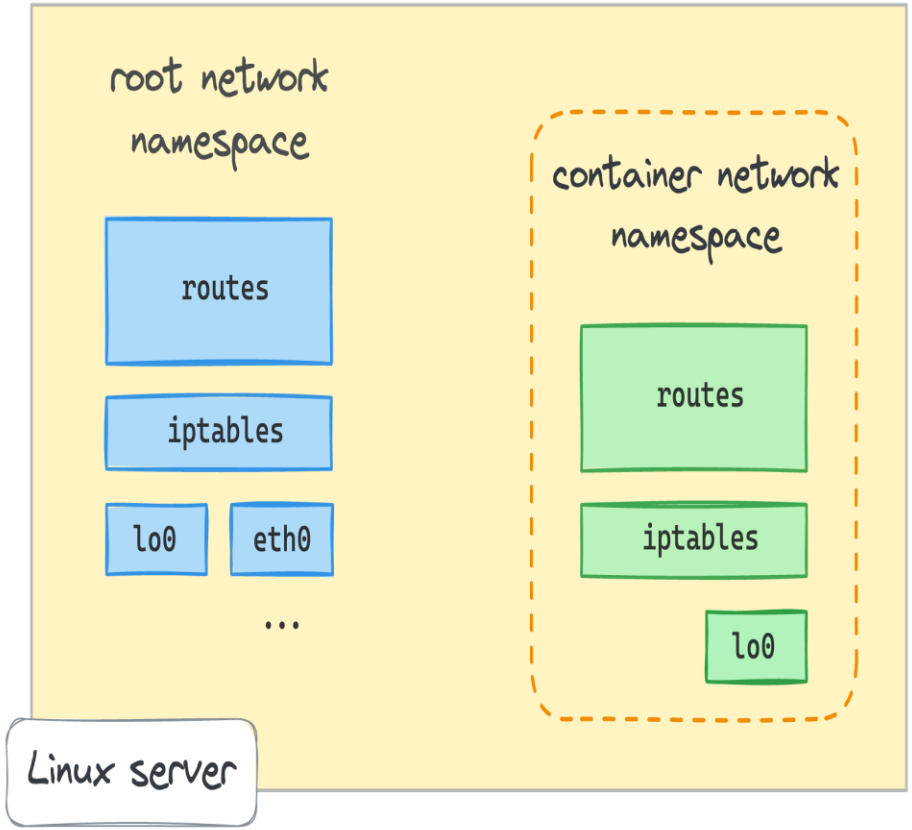
History of Linux Containers

- **Linux Namespaces (2002):**
  - **Isolation:** what process *sees*
  - Isolate resources that process can use
  - Fine-grain partitioning of global OS resources
- Remember:
  - Cgroup: what process *uses*

```
cryptonite@cryptonite:~ $ echo $$
4622
cryptonite@cryptonite:~ $ ls /proc/$$/ns -al
total 0
dr-x--x--x 2 cryptonite cryptonite 0 Jun 29 15:00 .
dr-xr-xr-x 9 cryptonite cryptonite 0 Jun 29 13:13 ..
lrwxrwxrwx 1 cryptonite cryptonite 0 Jun 29 15:00 cgroup -> 'cgroup:[4026531835]'
lrwxrwxrwx 1 cryptonite cryptonite 0 Jun 29 15:00 ipc -> 'ipc:[4026531839]'
lrwxrwxrwx 1 cryptonite cryptonite 0 Jun 29 15:00 mnt -> 'mnt:[4026531840]'
lrwxrwxrwx 1 cryptonite cryptonite 0 Jun 29 15:00 net -> 'net:[4026532008]'
lrwxrwxrwx 1 cryptonite cryptonite 0 Jun 29 15:00 pid -> 'pid:[4026531836]'
lrwxrwxrwx 1 cryptonite cryptonite 0 Jun 29 15:00 pid_for_children -> 'pid:[4026531836]'
lrwxrwxrwx 1 cryptonite cryptonite 0 Jun 29 15:00 time -> 'time:[4026531834]'
lrwxrwxrwx 1 cryptonite cryptonite 0 Jun 29 15:00 time_for_children -> 'time:[4026531834]'
lrwxrwxrwx 1 cryptonite cryptonite 0 Jun 29 15:00 user -> 'user:[4026531837]'
lrwxrwxrwx 1 cryptonite cryptonite 0 Jun 29 15:00 uts -> 'uts:[4026531838]'
```

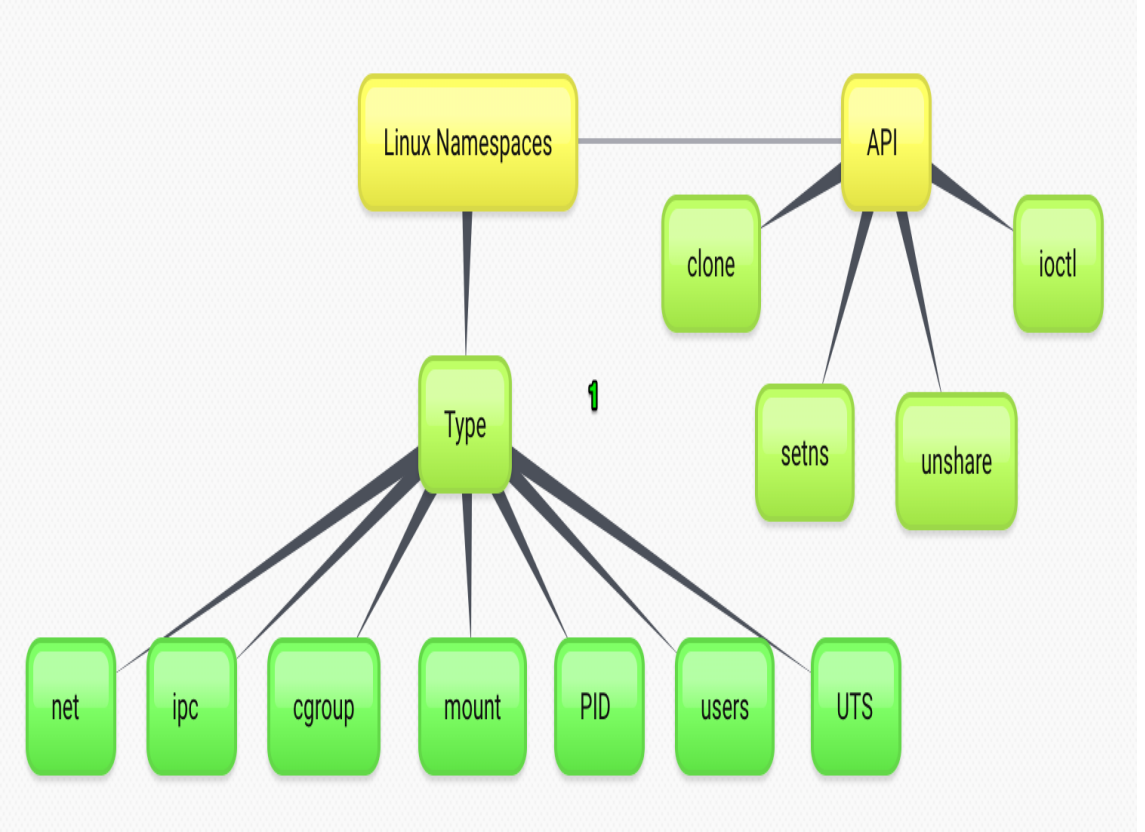
History of Linux Containers

- **Linux Namespaces (2002):**
  - **Box** containing global resources
  - **Add and remove from one box - no effect on content of the other boxes**.
  - Process in box A goes crazy and deletes the view of its whole filesystem or network stack - no issues for process in box B
  - Deleting namespace does not delete global resource – as namespace is only a view :)

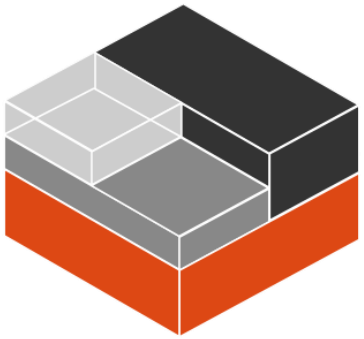


History of Linux Containers

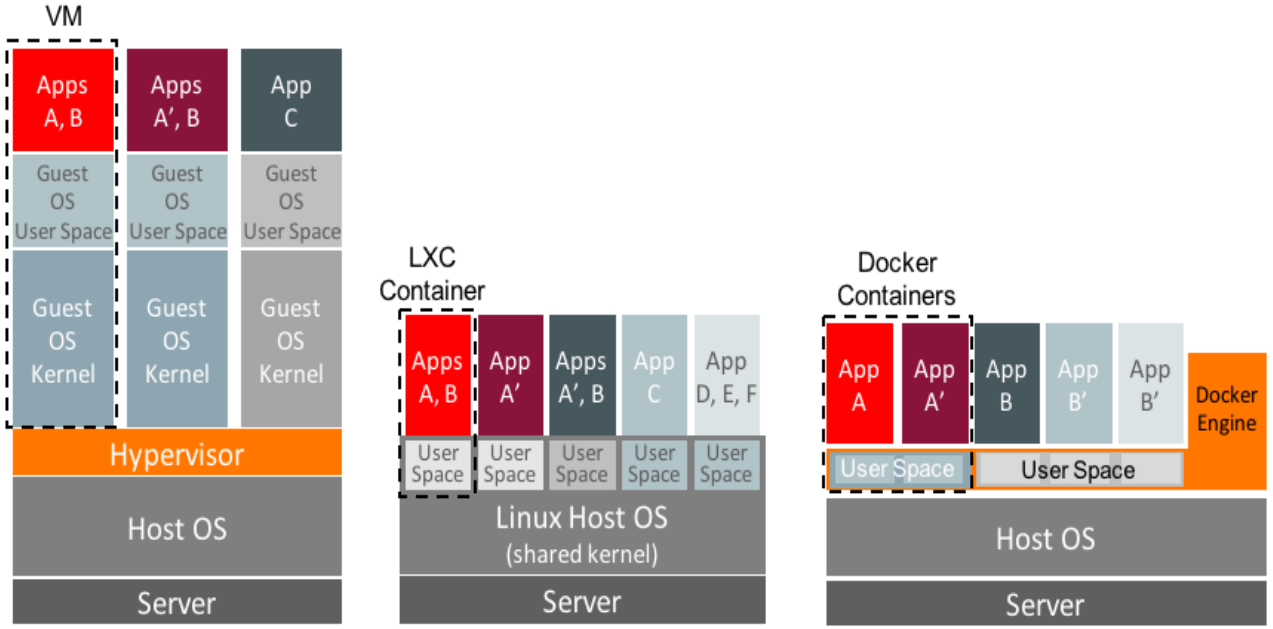
- **Linux Namespaces (2002):**
  - **PID**: isolate system process tree
  - **NET**: isolate host network stack
  - **MNT**: isolate filesystem mount points
  - **UTS**: isolate hostname
  - **IPC**: isolate IPC (semaphores)
  - **USER**: isolate system users IDs
  - **CGROUP**: isolate virtual cgroup filesystem of the host.



