2019

kubernetes

by ***Soojan***
WhatsApp : 09740165136
soojanknandy@gmail.com

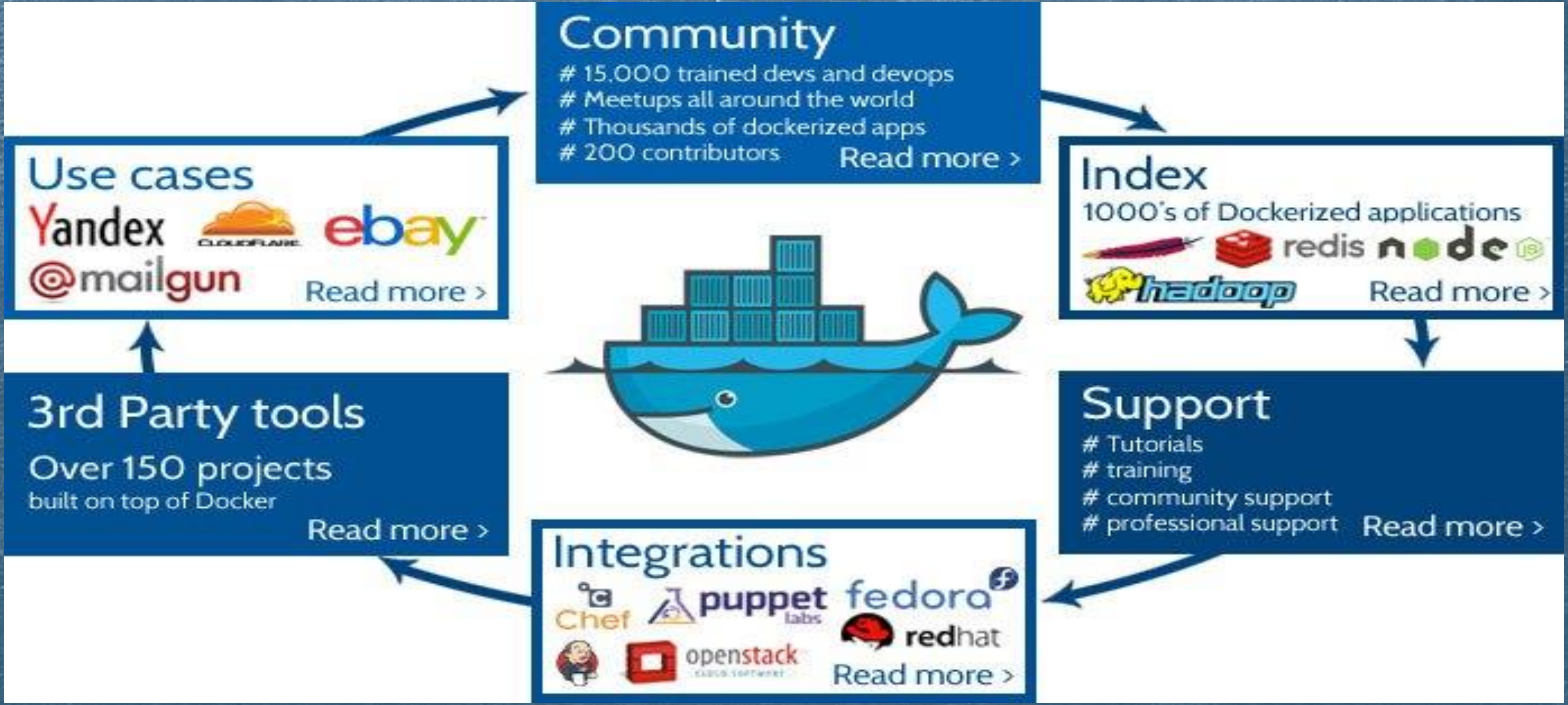
What is docker?

- Docker is a computer program that performs operating-system-level virtualization also known as containerization.
- Written in : Go, a programming language created at Google in 2009 by Robert Griesemer, Rob Pike, and Ken Thompson.

Docker : a bit more

- Release : 13th March, 2013 / 17th October, 2019
- Containers are isolated from each other and use their own set of tools and libraries.
- All containers use the same kernel & are created from "images" which specify their precise contents.
- Repository : github.com/docker/docker-ce
- Solomon Hykes | Andrea Luzzardi | Francois-Xavier Bourlet created docker while working for dotCloud, which later became Docker

Docker : ecosystem



Community
15,000 trained devs and devops
Meetups all around the world
Thousands of dockerized apps
200 contributors
Read more ›

Index
1000's of Dockerized applications
redis node hadoop
Read more ›

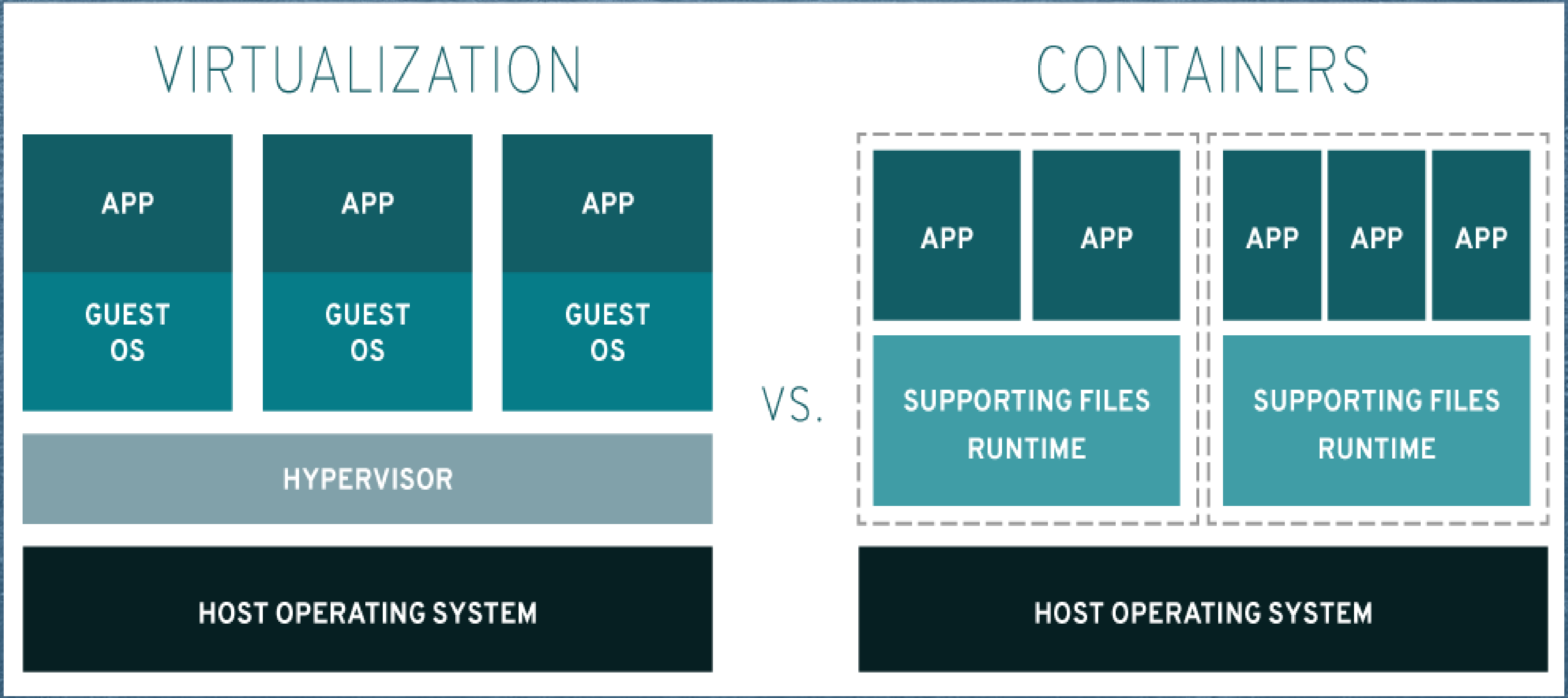
Support
Tutorials
# training
# community support
# professional support
Read more ›

Integrations
Chef puppet labs fedora redhat openstack
Read more ›

3rd Party tools
Over 150 projects
built on top of Docker
Read more ›

Use cases
Yandex CloudFlare ebay mailgun
Read more ›

Virtualization Vs Containerization



Containers are new?

- It has been built in to Linux in the form of lxc for almost 10 years, and similar operating system level virtualization has also been offered by freebsd jails, aix workload partitions and solaris containers. But being there for so many years containers were not talk of the town till docker.
- Virtualization includes an entire operating system as well as the application. A physical server running virtual machines has a hypervisor and 2 or more separate operating systems running on OS and then application on top of it.
- Containers like Docker can be run within a single virtual machine. Earlier system admins used to add one app per virtual machine. Now with containers you need not to create a virtual machines for every app. This means you can run multiple apps and you no longer need hundreds of VMs on one machine. Thus, containers are much more lightweight and use few resources than virtual machines.