History of Linux Containers

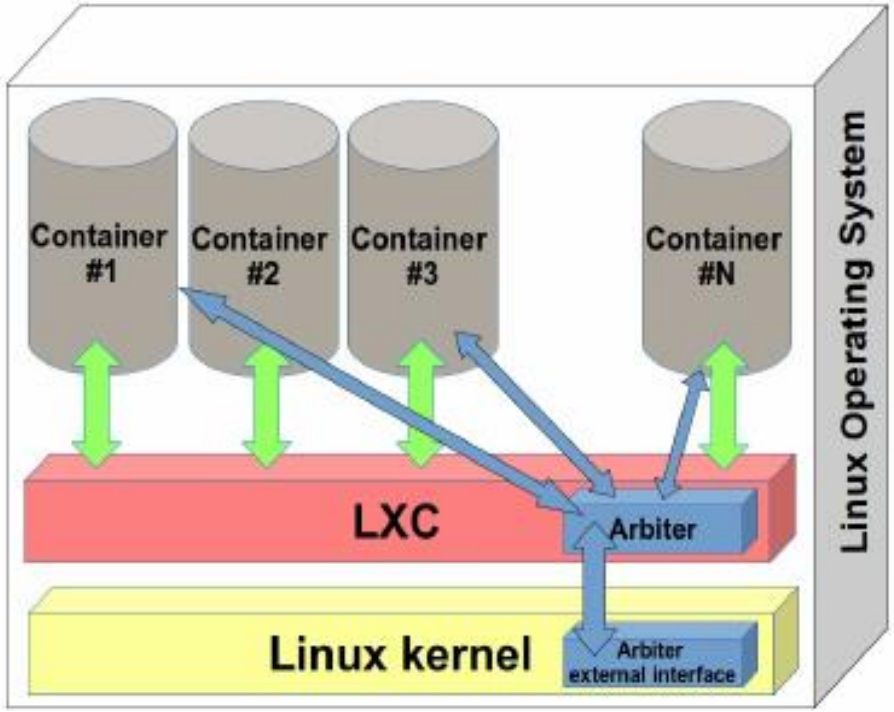


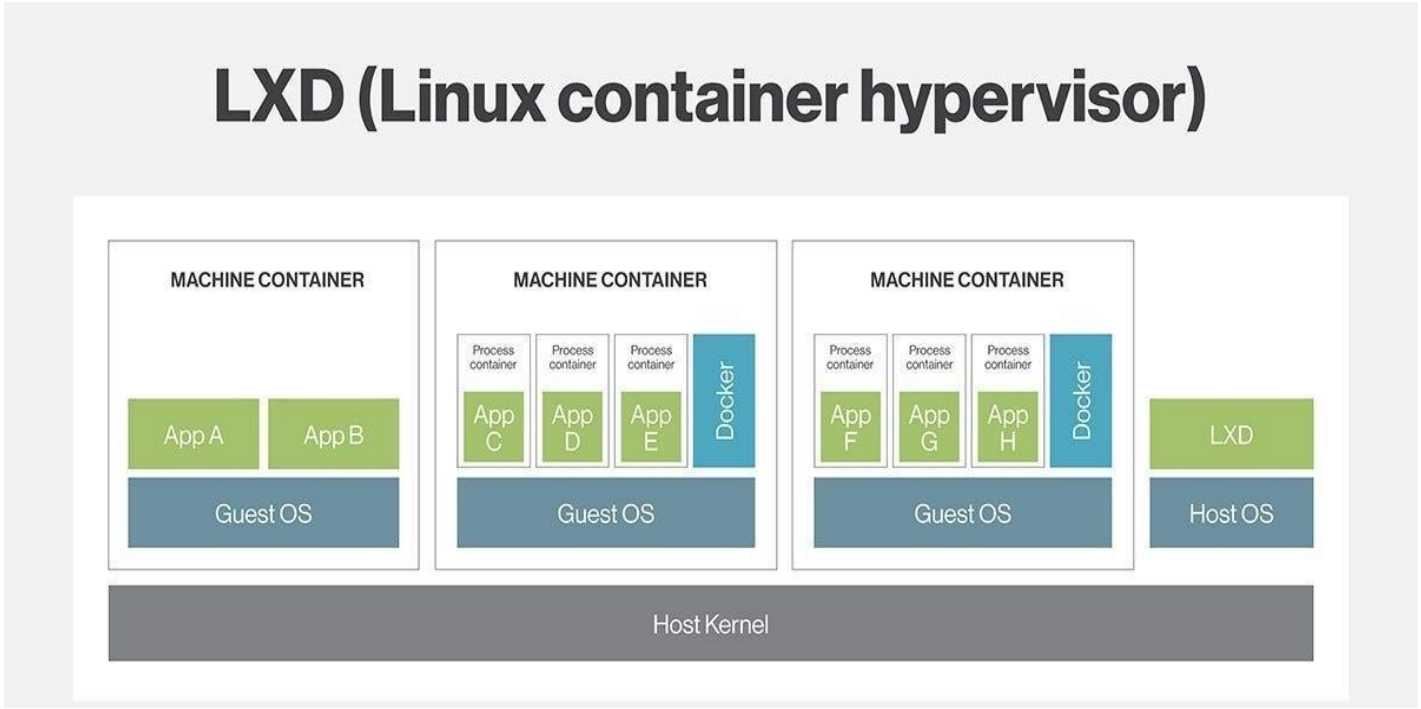
- **LXC - Linux Containers (2008):**
  - 1st complete implementation of containers on Linux.
  - Uses cgroups and namespaces
  - Lightweight virtualization
  - Multiple containers on single host.
  - Runtime Env: consists of tools, templates, libraries and language bindings.



https://linuxcontainers.org/

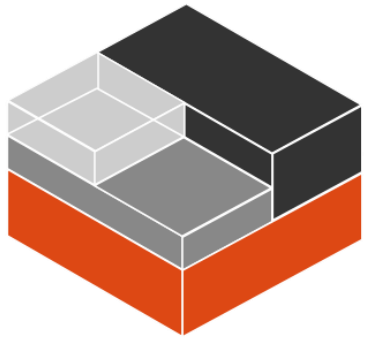
History of Linux Containers



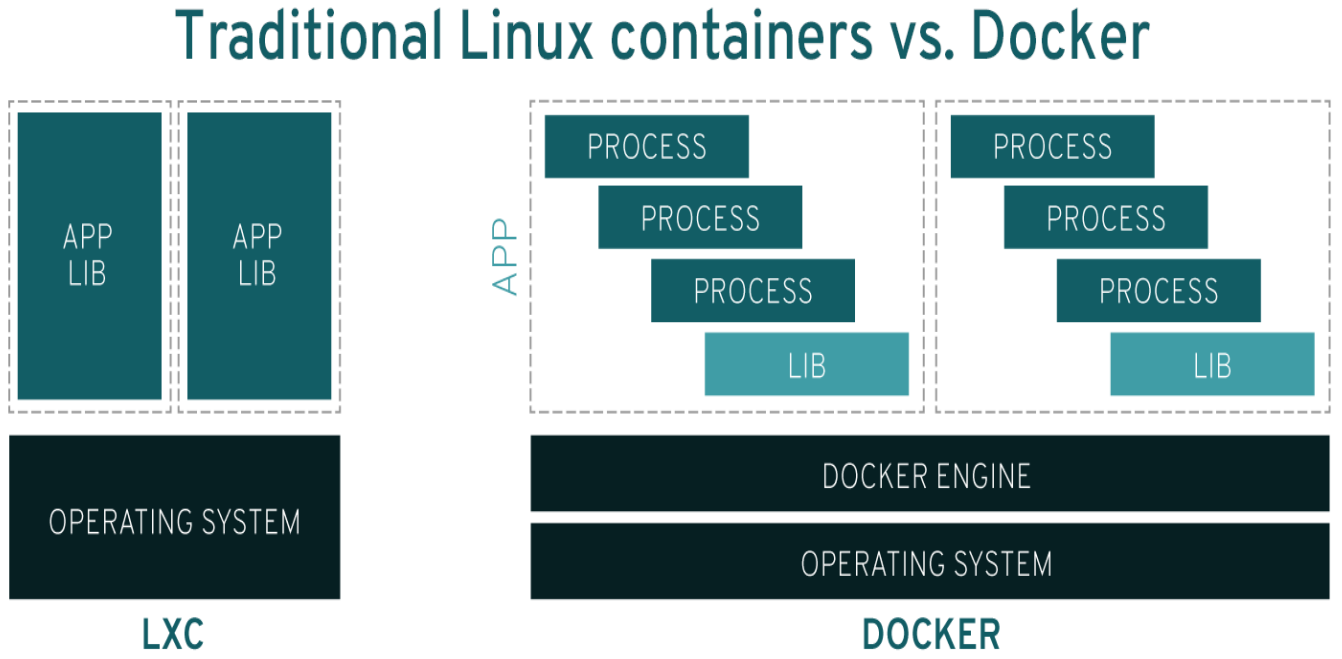


Machine Emulator and Virtualizer: https://www.qemu.org/
Turns Linux into a Hypervisor: https://www.redhat.com/en/topics/virtualization/what-is-KVM
Another famous Virtualizer: https://canonical.com/lxd

History of Linux Containers



- **Docker (2013):**
  - Made containers more accessible, portable, and manageable.
  - Used LXC initially
  - Later developed libcontainer: interact directly with cgroups and namespaces.
  - Introduced container images: easy now to package and ship apps with their dependencies.



LXC vs Docker – Comparison Summary

	LXC	Docker
Developed By	Created by IBM, Google, Virtuozzo, & Eric Biederman	Founded as DotCloud by Solomon Hykes
Host-Machine Utilization	Lightweight OS-Level Virtualization	Application-Level Virtualization
Usability	Multi-Purpose Solution for Virtualization	Single-Purpose Solution for Virtualization
Performance	Commendable Speed, But Not Lightweight and Can Take Time	Fast Container Creation and Deployment Due to Lightweight Nature
Security	Privileged Containers are Insecure and Require Kernel Features, whereas Unprivileged Containers are Relatively Safer	Reliable Isolation Techniques, such as Seccomp and User Namespaces
Ease of Use	LXC Requires a Steeper Learning Curve, Not Easy for Newcomers; Configuration is a little complex	User-Friendly Interface Makes It Approachable For Users of All Levels
Scalability	Little Difficult To Scale, But Doable	Lightweight Nature Allows Easy Scalability Due to Docker Swarm and Kubernetes
Tools	Lacks User-Friendly Tools Like Docker, Making Container Management Complicated	Rich Set of Commands That Makes Managing Containers Easy
Data Retrieval	Not Supported After Processed	Data Retrieval Supported
Cloud Support	Not Required as Each Feature Provided by Linux	Cloud Storage Needed for a Sizeable Ecosystem
Community & Ecosystem	Smaller Community Support and Ecosystem Compared to Docker	Extensive Ecosystem with a Vast Repository of Pre-Built Images and 3rd Party Tools

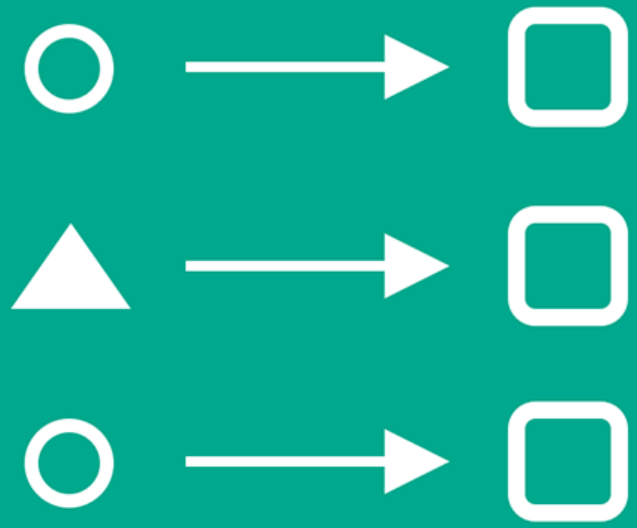
Orchestration

- Automated configuration, coordination, and management of complex systems and services.
- Streamline deployment, scaling, and operations
- Needs distributed infrastructure: cloud or data centers.

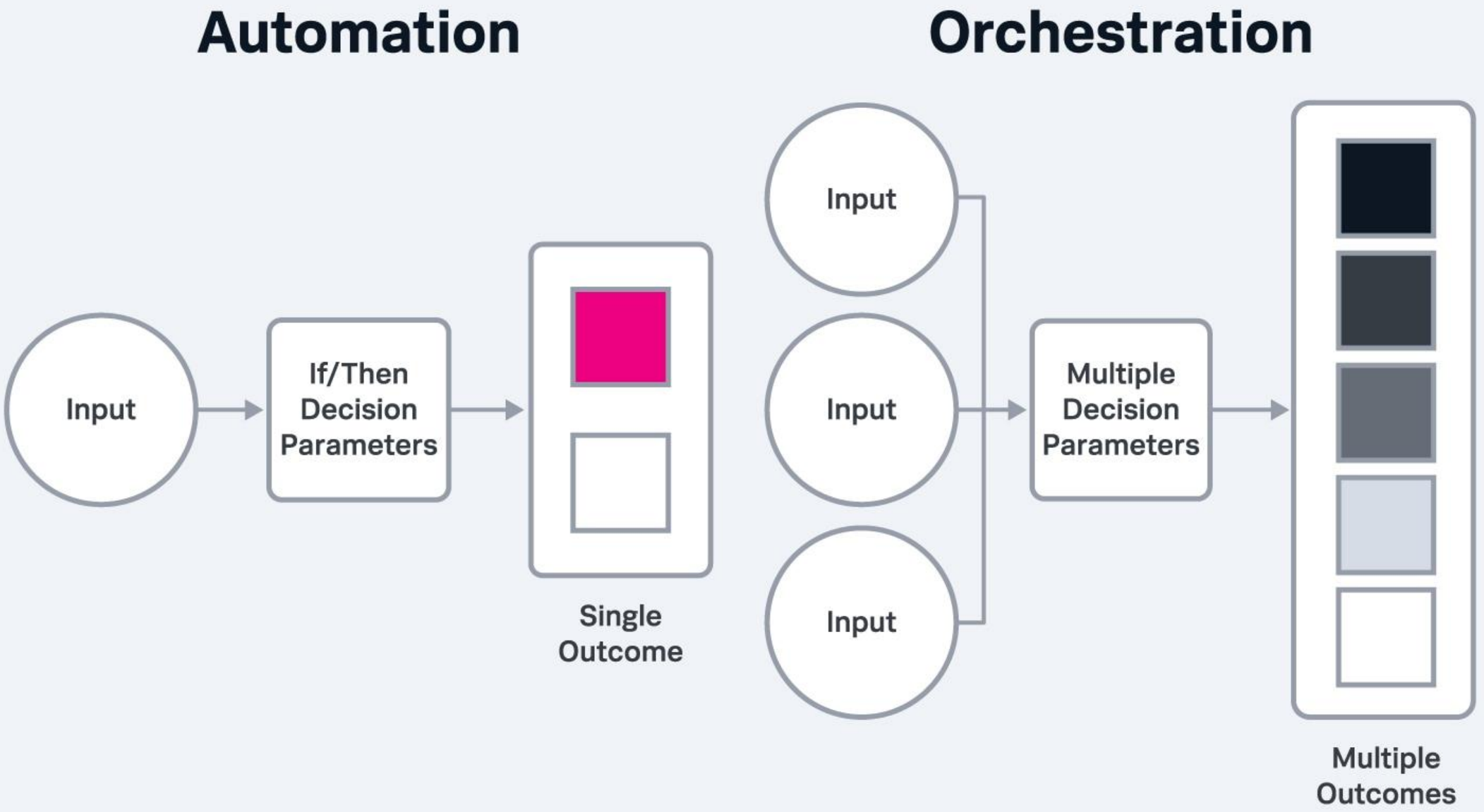
Orchestration

Automating many tasks as a process or workflow.

Automation



Setting up one task to run on its own.


Automation
Orchestration
Input
If/Then Decision Parameters
Single Outcome
Input
Input
Input
Multiple Decision Parameters
Multiple Outcomes

Orchestration

- Where should each container run?
- How do I keep it running if it crashes?
- How do I connect containers across hosts?
- How do I scale up/down automatically?
- How do I update apps with zero downtime?

Orchestration – What I do?