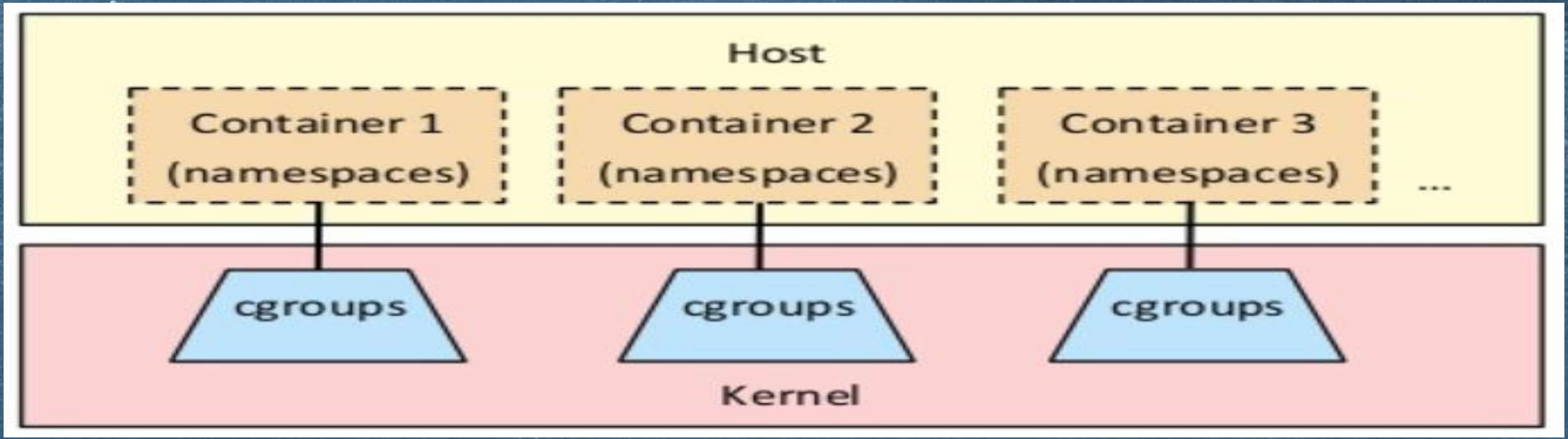
Lightweight VMs?

- does not boot a separate OS per VM and is therefore faster to start/stop
- requires less disk space due to sharing of common layers across images
- due to the image layering, incremental deployments of new app versions are smaller and therefore faster than VMs
- shares a kernel across containers and therefore uses less memory

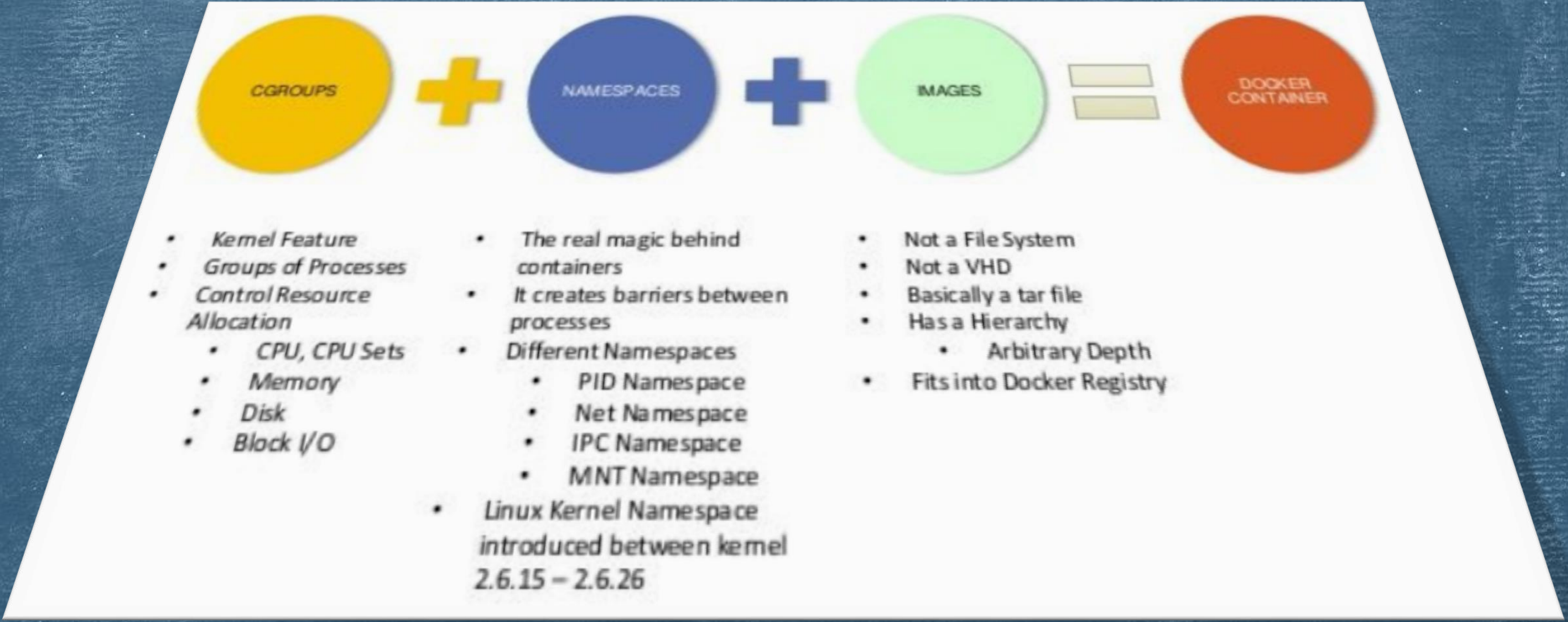
Docker and Linux container technology

- linux containers are self-contained execution environments.
- isolated CPU, memory, block I/O, and network resources

aufs cgroups namespaces iptables



Make Docker

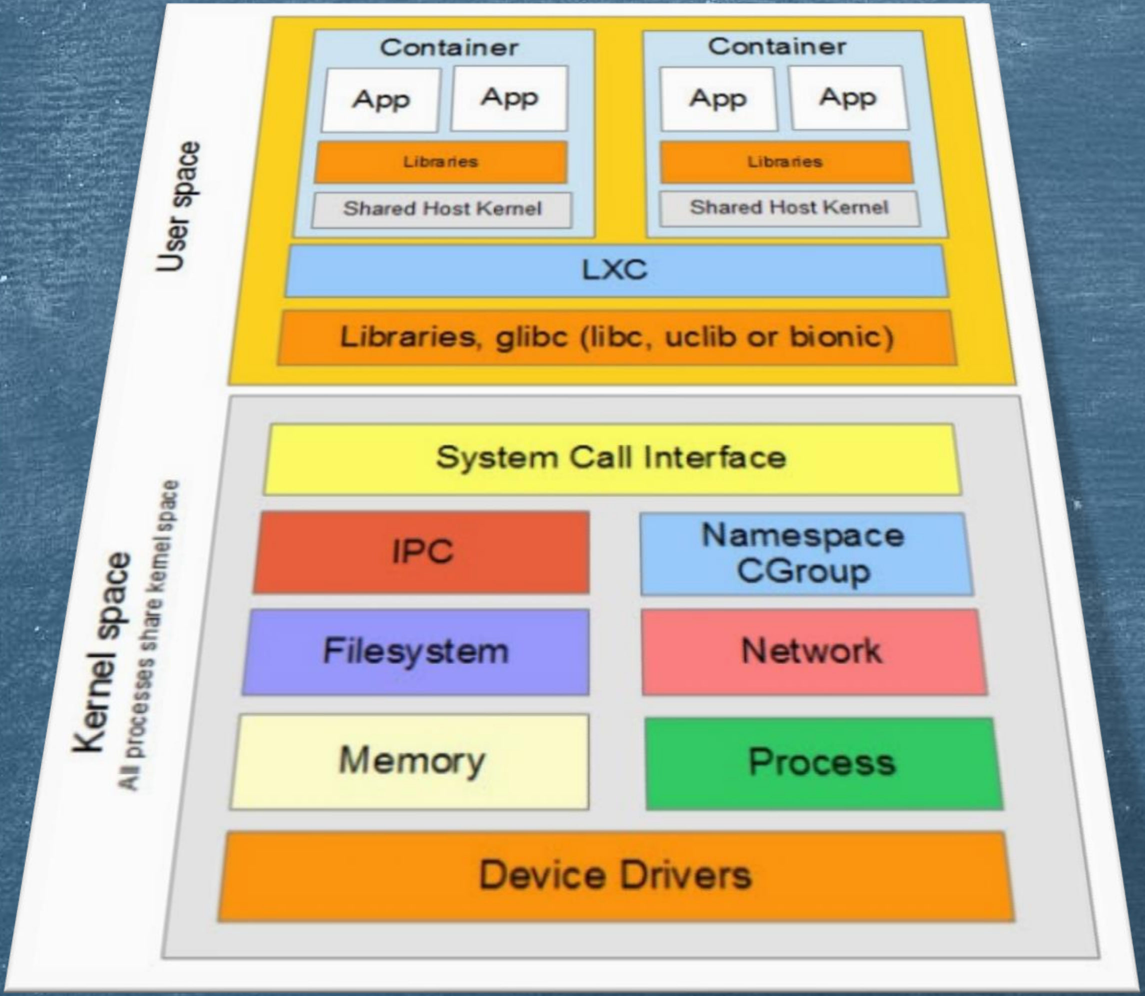


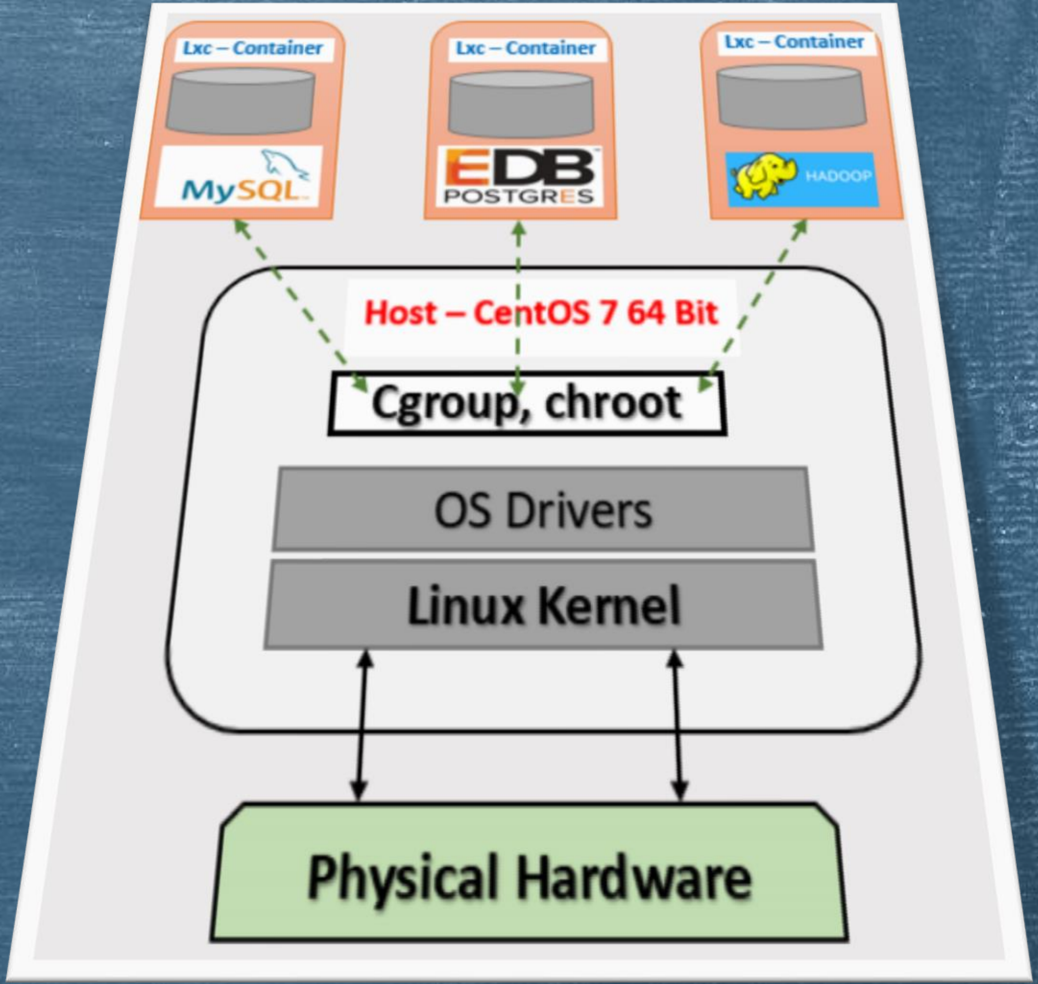
- *Kernel Feature*
- *Groups of Processes*
- *Control Resource Allocation*
  - *CPU, CPU Sets*
  - *Memory*
  - *Disk*
  - *Block I/O*

- The real magic behind containers
- It creates barriers between processes
- Different Namespaces
  - PID Namespace
  - Net Namespace
  - IPC Namespace
  - MNT Namespace
- Linux Kernel Namespace introduced between kernel 2.6.15 – 2.6.26

- Not a File System
- Not a VHD
- Basically a tar file
- Has a Hierarchy
  - Arbitrary Depth
- Fits into Docker Registry

How it works?


User space
Container
App
App
Libraries
Shared Host Kernel
Container
App
App
Libraries
Shared Host Kernel
LXC
Libraries, glibc (libc, uclib or bionic)
Kernel space
All processes share kernel space
System Call Interface
IPC
Namespace CGroup
Filesystem
Network
Memory
Process
Device Drivers


Lxc – Container
MySQL
Lxc – Container
EDB POSTGRES
Lxc – Container
HADOOP
Host – CentOS 7 64 Bit
Cgroup, chroot
OS Drivers
Linux Kernel
Physical Hardware

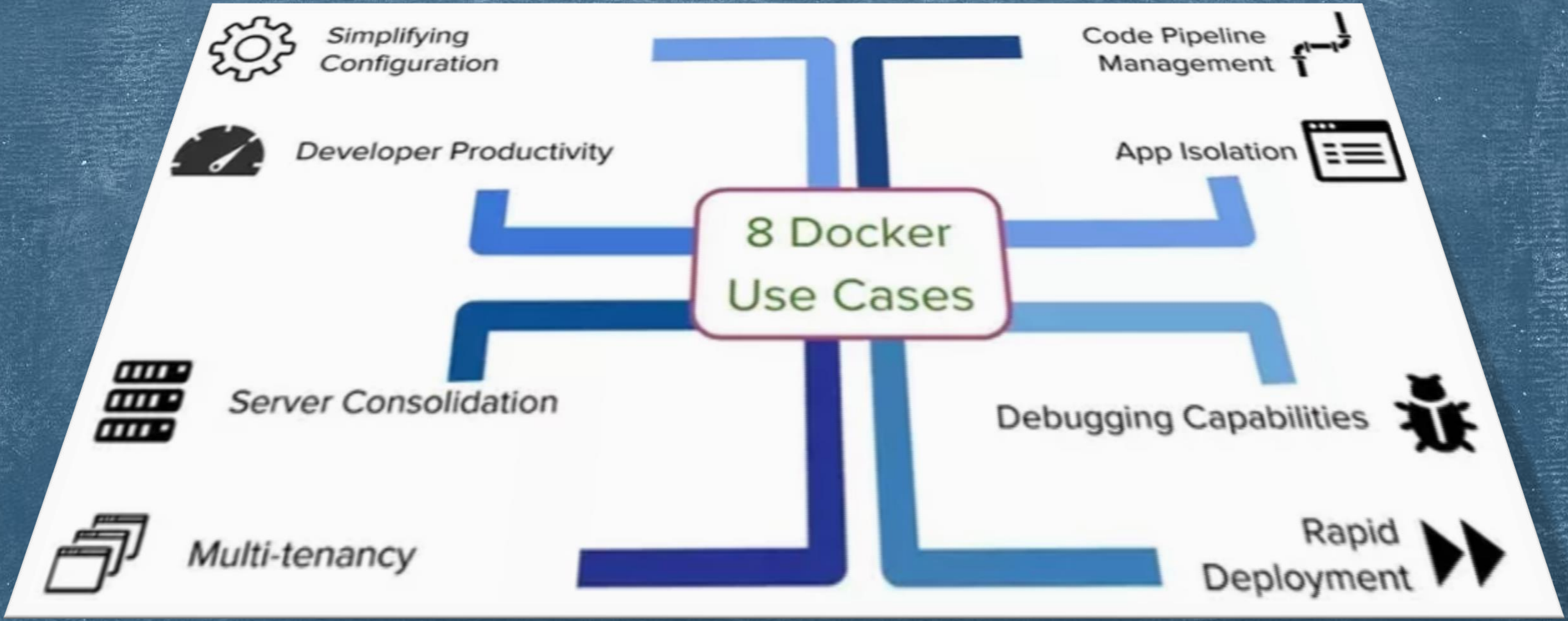
Docker Key Advantages

RoI & Cost Savings	Standardization & Productivity
reduced infrastructure requirements	repeatable dev, build, test, & prod environments
Compatibility & Maintainability	Simplicity & Faster Configurations
NO MORE "it works on my machine"	Infra requirements & app environment are delinked
Rapid Deployment	Continuous Deployment & Testing
Process based booting	maintain configurations & dependencies internally
Multi-Cloud Platforms	Isolation
Aws, azure, gcp others : YES	clean app removal alongwith with container