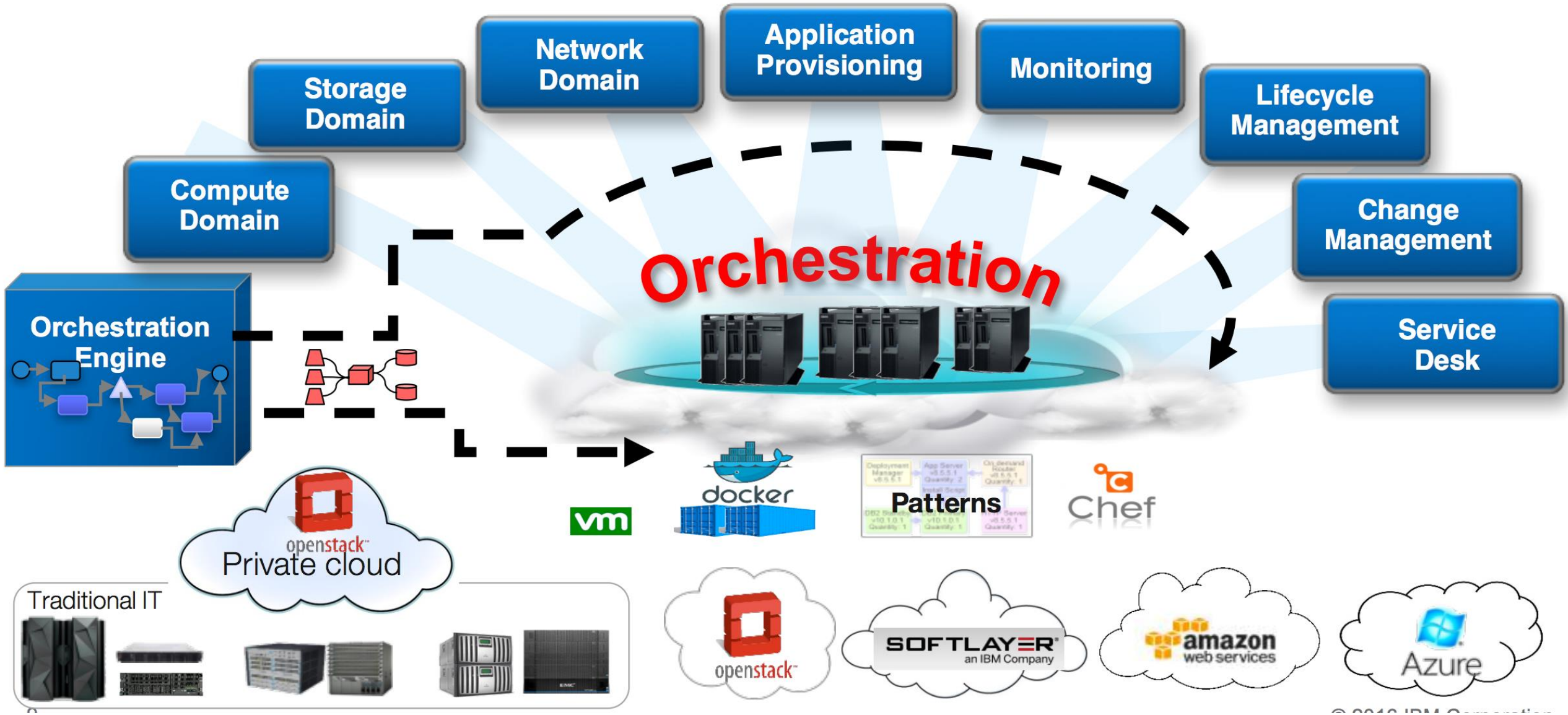
- Provision and Deployment – which container to run where?
- Resource management – ensures quota and fairness
- Scheduling – find “best fit”node for each container
- Networking – each container has a virtual IP
- Load Balancing
- Scaling – add/remove replicas based on demand
- Secrets Management
- Rolling updates

Popular

- Docker swarm
- Kubernetes (K8s)
- Apache Mesos
- Nomad (Hashicorp)

```
apiVersion: apps/v1

kind: Deployment

metadata:
  name: webapp

spec:
  replicas: 3
  selector:
    matchLabels:
      app: webapp
  template:
    metadata:
      labels:
        app: webapp
    spec:
      containers:
      - name: webapp
        image: nginx
        ports:
        - containerPort: 80
```

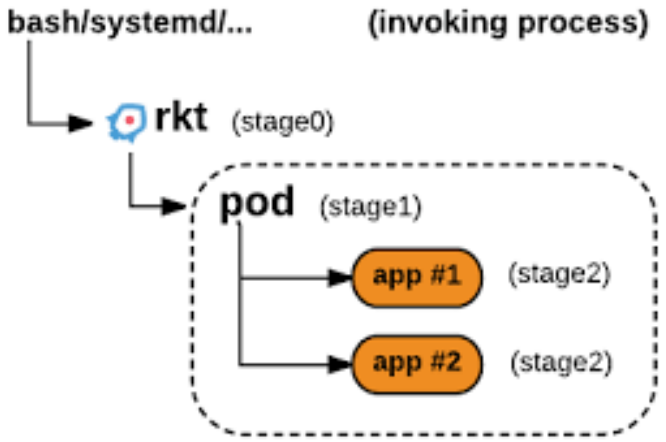
Run Web App with 3 replicas - Kubernetes

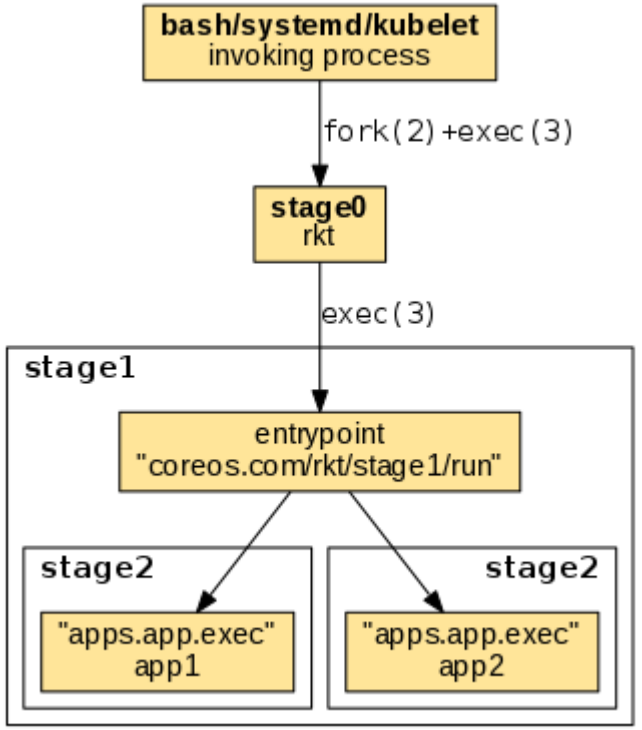

Storage Domain
Network Domain
Application Provisioning
Monitoring
Lifecycle Management
Compute Domain
Change Management
Service Desk
Orchestration
Orchestration Engine
docker
Patterns
Chef
openstack
Private cloud
Traditional IT
openstack
SOFTLAYER
an IBM Company
amazon
web services
Azure

https://rocket.readthedocs.io/en/latest/CHANGELOG/

Containers in the cloud

- **rkt (2014):**
  - A container runtime alternative to Docker.
  - Designed with a stronger focus on security, simplicity, and standards compliance.
  - Packaged as a CLI tool (rkt) to run container images.
  - No central daemon (as in Docker initially)

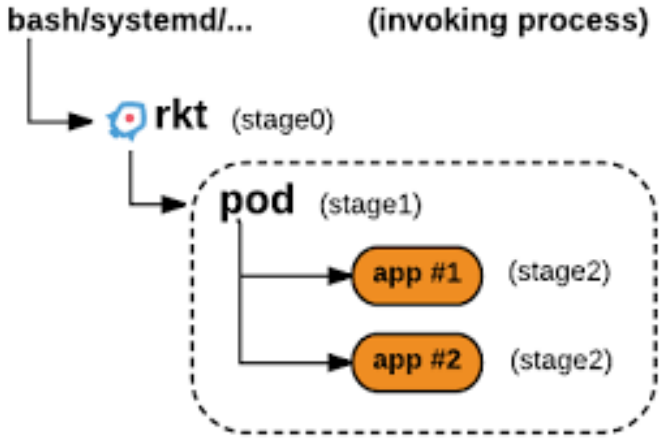


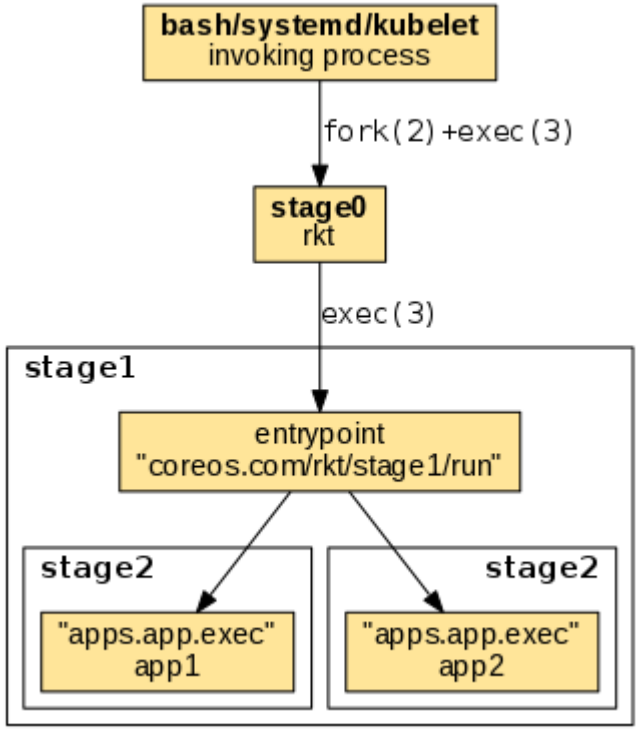


https://rocket.readthedocs.io/en/latest/CHANGELOG/

Containers in the cloud

- **rkt (2014):**
  - Core execution unit: pod - a pluggable collection of one or more applications executing in a shared context
  - Allows users to apply different configurations at both pod-level and at the more granular per-application level.

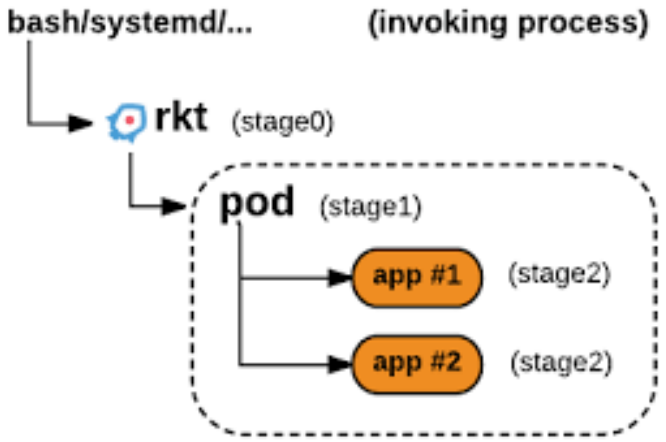


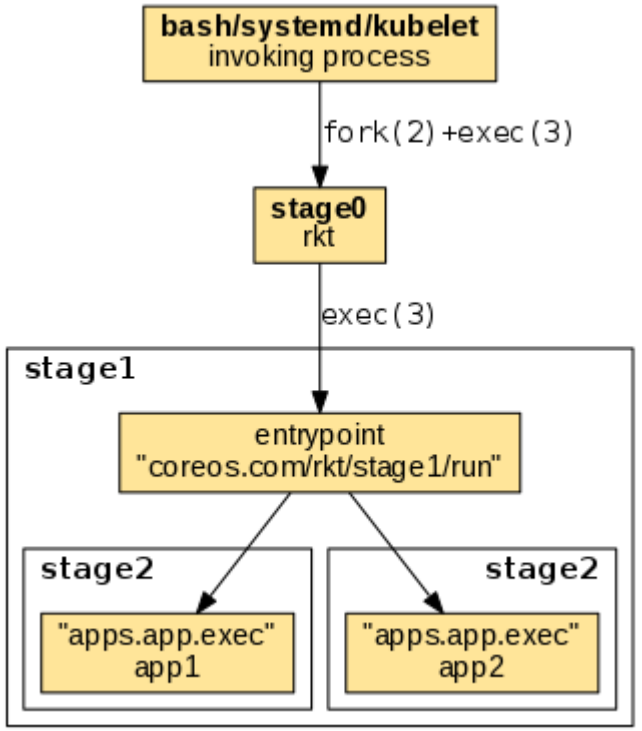


https://www.redhat.com/en/topics/containers/what-is-rkt

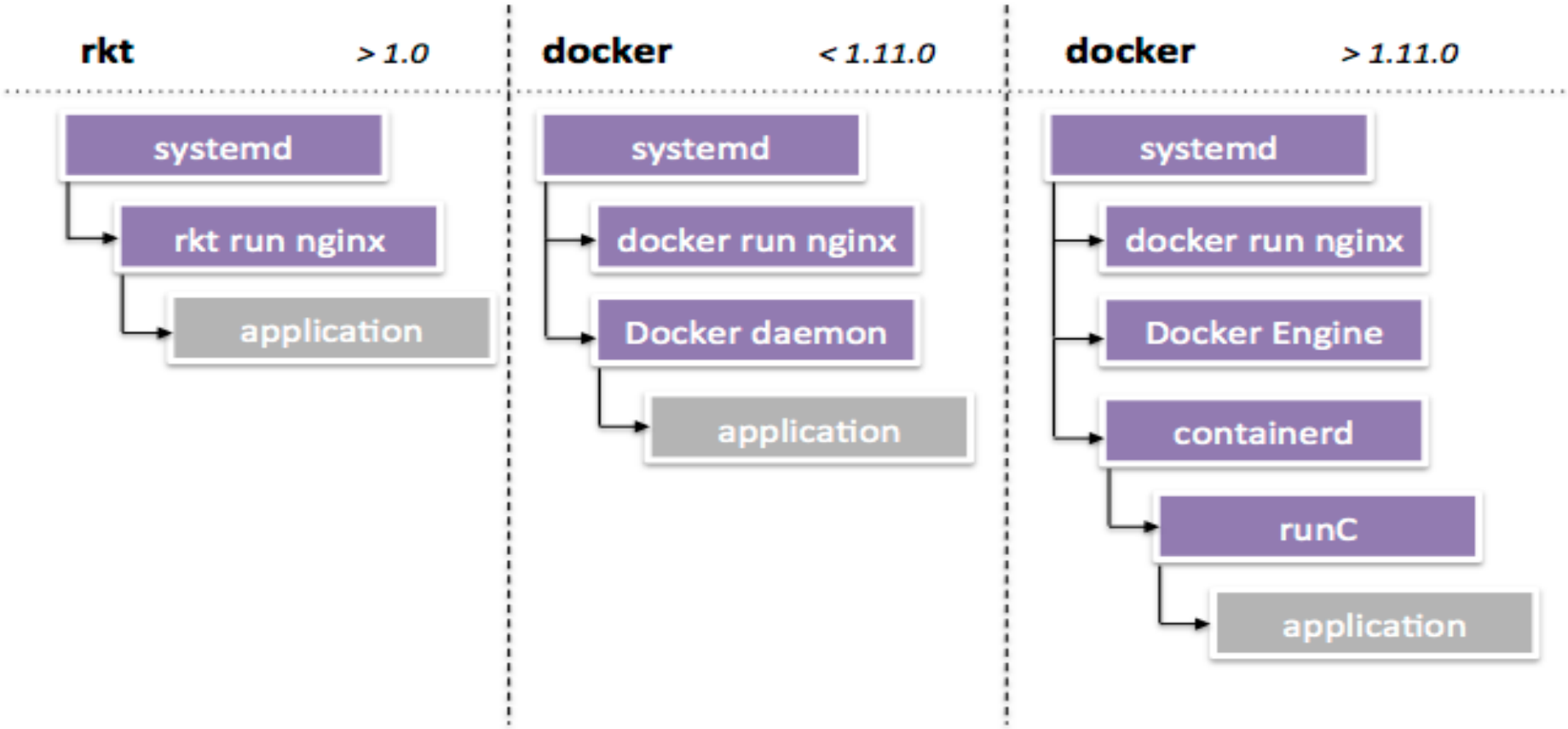
Containers in the cloud

- **rkt (2014):**
  - rkt implements App Container (appc) – an image-based runtime
  - But rkt also compatible with Docker images





GitHub - appc/spec: App Container Specification and Tooling (archived, see https://github.com/rkt/rkt/issues/4024)

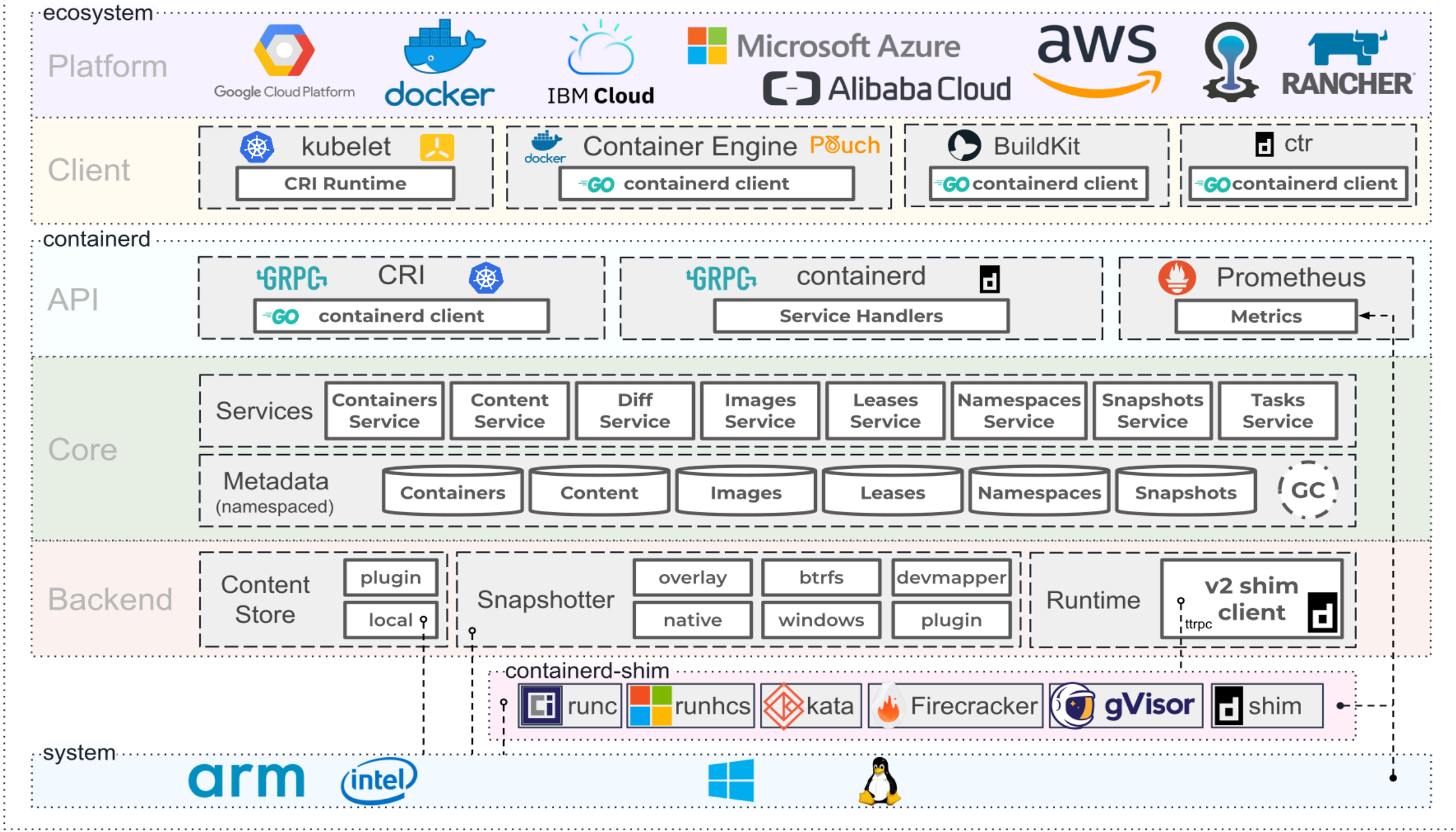


systemd - init system and manager - first daemon to start and last daemon to terminate

containerd - handles low-level details of Docker - complete container lifecycle from image transfer and storage to container execution and supervision to low-level storage to network attachments and beyond.

https://containerd.io/

https://medium.com/@williamwarley/mastering-the-container-universe-the-ultimate-guide-to-harnessing-the-power-of-containerd-for-1d362681187e


ecosystem
Platform
Google Cloud Platform
docker
IBM Cloud
Microsoft Azure
Alibaba Cloud
aws
RANCHER
Client
kubelet
CRI Runtime
Container Engine
Pouch
containerd client
BuildKit
containerd client
ctr
containerd client
containerd
API
GRPC
CRI
containerd client
GRPC
containerd
Service Handlers
Prometheus
Metrics
Core
Services
Containers Service
Content Service
Diff Service
Images Service
Leases Service
Namespaces Service
Snapshots Service
Tasks Service
Metadata
(namespaced)
Containers
Content
Images
Leases
Namespaces
Snapshots
GC
Backend
Content Store
plugin
local
Snapshotter
overlay
btrfs
devmapper
native
windows
plugin
Runtime
v2 shim client
ttrpc
containerd-shim
runc
runhcs
kata
Firecracker
gVisor
shim
system
arm
intel

Built to be modular and service-oriented

Open Container Initiative - Open Container Initiative (opencontainers.org)

- ***Daemon:*** `containerd` daemon runs in background: manage life cycles, image transfer and storage, network configs, runtime tasks.

- ***Client-Server Architecture:*** `containerd` acts as server: any # clients can interact through gRPC API calls.

gRPC API: Primary method for interacting with `containerd`,

Container Execution and Supervision: Execute containers according to OCI

Image Management: Handles the downloading, storage, and distribution of container images.

Storage and Volume Management: Integrates with the underlying host system to manage storage layers and volumes for containers,

Networking: Through plugins and integrations, configures and manages network settings for containers - ensures their communication

Security Features: several security mechanisms: namespace isolation and support for container-specific security profiles

1. ***Client Invocation:*** A client (CLI) makes gRPC call to `containerd` to perform an operation, such as creating a new container.

2. ***Processing the Request:*** `containerd` receives request and processes: pull image from a registry, prepare container's filesystem, set up network interfaces, startthe container using container runtime interface (CRI).

3. ***Container Execution:*** Once started, `containerd` supervises container execution - logging, resource allocation, and lifecycle management.

4. ***Response and Management:*** `containerd` responds to client with result

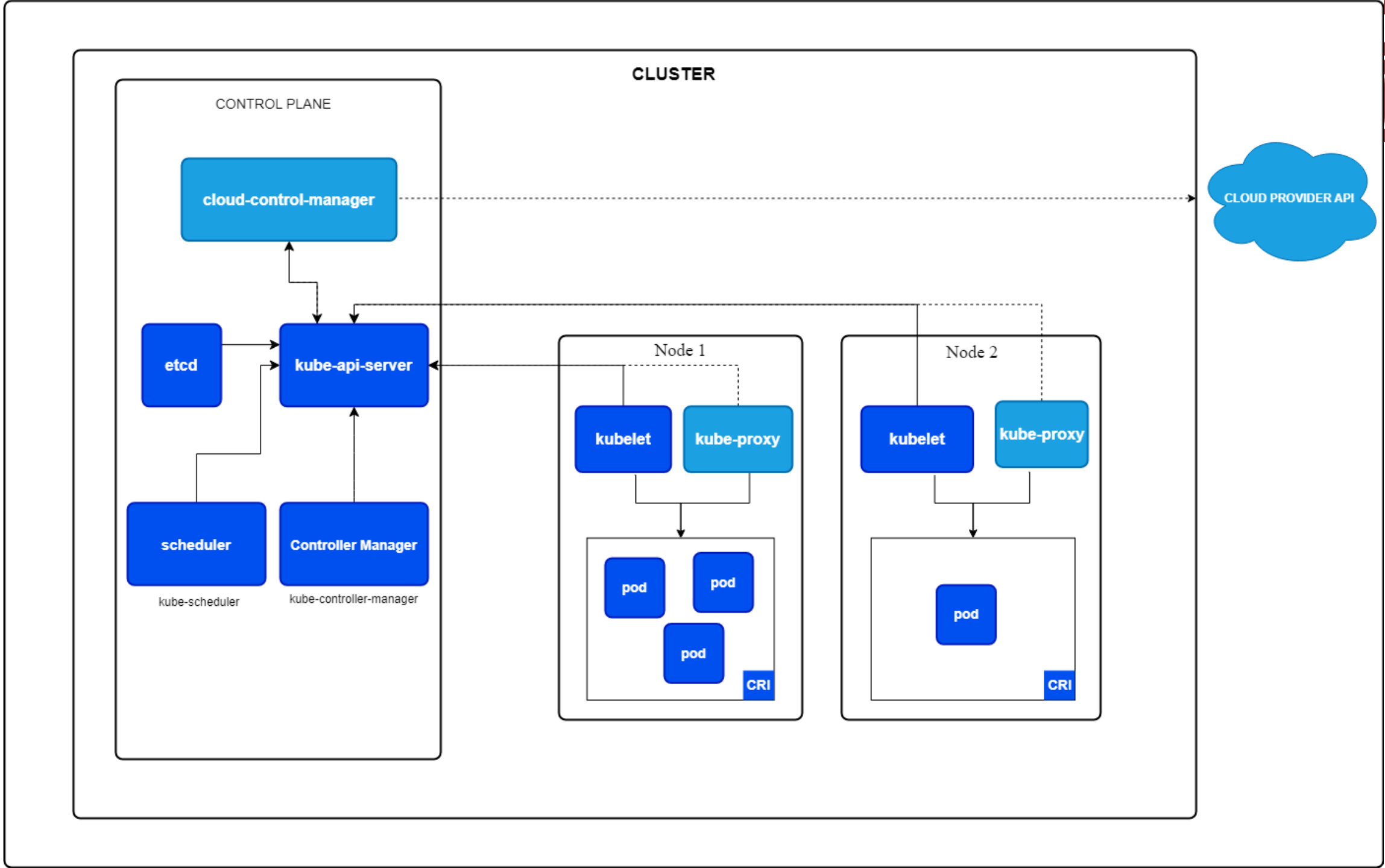
https://kubernetes.io/docs/concepts/overview/

Containers in the Cloud

https://queue.acm.org/detail.cfm?id=2898444

- Kubernetes (K8s): open-source system for automating deployment, scaling, and management of containerized applications
- It groups containers that make up an application into logical units for easy management and discovery.
- Builds upon 15 years of running production workloads at Google
- Provides a framework to run distributed systems resiliently.
- Takes care of scaling and failover for your application, provides deployment patterns, and more.