Security

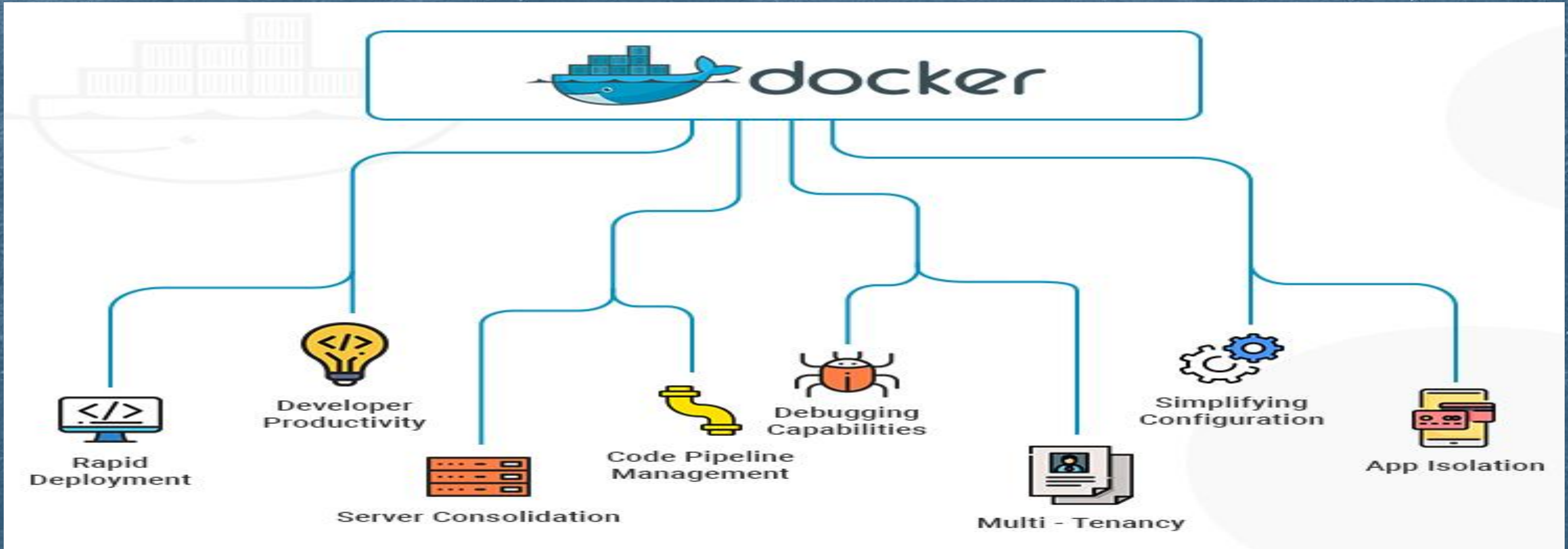
applications that are running on containers are completely segregated and isolated from each other, granting you complete control over traffic flow and management. No Docker container can look into processes running inside another container. From an architectural point of view, each container gets its own set of resources ranging from processing to network stacks.

Docker Key Advantages

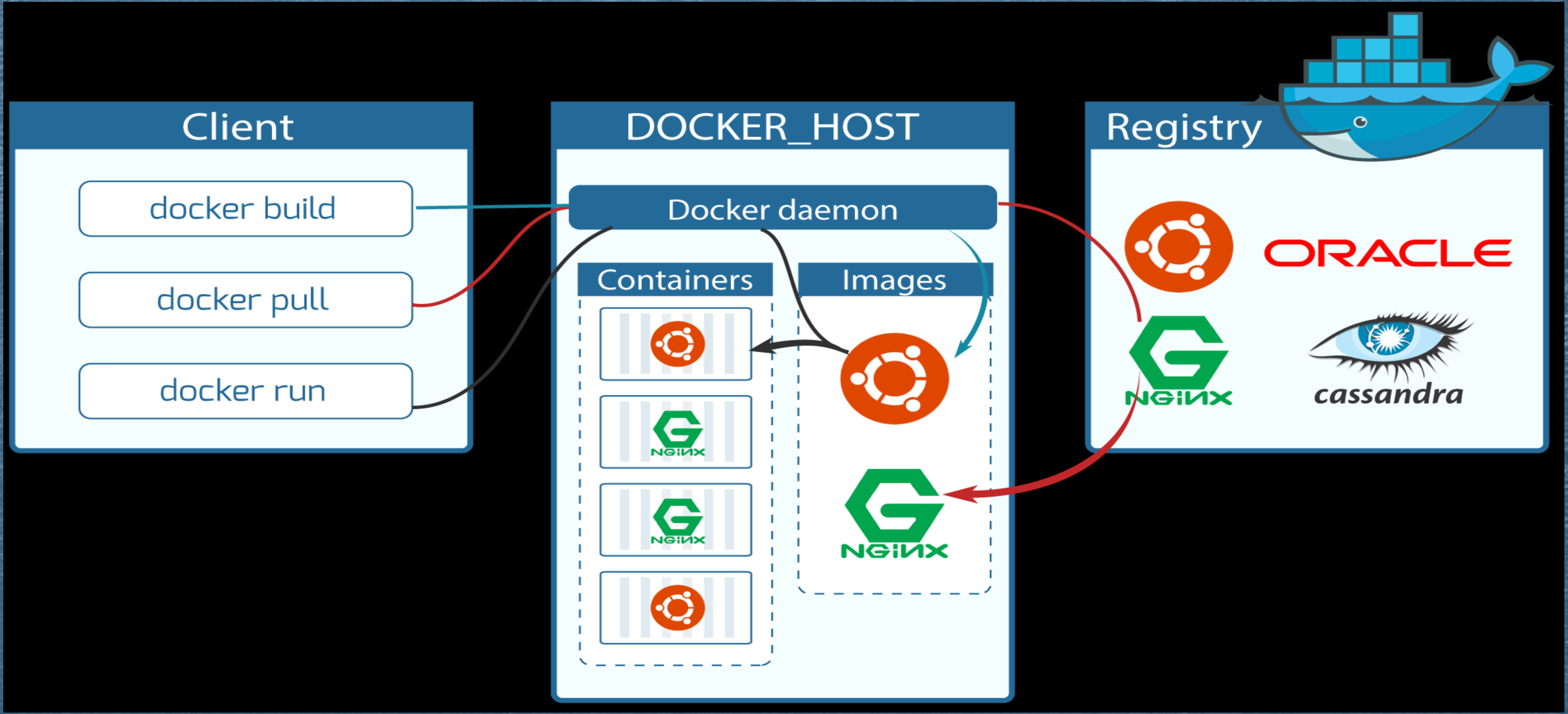


Docker Key Advantages a bit more..

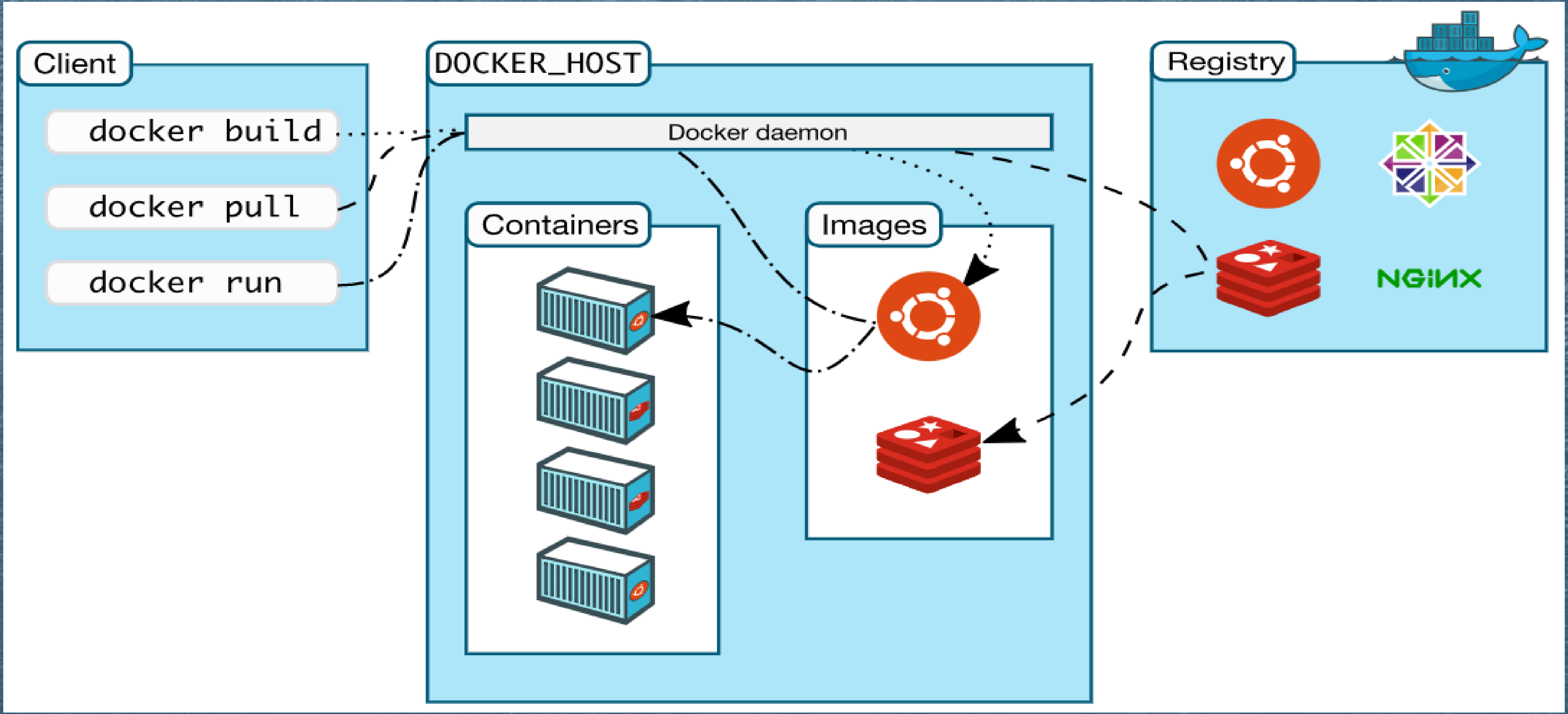
Simplifying Configuration
Developer Productivity
Server Consolidation
Multi-tenancy
Code Pipeline Management
App Isolation
Debugging Capabilities
Rapid Deployment