https://kubernetes.io/

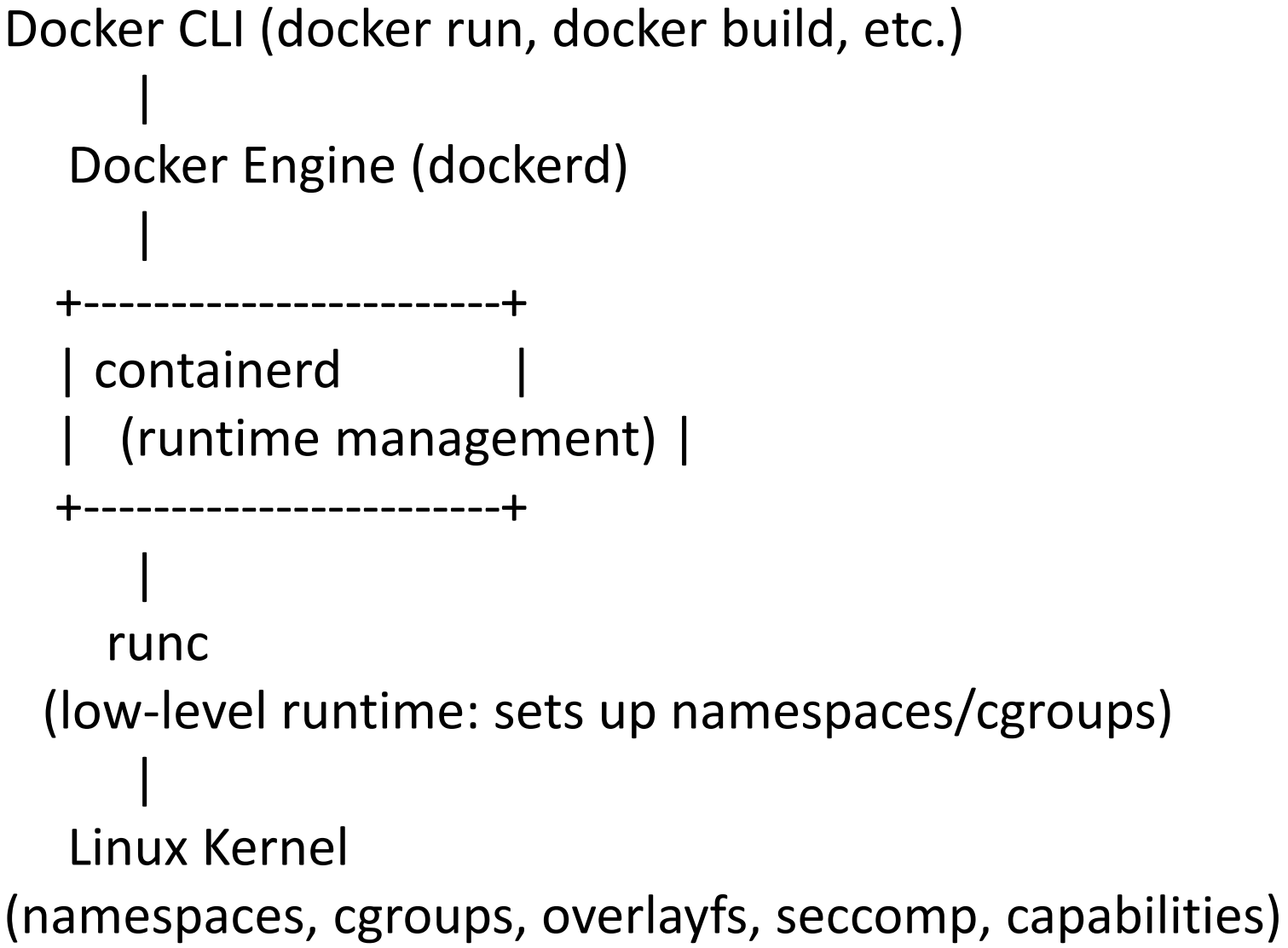

CLUSTER
CONTROL PLANE
cloud-control-manager
CLOUD PROVIDER API
etcd
kube-api-server
scheduler
Controller Manager
kube-scheduler
kube-controller-manager
Node 1
kubelet
kube-proxy
pod
pod
pod
CRI
Node 2
kubelet
kube-proxy
pod
CRI

Docker

- Docker is an open platform that developers use to package their applications with all dependencies, such as libraries, and configuration files, into a standardized unit, called a container.
- Docker is also a virtualization technology.
- By deploying applications in containers, developers can run their applications in any hosting system
- Docker runs on top of the host OS; hence, Docker is lightweight and fast.

Docker

- Package applications with all dependencies, such as libraries, and configuration files, into containers.
- Lightweight
- Portable
- Fast startup
- Standard workflows (build -> ship -> run)


Docker CLI (docker run, docker build, etc.)
Docker Engine (dockerd)
containerd
(runtime management)
runc
(low-level runtime: sets up namespaces/cgroups)
Linux Kernel
(namespaces, cgroups, overlayfs, seccomp, capabilities)

Example: docker run nginx

CLI sends request to **dockerd**.

dockerd asks **containerd** to create a container.

containerd pulls the **image** if not available locally.

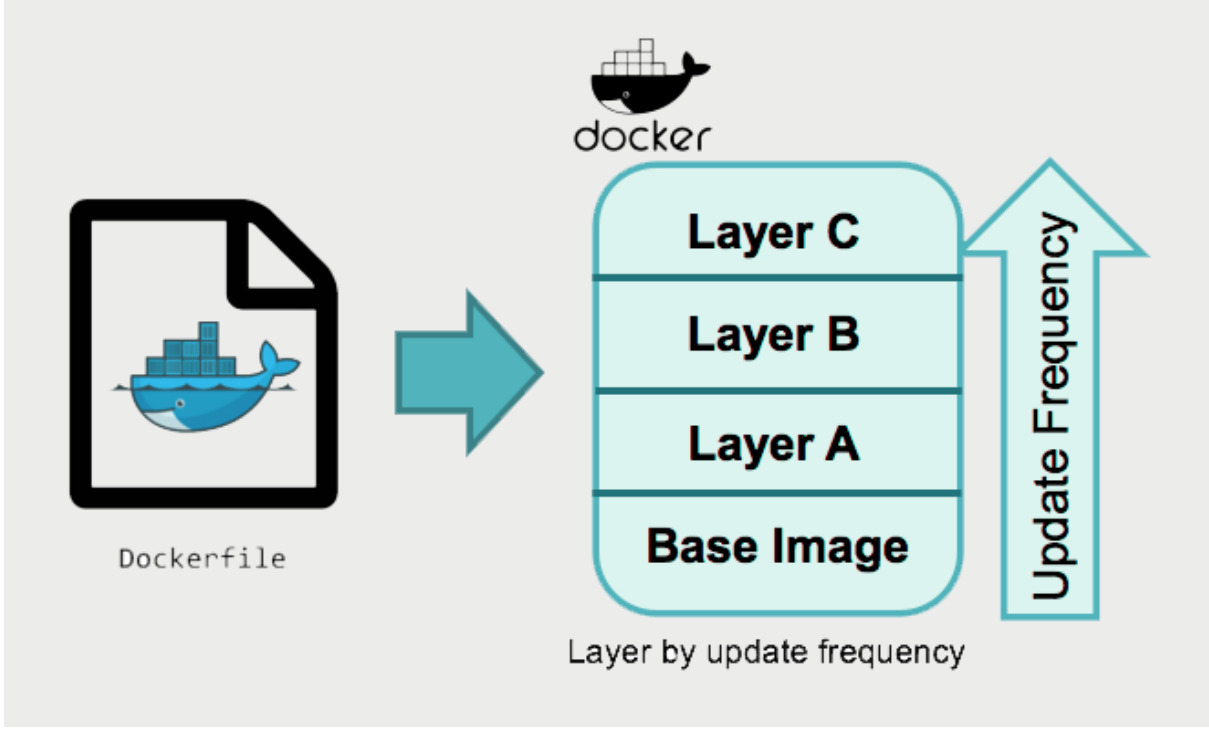
containerd calls **runc** to:

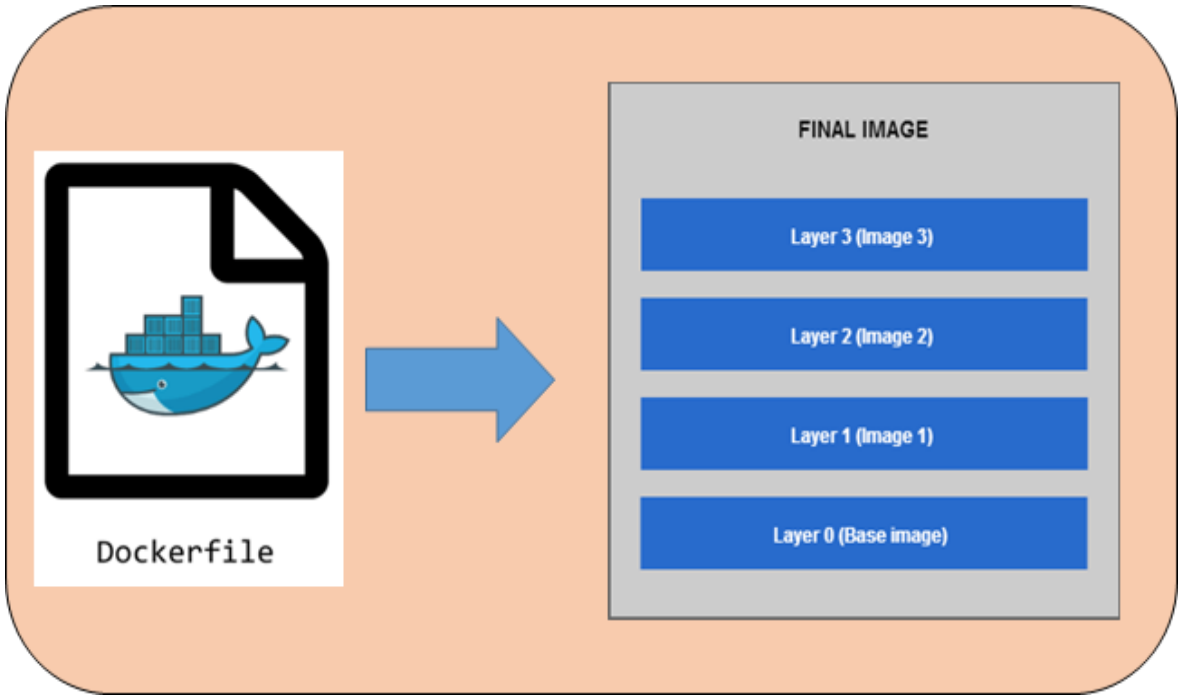
- Set up namespaces + cgroups.
- Mount image layers.
- Start container process.

Container is now running → isolated, but shares kernel.

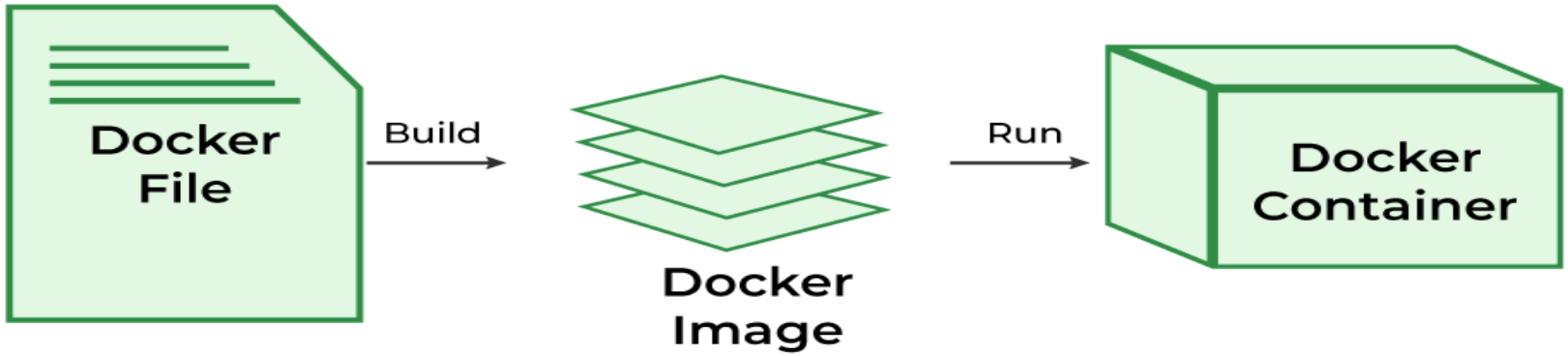
Docker Image

- A “template” for running a container
- Just code executed in the container
- Pull – instantiate – run container
- Each file in an image is called a layer
- Can contain many layers – a base image and other child images
- Each layer represents one service – programs depend on programs
- Layer: installers, app code, dpndncs.
- If you change one layer, you need to rebuild the whole image
- Increases the reusability and decreases disk usage.





Dockerfile



- DSL (Domain Specific Language): instructions for generating an image.
- A text doc with commands which execute to assemble image.
- Defines processes to quickly produce an image - Create your Dockerfile in order since daemon runs instructions from top to bottom.

Dockerfile - myhttpd:0.1

```
1  # A simple web app served by httpd
2  FROM httpd:2.4
3
4  LABEL AUTHOR=user@example.com
5
6  LABEL VERSION=0.1
7
8  # COPY mypage.html /usr/local/apache2/htdocs/mypage.html
9  # WORKDIR /usr/local/apache2
10
11 COPY mypage.html htdocs/mypage.html
```

```
# A simple wget container

FROM alpine

LABEL VERSION=0.1 \
      AUTHOR=LMCS \
      EMAIL=lmcsdeveloper@gmail.com

RUN apk update \
 && apk add wget \
 && rm -rf /var/cache/apk/*

WORKDIR /root/

ENTRYPOINT [ "wget" ]

CMD [ "--help" ]

```

wget

- `wget http://example.com/`
- `wget -c https://releases.ubuntu.com/20.04/ubuntu-20.04-desktop-amd64.iso`
- `docker build -t dget .`
- `docker run --rm dget http://example.com/`
- `alias dget='docker run --rm dget'`
- `dget http://example.com/`

Command	Purpose	Example
FROM	Base image to build upon	FROM ubuntu:22.04
RUN	Execute a command in a new layer (often install software)	RUN apt-get update && apt-get install -y curl
CMD	Default command to run when the container starts	CMD ["python3", "app.py"]
ENTRYPOINT	Similar to CMD, but fixes the main program (CMD can override arguments)	ENTRYPOINT ["python3"]
COPY	Copy files from local machine into image	COPY app/ /app/
ADD	Like COPY, but also allows URLs and archives	ADD https://example.com/file.tar.gz /files/
WORKDIR	Set the working directory inside container	WORKDIR /app
ENV	Set environment variables	ENV PORT=5000
EXPOSE	Document a port the app listens on (for networking)	EXPOSE 8080
VOLUME	Declare a mount point for persistent data	VOLUME /data
USER	Set which user the container runs as	USER 1001
LABEL	Metadata (author, version, description)	LABEL maintainer="teacher@example.com"
ARG	Build-time variable (used with --build-arg)	ARG version=1.0

Base Image + Default Command

```
FROM alpine:3.19
CMD ["echo", "Hello, World!"]
```

Installing Packages in your Image

```
FROM ubuntu:22.04
RUN apt-get update && apt-get install -y curl
CMD ["bash"]
```

Copy Code in your Container and Run it

```
FROM python:3.11-slim
WORKDIR /app
COPY app.py .
CMD ["python", "app.py"]
```

Environment Variables

```
FROM node:20-alpine
WORKDIR /usr/src/app
COPY . .
RUN npm install
ENV PORT=3000
EXPOSE 3000
CMD ["npm", "start"]
```

Copy project folder
Define env variable
Container will listen on 3000
Start node.js app

ADD and ENTRYPOINT

```
FROM alpine:3.19
WORKDIR /tools
ADD https://example.com/tool.tar.gz /tools/
ENTRYPOINT ["sh"]
CMD ["-c", "echo 'Tool downloaded and ready!'"]
```

Downloads and extracts a file from the internet.
Makes sure the container always starts with sh
Default argument passed to shell.