Docker components



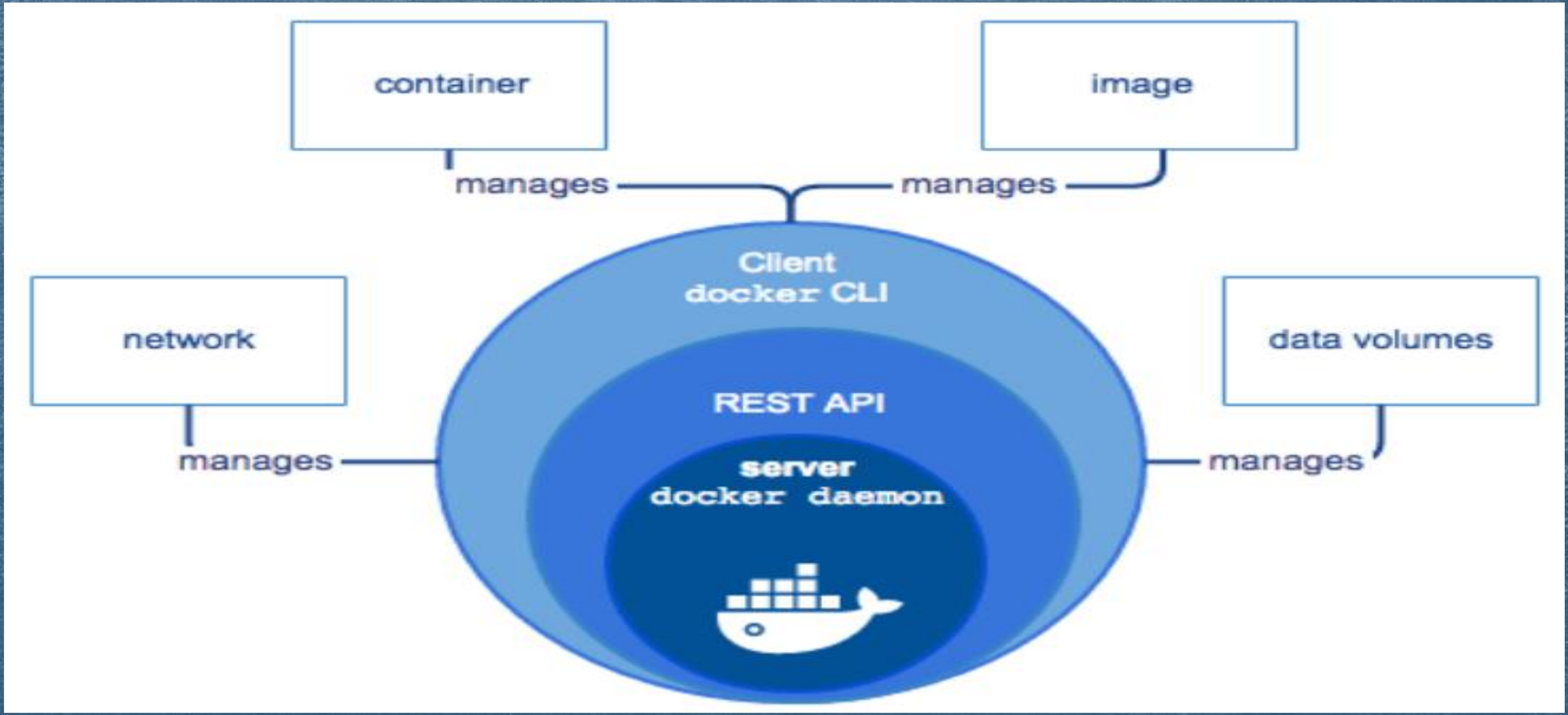
Docker components



Docker components : explain

- The Docker daemon (dockerd) listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes. A daemon can also communicate with other daemons to manage Docker services.
- The Docker client (docker) is the primary way that many Docker users interact with Docker. When you use commands such as docker run, the client sends these commands to dockerd, which carries them out. The docker command uses the Docker API. The Docker client can communicate with more than one daemon.
- A Docker *registry* stores Docker images. Docker Hub and Docker Cloud are public registries that anyone can use, and Docker is configured to look for images on Docker Hub by default.
- When you use the docker pull or docker run commands, the required images are pulled from your configured registry. When you use the docker push command, your image is pushed to your configured registry.

Docker components



Docker Engine : explain

- *It* is a client-server application with these major components:
  - Serve, a long-running program called a daemon process (the dockerd command).
  - REST API, specifies interfaces that programs can use to talk to the daemon and instruct it what to do.
  - A command line interface (CLI) client (the docker command).

- The CLI uses the Docker REST API to control or interact with the Docker daemon through scripting or direct CLI commands.
- The daemon creates and manages Docker *objects*, such as images, containers, networks, and volumes.

Docker objects : Images

- An *image* is a read-only template with instructions for creating a Docker container.
- An image is *based on* another image, with some additional customization.
- To build your own image, you create a *Dockerfile* with a simple syntax for defining the steps needed to create the image and run it.
- When you change the Dockerfile and rebuild the image, only those layers which have changed are rebuilt. This is part of what makes images so lightweight, small, and fast, when compared to other virtualization technologies.

Docker objects : Containers

- A container is a runnable instance of an image.
- You can connect a container to one or more networks, attach storage to it, or even create a new image based on its current state.
- By default, a container is relatively well isolated from other containers and its host machine.
- A container is defined by its image as well as any configuration options you provide to it when you create or start it. When a container is removed, any changes to its state that are not stored in persistent storage disappear.

Docker objects : Services

- Services allow you to scale containers across multiple Docker daemons, which all work together as a *swarm* with multiple *managers* and *workers.*
- Each member of a swarm is a Docker daemon, and the daemons all communicate using the Docker API.
- By default, the service is load-balanced across all worker nodes.
- Docker Engine supports swarm mode in Docker 1.12 and higher.

Namespaces

- It provides the isolated workspace called container. When you run a container, Docker creates a set of namespaces for that container.
- It provides a layer of isolation and each aspect of a container runs in separate ns and is limited to that ns.
- DE uses namespaces such as the following on Linux:
  - The pid namespace: Process isolation (PID: Process ID).
  - The net namespace: Managing network interfaces (NET: Networking).
  - The ipc namespace: Managing access to IPC resources (IPC: InterProcess Communication).
  - The mnt namespace: Managing filesystem mount points (MNT: Mount).
  - The uts namespace: Isolating kernel and version identifiers. (UTS: Unix Timesharing System).

Control Groups

- Docker Engine on Linux also relies on another technology called control groups (cgroups).
- A cgroup limits an application to a specific set of resources. Control groups allow DE to share available hardware resources to containers and optionally enforce limits and constraints.
- For example, you can limit the memory available to a specific container.

Union File Systems

- Union file systems, or UnionFS, are file systems that operate by creating layers, making them very lightweight and fast.
- Docker Engine uses UnionFS to provide the building blocks for containers.
- Docker Engine can use multiple UnionFS variants, including
  - AUFS(Another Union File System) : It allows files and directories of separate filesystem to co-exist under a single roof.
  - Btrfs(b Tree FS) : address the lack of pooling, snapshots, checksums, and integral multi-device spanning in Linux file systems.
  - Vfs(Virtual File system) : allow client applications to access different types of concrete file systems in a uniform way.
  - DeviceMapper : framework provided by the Linux kernel for mapping physical block devices onto higher-level virtual block devices.

Container Format

- Docker Engine combines the namespaces, control groups, and UnionFS into a wrapper called a container format.
- The default container format is libcontainer.

Swarm

- It consists of multiple Docker hosts which run in swarm mode and act as managers (to manage membership and delegation) and workers (which run swarm services).
- Docker host can be a manager, a worker, or perform both roles.
- A *task* is a running container which is part of a swarm service and managed by a swarm manager, as opposed to a standalone container.
- A key difference between standalone containers and swarm services is that only swarm managers can manage a swarm, while standalone containers can be started on any daemon.

Swarm : nodes

- Instance of the Docker engine participating in the swarm.
- You can run one or more nodes on a single physical computer or cloud server.
- Production swarm deployments typically include Docker nodes distributed across multiple physical and cloud machines.
- To deploy your application to a swarm, you submit a service definition to a **manager node**.
- The manager node dispatches units of work called tasks to worker nodes.
- Manager nodes also perform the orchestration and cluster management functions.
- **Worker nodes** receive and execute tasks dispatched from manager nodes.
- You can configure them to run manager tasks exclusively and be manager-only nodes.
- Worker node notifies the manager node of the current state of its assigned tasks so that the manager can maintain the desired state of each worker.

Docker Basic Commands & File

- When an operator executes docker run, the container process that runs is isolated in that it has its own file system, its own networking, and its own isolated process tree separate from the host.
  - Operator exclusive options : Only the operator can set the options.
  - Container identification : Operator can identify a container in three ways.

- Few commands listed are : docker run, docker attach, docker build, docker checkpoint, docker commit, docker config, docker container, docker cp, docker create, docker deploy etc..

Docker : volumes

- Volumes are the preferred mechanism for persisting data generated by and used by Docker containers. While bind mounts are dependent on the directory structure of the host machine, volumes are completely managed by Docker. Volumes have several advantages over bind mounts:
- Volumes are easier to back up or migrate than bind mounts.
- You can manage volumes using Docker CLI commands or the Docker API.
- Volumes work on both Linux and Windows containers.
- Volumes can be more safely shared among multiple containers.
- Volume drivers let you store volumes on remote hosts or cloud providers, to encrypt the contents of volumes, or to add other functionality.
- New volumes can have their content pre-populated by a container.

Docker : mount

- Bind Mounts : Bind mounts have been around since the early days of Docker. Bind mounts have limited functionality compared to volumes. 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.
- TMFS Mounts : If you're running Docker on Linux, you have a third option: tmpfs mounts. When you create a container with a tmpfs mount, the container can create files outside the container's writable layer. As opposed to volumes and bind mounts, a tmpfs mount is temporary, and only persisted in the host memory. When the container stops, the tmpfs mount is removed, and files written there won't be persisted.

Docker : layers

- Layers of a Docker image are essentially just files generated from running some command. You can view the contents of each layer on the Docker host at /var/lib/docker/aufs/diff. Layers are neat because they can be re-used by multiple images saving disk space and reducing time to build images while maintaining their integrity.
- Each container is an image with a readable/writeable layer on top of a bunch of read-only layers. These layers (also called intermediate images) are generated when the commands in the Dockerfile are executed during the Docker image build.
- Layer created is listed represented by its random generated ID
- Layers are found : /var/lib/docker/aufs

Docker : Compose

- Compose is a tool for defining and running multi-container Docker applications.
  - With Compose, YAML file is used to configure your application’s services.
  - Compose works in all environments: prod, stage, dev, test & CI workflows.

- Using Compose is basically a three-step process:
  - Define your app’s environment with a Dockerfile to reproduce anywhere.
  - Define the services that make up your app in docker-compose.yml so they can be run together in an isolated environment.
  - Run docker-compose up and Compose starts and runs your entire app.

Docker : Compose

- Compose has commands for managing the whole lifecycle of your application:
  - Start, stop, and rebuild services
  - View the status of running services
  - Stream the log output of running services
  - Run a one-off command on a service

- The features of Compose that make it effective are:
  - Multiple isolated environments on a single host
  - Preserve volume data when containers are created
  - Only recreate containers that have changed
  - Variables and moving a composition between environments

Docker : Compose

- Compose uses a project name to isolate environments from each other:
  - on a dev host, to create multiple copies of a single environment, such as when you want to run a stable copy for each feature branch of a project
  - on a CI server, to keep builds from interfering with each other, you can set the project name to a unique build number
  - on a shared host or dev host, to prevent different projects, which may use the same service names, from interfering with each other

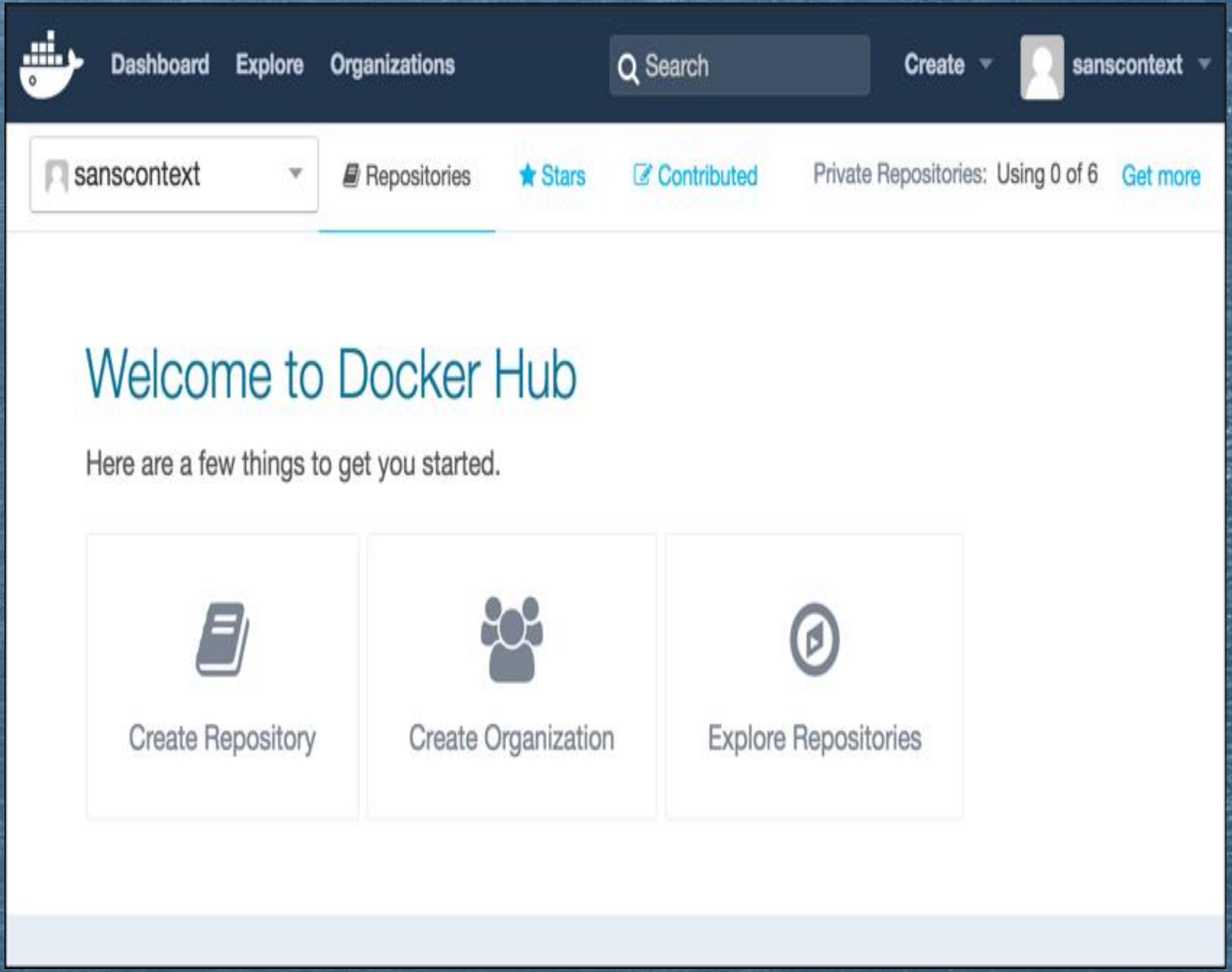
- Preserve volume data when containers are created:
  - Compose preserves all volumes used by your services. When docker-compose up runs, if it finds any containers from previous runs, it copies the volumes from the old container to the new container. This process ensures that any data you've created in volumes isn't lost.