LABELS and ARG (for documentation etc.)

```
FROM ubuntu:22.04
LABEL maintainer="teacher@example.com"
ARG VERSION=1.0
RUN echo "Building version $VERSION"
CMD ["bash"]
```

Add a build-time variable

Users and Volumes

```
FROM alpine:3.19
RUN adduser -D student
USER student
WORKDIR /data
VOLUME /data
CMD ["sh", "-c", "echo 'Data will persist in volume!'"]
```

Create a new user inside container.
Run everything as this user (not root).
/data = persistent storage
Prints message about data persistence.

Small Images by Separating Build and Runtime

```
# Stage 1: Build
FROM golang:1.22 AS builder
WORKDIR /src
COPY . .
RUN go build -o myapp

# Stage 2: Runtime
FROM alpine:3.19
WORKDIR /app
COPY --from=builder /src/myapp .
CMD ["./myapp"]
```

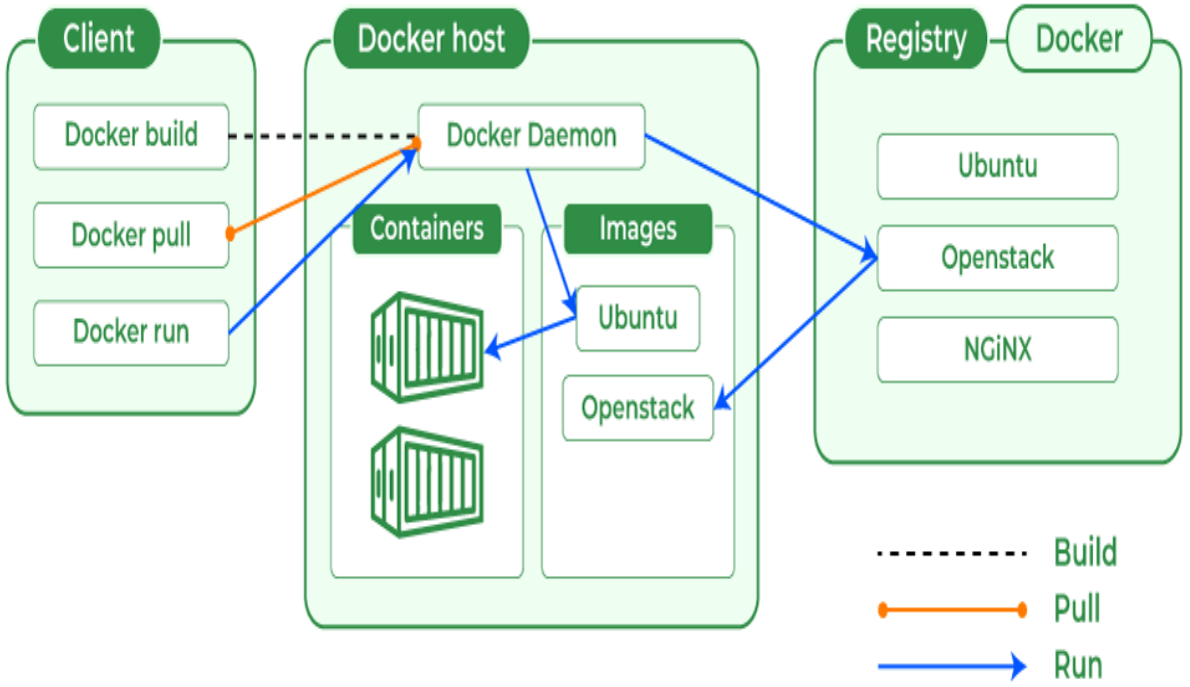
Stage 1: Compiles an app
Stage 2: Copies app in another container


```
FROM python:3.11-slim AS base
LABEL maintainer="teacher@example.com"
ARG APP_VERSION=1.2.3
ENV APP_ENV=production \
    PORT=8080

WORKDIR /app
COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .

EXPOSE 8080
VOLUME /app/data

USER 1001
ENTRYPOINT ["python", "app.py"]
CMD ["--port=8080", "--env=production"]
```

Web App with Full Features

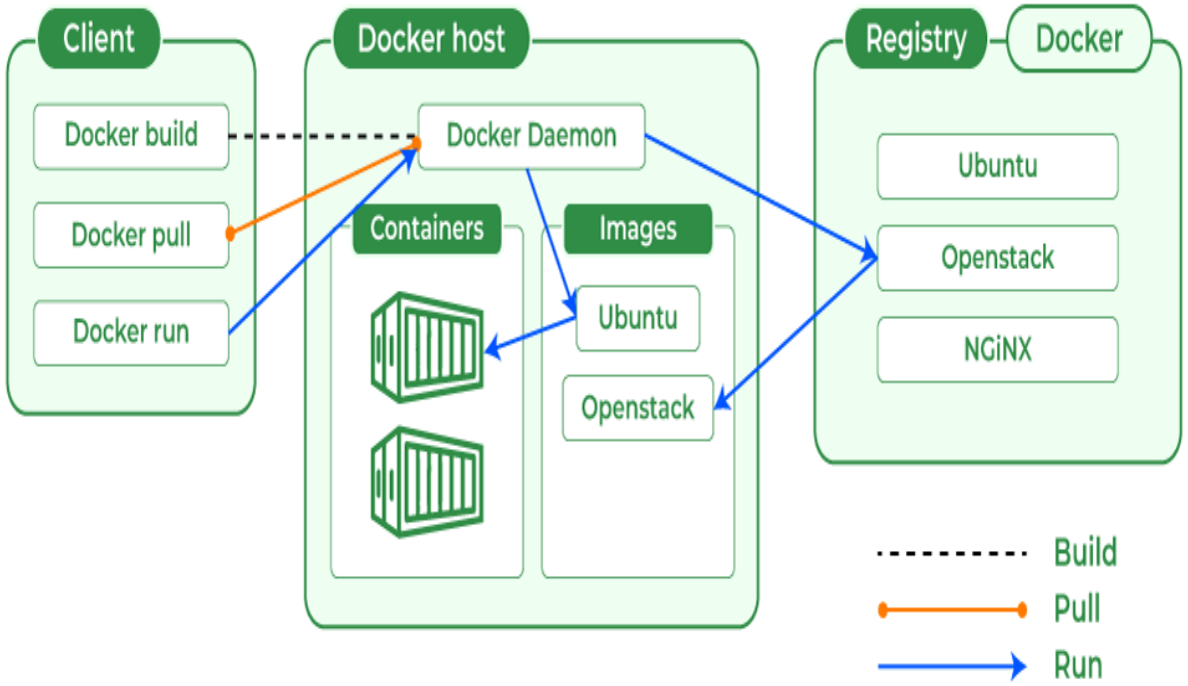
Docker Architecture

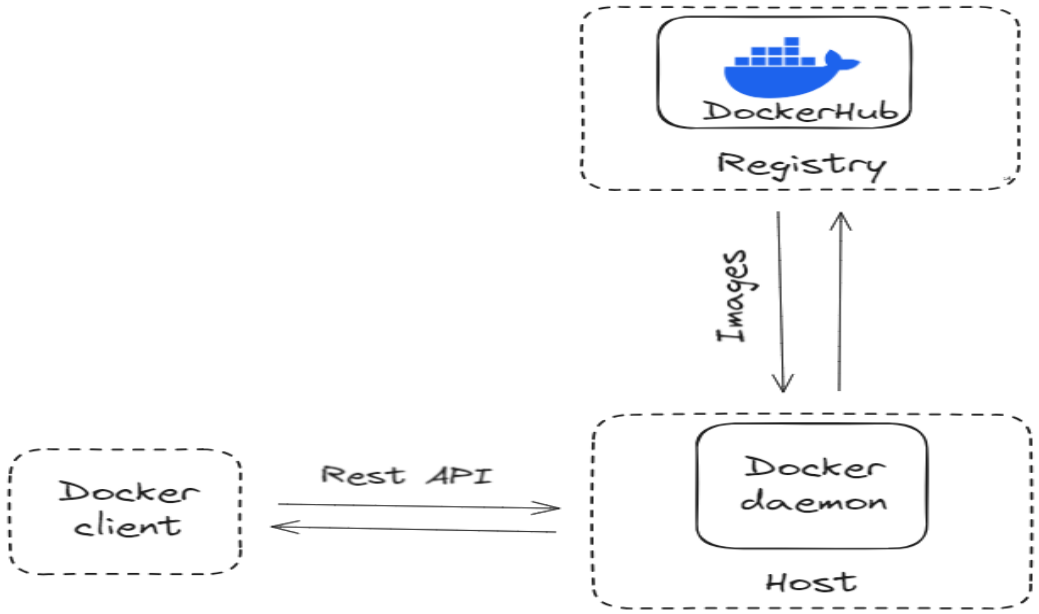
- Client-server architecture.
- Client talks with daemon
- Daemon builds, runs containers.
- Client runs with daemon on the same system (can connect client with daemon remotely).
- With REST API over a UNIX socket or a network, client and daemon interact.



Docker Dameon

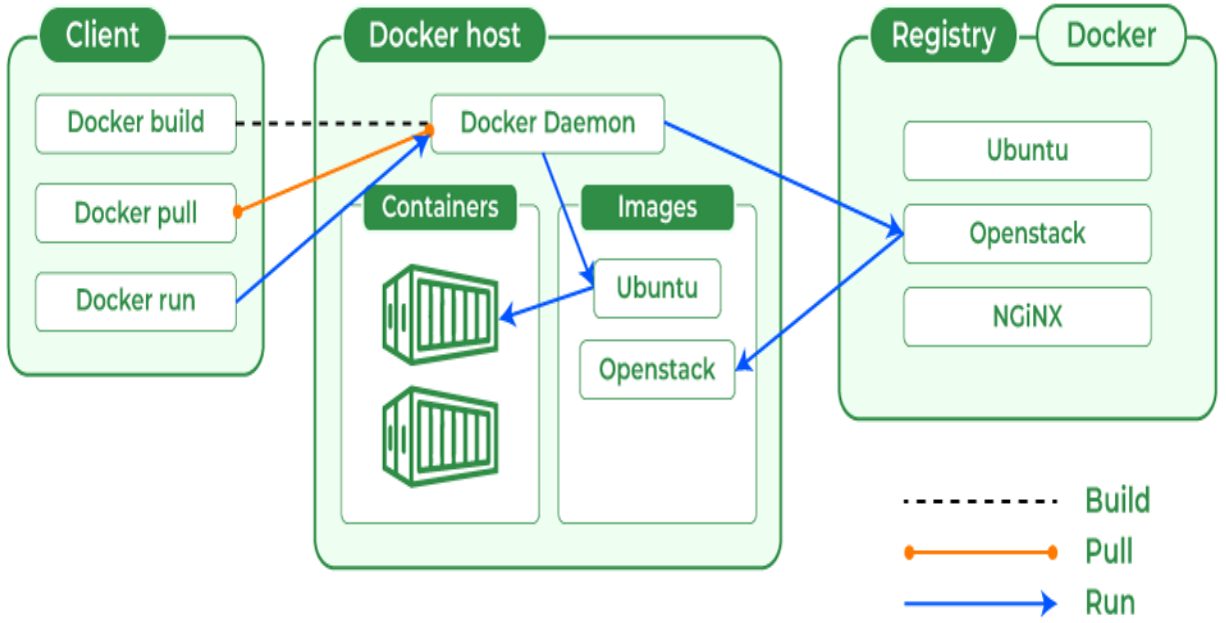
- Docker daemon manages all the services by communicating with other daemons.
- It manages docker objects such as images, containers, networks, and volumes with the help of the API requests of Docker.

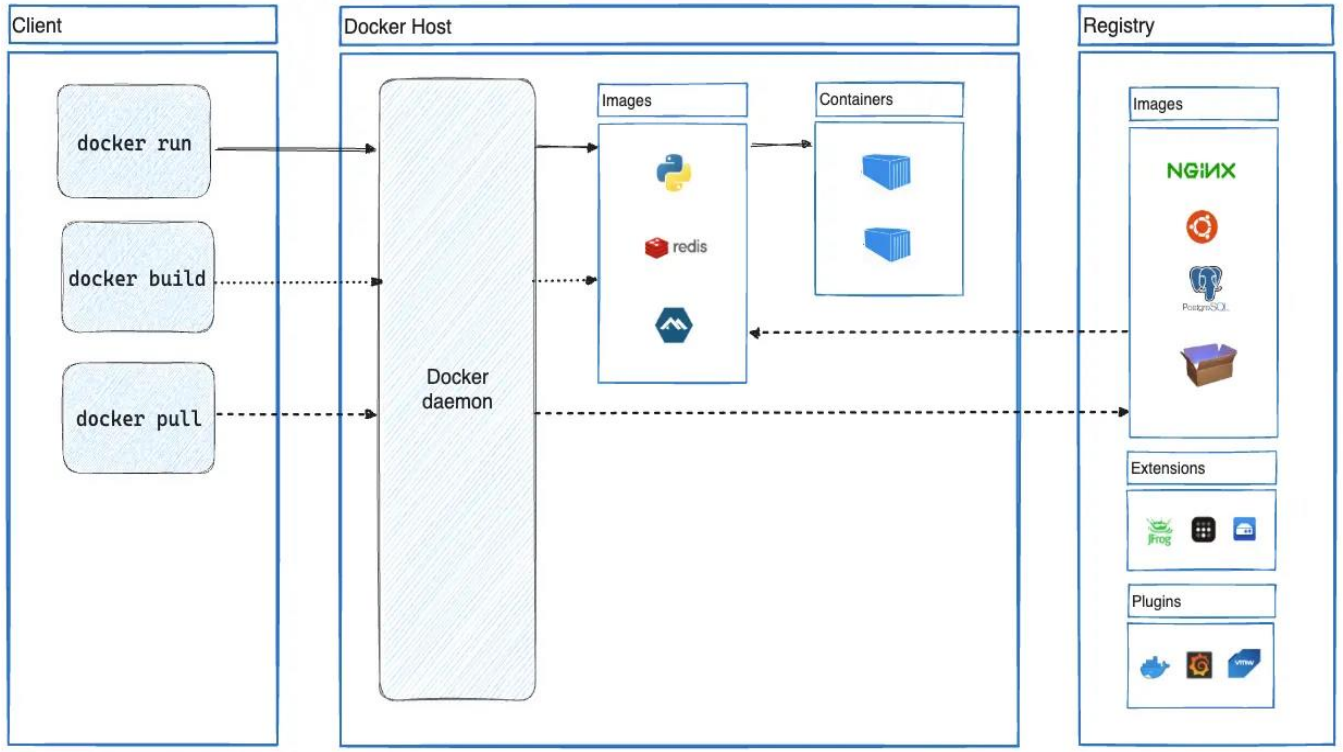




Docker Client

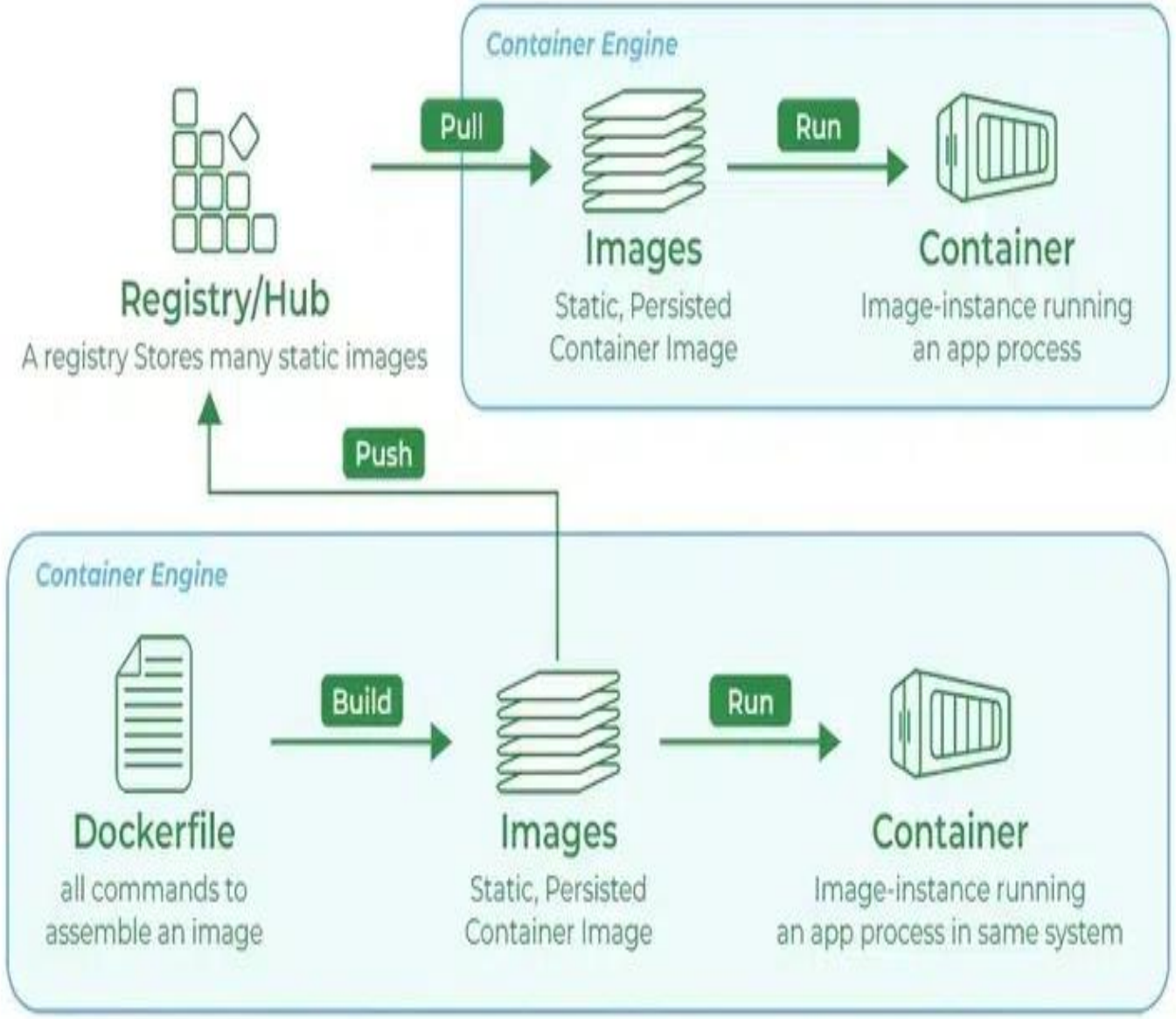
- Docker users interact with docker.
- The docker command uses the Docker API.
- When a docker client runs any docker command, terminal sends instructions to the daemon.
- Daemon gets it as command and REST API's request.
- Provide a way to direct the pull of images from the docker registry and run them on the docker host.
- The common commands which are used by clients are docker build, docker pull, and docker run.
- The Docker client can communicate with multiple daemons.





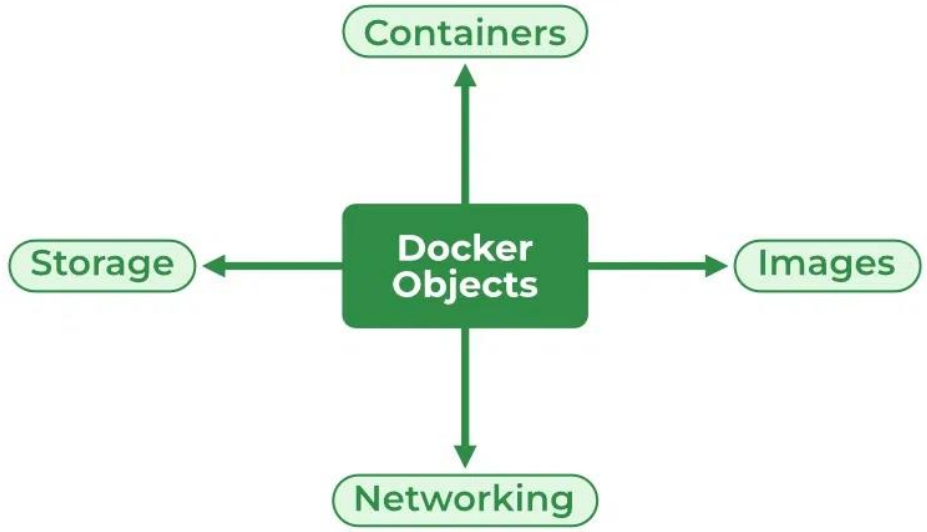
Going Further

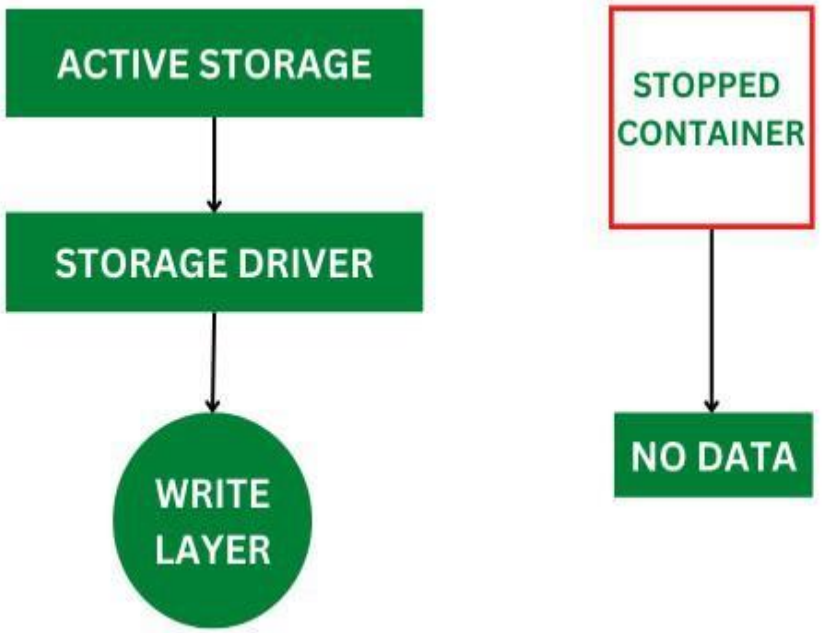
- Docker Host: a machine that is responsible for running more than one container.
  - Comprises Docker daemon, Images, Containers, Networks, and Storage.
- Docker Registry: All docker images are stored in the docker registry.
  - There is a public registry (docker hub) that can be used by anyone.
  - Private registry possible
  - docker run or docker pull: pull required images from registry
  - docker push: push images into configured registry



Going Further

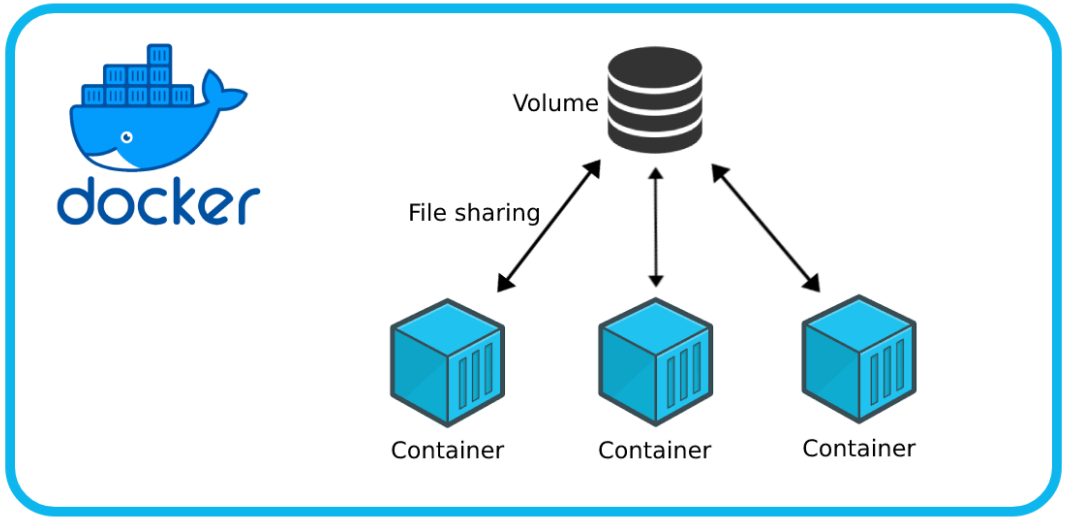
- Docker Objects
- Whenever we are using a docker, we are creating and use images, containers, volumes, networks, and other objects.
- We can store data within the writable layer of the container, but it requires a storage driver.
- Storage driver controls and manages the images and containers on our docker host.

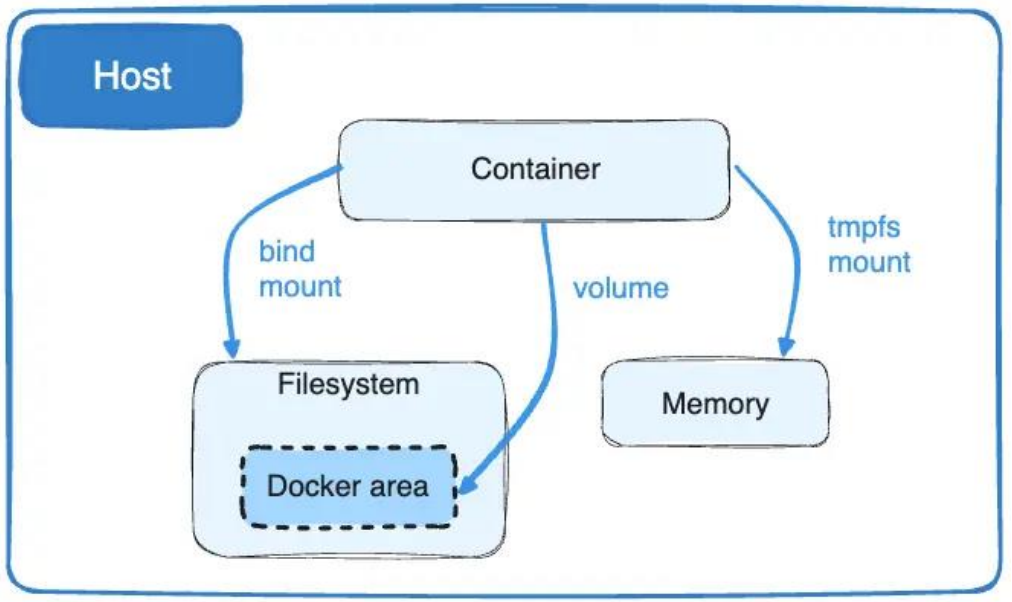




Types of Docker Storage

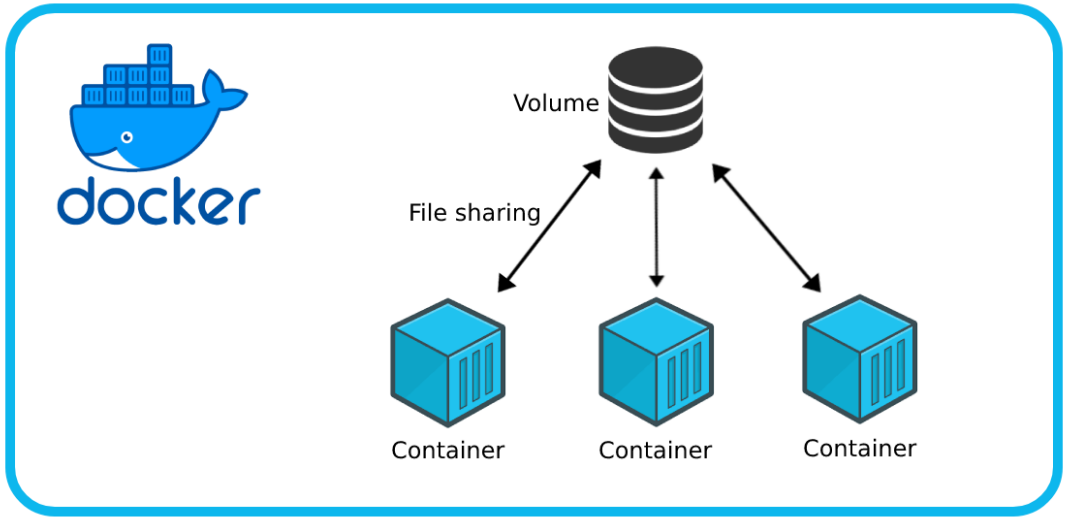
- Data Volumes: Can be mounted directly into the filesystem of the container and are essentially directories or files on the Docker Host filesystem.
- Provide a way to persist data generated and used by Docker containers.
- Unlike the container's writable layer, which is ephemeral and gets destroyed when the container stops, volumes exist outside of container's lifecycle.
- Data stored in volumes remains intact even when the container is removed.