Docker : Compose

- Common use cases:
  - Development environments : When you're developing software, the ability to run an application in an isolated environment and interact with it is crucial. The Compose command line tool can be used to create the environment and interact with it.
  - Automated testing environments : An important part of any Continuous Deployment or Continuous Integration process is the automated test suite. Automated end-to-end testing requires an environment in which to run tests.
  - Single host deployments : Compose has traditionally been focused on development and testing workflows, but with each release we're making progress on more production-oriented features. You can use Compose to deploy to a remote Docker Engine.

Docker Hub and Docker Cloud for scalable infra

Docker Hub is a cloud-based registry service which allows you to link to code repositories, build your images and test them, stores manually pushed images, and links to Docker Cloud so you can deploy images to your hosts. It provides a centralized resource for container image discovery, distribution and change management, user and team collaboration, and workflow automation throughout the development pipeline.



Welcome to Docker Hub

Here are a few things to get you started.

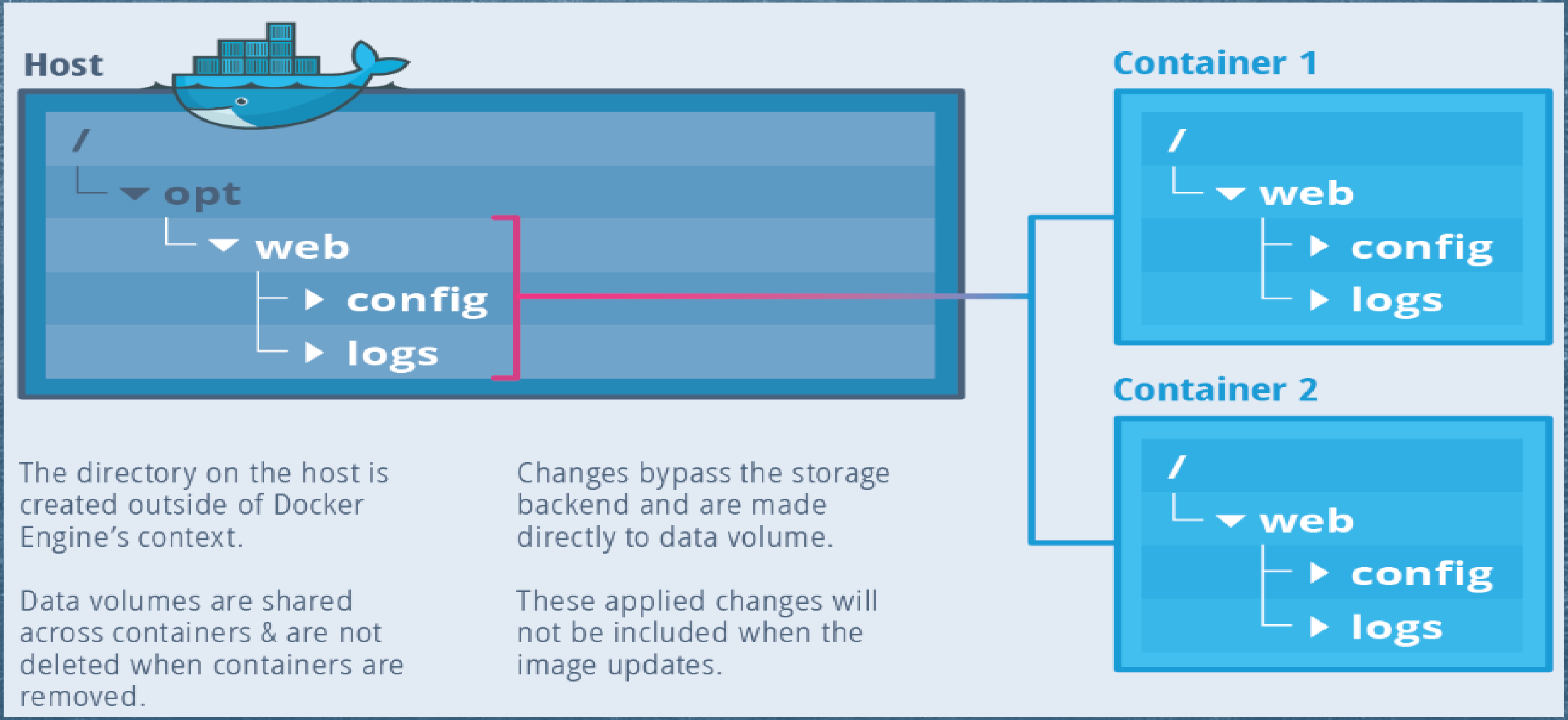
Docker Hub and Docker Cloud for scalable infra

- Docker Hub provides the following major features:
  - Image Repositories: Find and pull images from community and official libraries, and manage, push to, and pull from private image libraries to which you have access.
  - Automated Builds: Automatically create new images when you make changes to a source code repository.
  - Webhooks: A feature of Automated Builds, Webhooks let you trigger actions after a successful push to a repository.
  - Organizations: Create work groups to manage access to image repositories.
  - GitHub and Bitbucket Integration: Add the Hub and your Docker Images to your current workflows.

Docker : Volumes

- Volumes are the preferred mechanism for persisting data generated by and used by Docker containers.
- volumes are often a better choice than persisting data in a container’s writable layer, because a volume does not increase the size of the containers using it, and the volume’s contents exist outside the lifecycle of a given container.
- If your container generates non-persistent state data, consider using a tmpfs mount to avoid storing the data anywhere permanently, and to increase the container’s performance by avoiding writing into the container’s writable layer.

Persistence of data in containers



The directory on the host is created outside of Docker Engine's context.

Data volumes are shared across containers & are not deleted when containers are removed.

Changes bypass the storage backend and are made directly to data volume.

These applied changes will not be included when the image updates.

Persistence of data in containers

- Application container management
  - Stackable image layers
    - Docker images are made from stackable layers which means developers effectively deploy changes. The stackable image layers and a writable top 'container layer' are what constitute a container. The act of removing a container is to delete the top read-write layer leaving the underlying image layers untouched.
  - Copy-on-write 'container layer'
    - continuously delivered containers. As applications are versioned, new layers are applied or removed from the previous version. Only changes need to be applied and deployed against the previous version. When you launch a container, a copy-on-write R/W 'container layer' is applied to the top of the R/O image stack making it read-writable.

Solution?

Solution to persistent data? vm level!

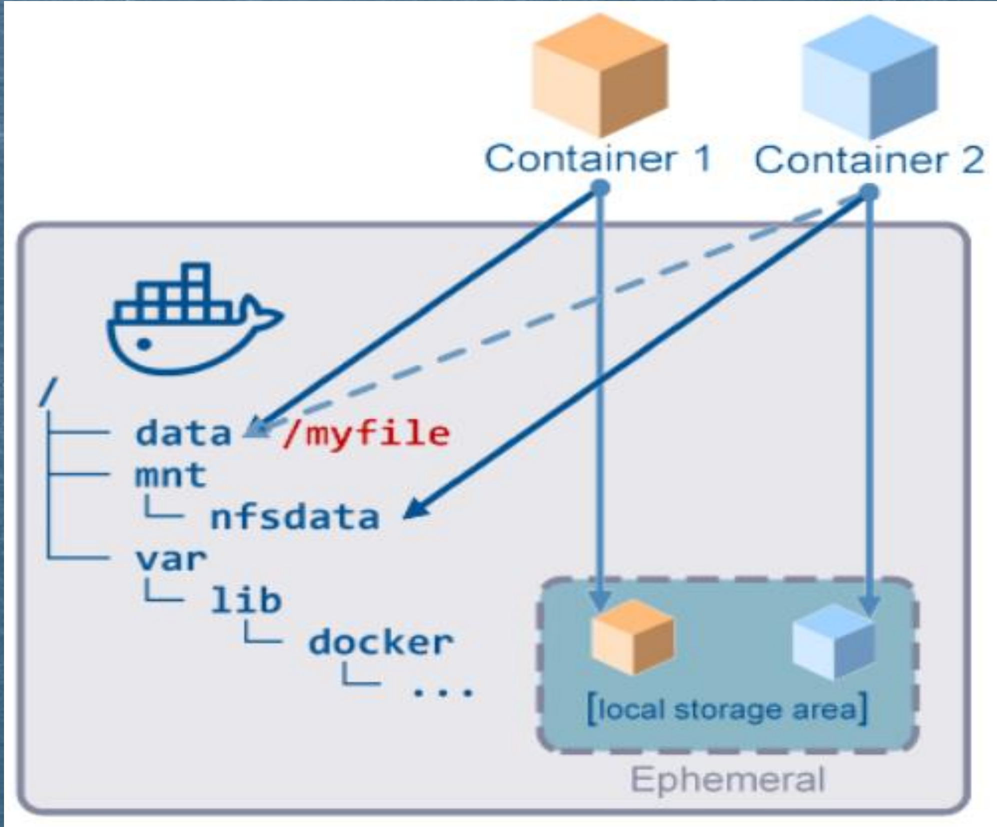
- Local directory mounts
  - Local directory mounts can be a directory or file or an NFS mount in the Docker host’s filesystem >> mounted into vm’s container

To illustrate a local directory mount, we can write some data to ‘~/tmp’ and terminate.

```
docker run --rm -v ~/tmp:/data alpine ash -c \
"echo hello world > /data/myfile“
```

We created a container and wrote hello world to Myfile which is in host tmp folder and data persists Even after the container is removed.

```
sudo cat ~/tmp/myfile hello world
```



Solution to persistent data? vm level!