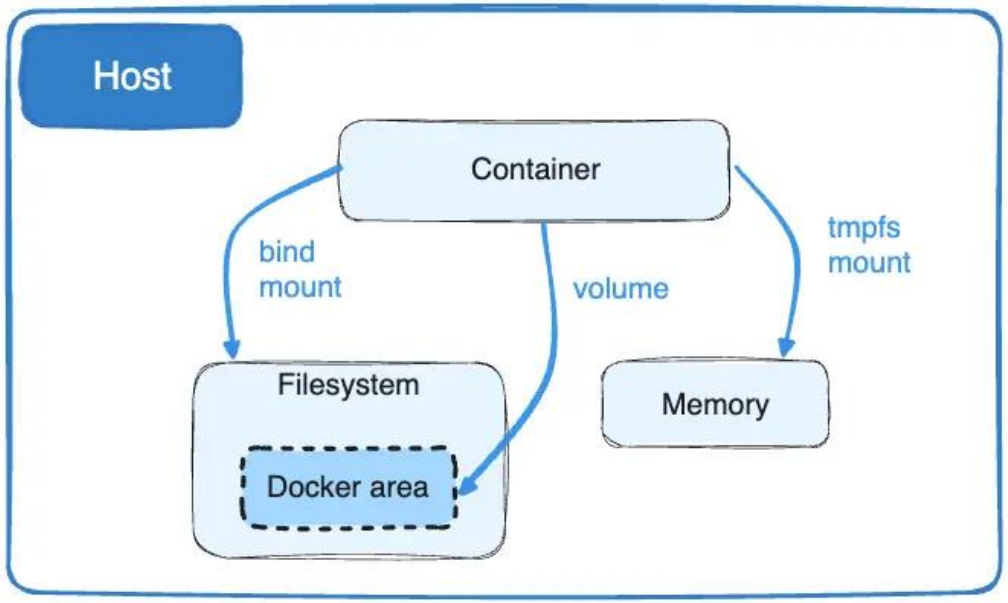




Types of Docker Storage

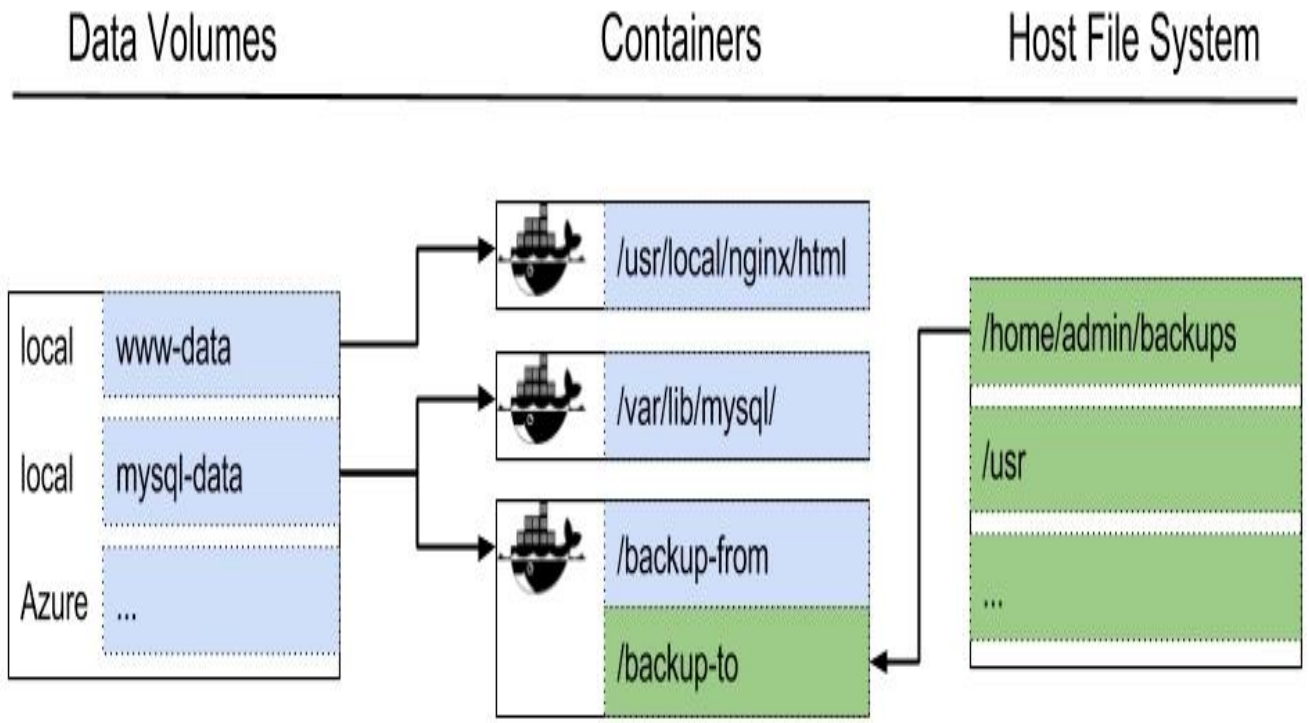
- Data Persistence: Volumes ensure that critical data persists across container restarts and updates - vital for databases, file storage, and other applications where data integrity is paramount.
- Sharing Data Between Containers: Volumes facilitate sharing data and collaboration between multiple containers
- Backup and Restore: Easily back up important data and restore it when needed, enhancing resilience



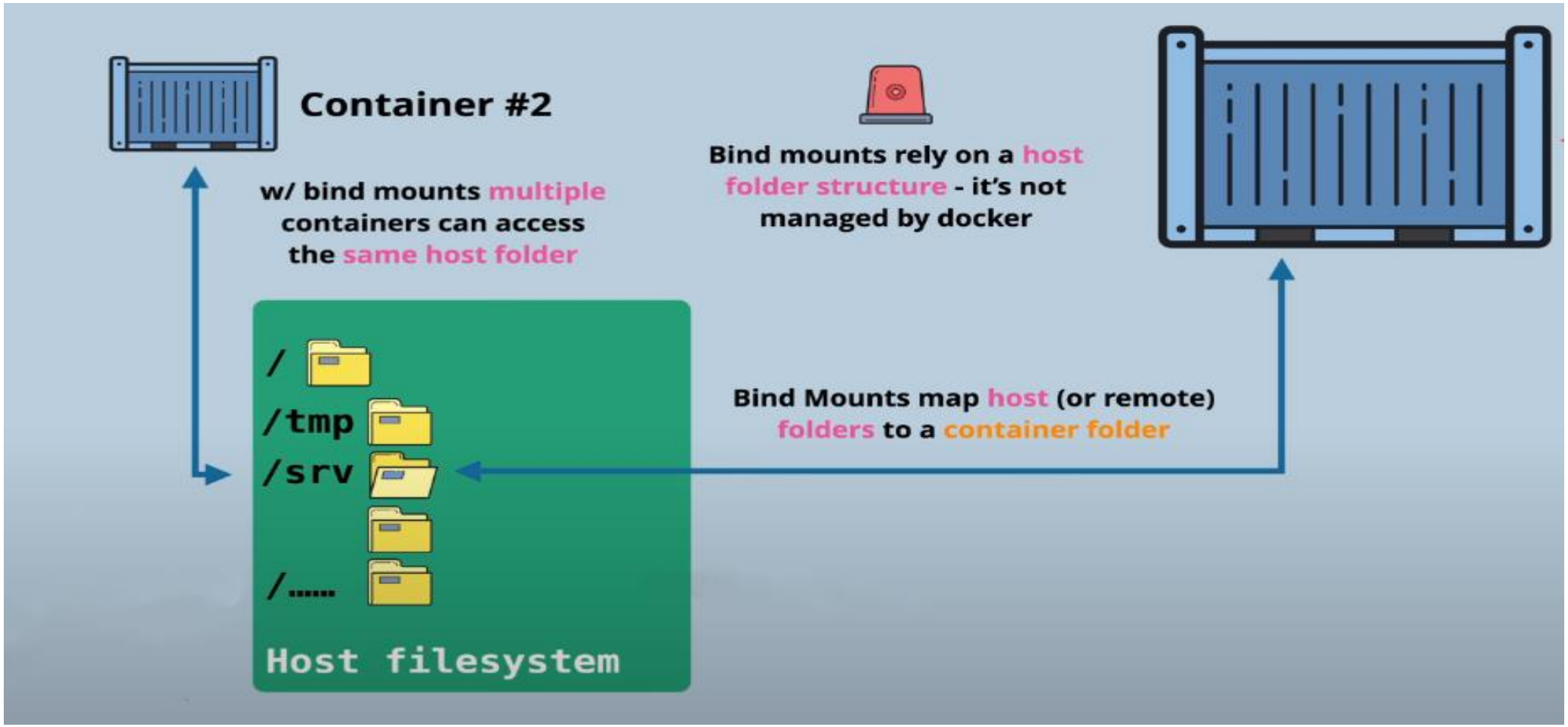


Types of Docker Storage

- Volume Container: To maintain the state of the containers (data) produced by the running container, Docker volumes file systems are mounted on Docker containers.
- Independent container life cycle
- Volumes stored on the host.
- Makes it simple for users to exchange file systems among containers and backup data.

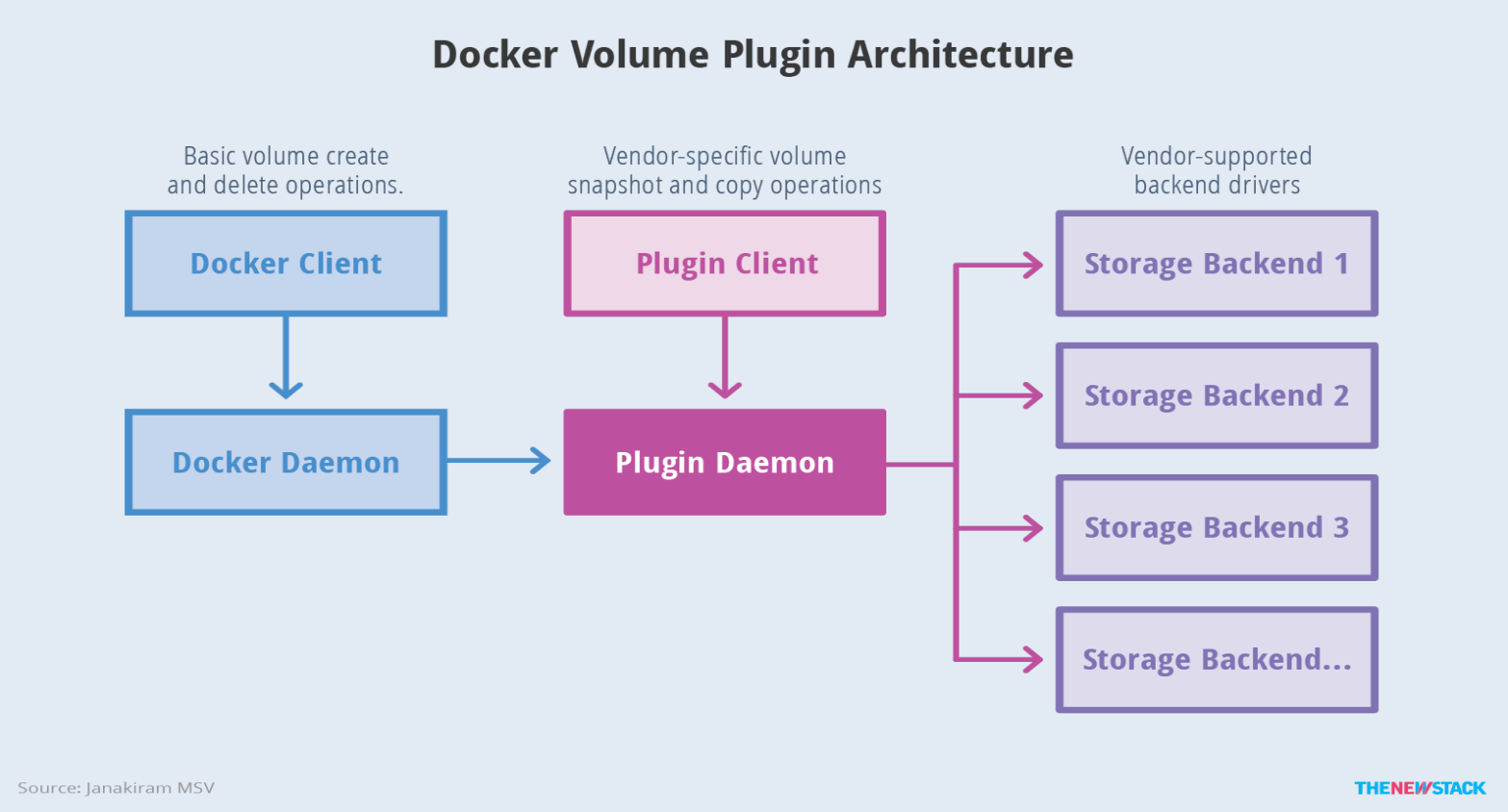


Types of Docker Storage



- Directory Mounts: A host directory that is mounted as a volume in your container

Types of Docker Storage



- Storage Plugins: Docker volume plugins enable us to integrate the Docker containers with external volumes like Amazon EBS by this we can maintain the state of the container.