- Local named volumes
  - Docker local named volumes place data into Docker’s data storage region ‘var/lib/docker/volumes’. Local named volumes can be shared between containers

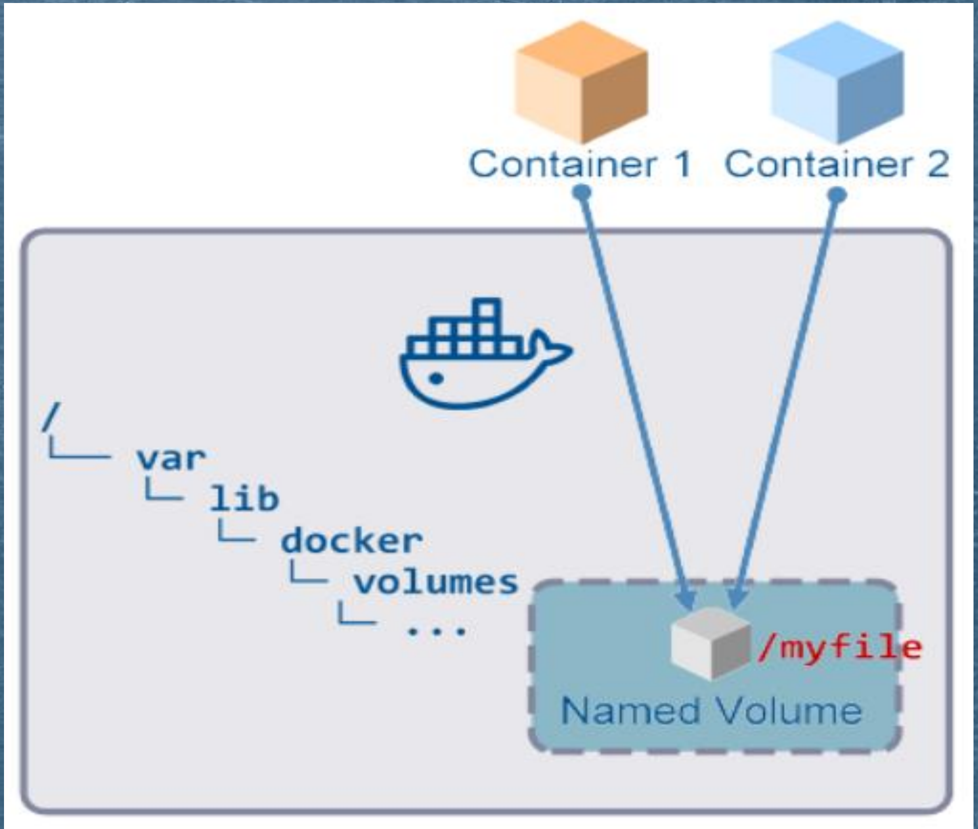
we will create a persistent volume under ‘var/lib/docker/volumes

docker volume create --name mydata

we can write some data to the volume and terminate our container

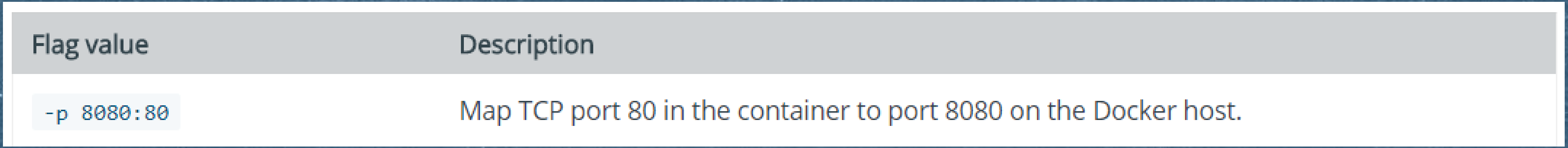
docker run --rm -v mydata:/data:rw alpine ash -c \ "echo hello world > /data/myfile“

sudo cat /var/lib/docker/volumes/mydata/_data/ myfile hello world



Network communication between containers

- The type of network a container uses, whether it is a bridge, an overlay, a macvlan network, or a custom network plugin, is transparent from within the container.
- By default, when you create a container, it does not publish any of its ports to the outside world. To make a port available to services outside of Docker, or to Docker containers which are not connected to the container's network, use the --publish or -p flag. This creates a firewall rule which maps a container port to a port on the Docker host.



Flag value	Description
`-p 8080:80`	Map TCP port 80 in the container to port 8080 on the Docker host.

Network communication between containers

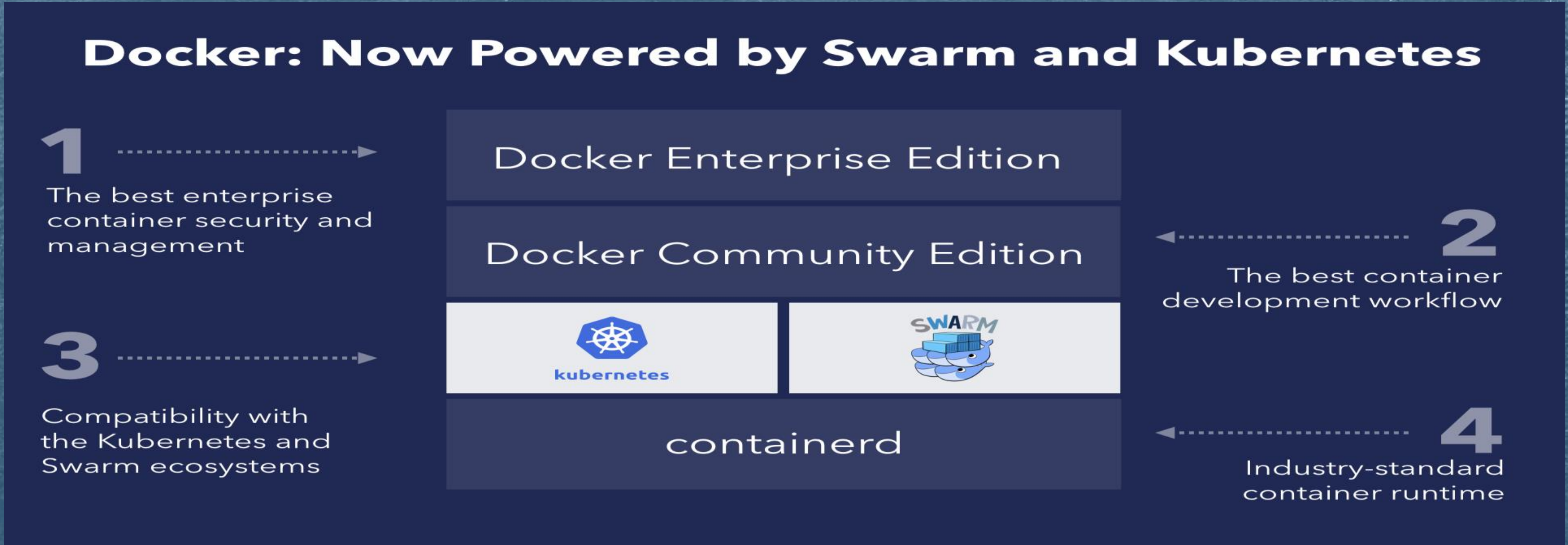
IP address and hostname

- By default, the container is assigned an IP address for every Docker network it connects to. The IP address is assigned from the pool assigned to the network, so the Docker daemon effectively acts as a DHCP server for each container. Each network also has a default subnet mask and gateway.
- you can connect a running container to multiple networks using docker network connect
- By default, a container inherits the DNS settings of the Docker daemon, including the /etc/hosts and /etc/resolv.conf.You can override these settings on a per-container basis.

kubernetes

Container orchestration

- The Docker platform now integrates with Kubernetes. This means that developers and operators can build apps with Docker and seamlessly test and deploy them using both Docker Swarm and Kubernetes.



Docker: Now Powered by Swarm and Kubernetes
1
The best enterprise container security and management
2
The best container development workflow
3
Compatibility with the Kubernetes and Swarm ecosystems
4
Industry-standard container runtime
Docker Enterprise Edition
Docker Community Edition
kubernetes
SWARM
containerd

Container orchestration

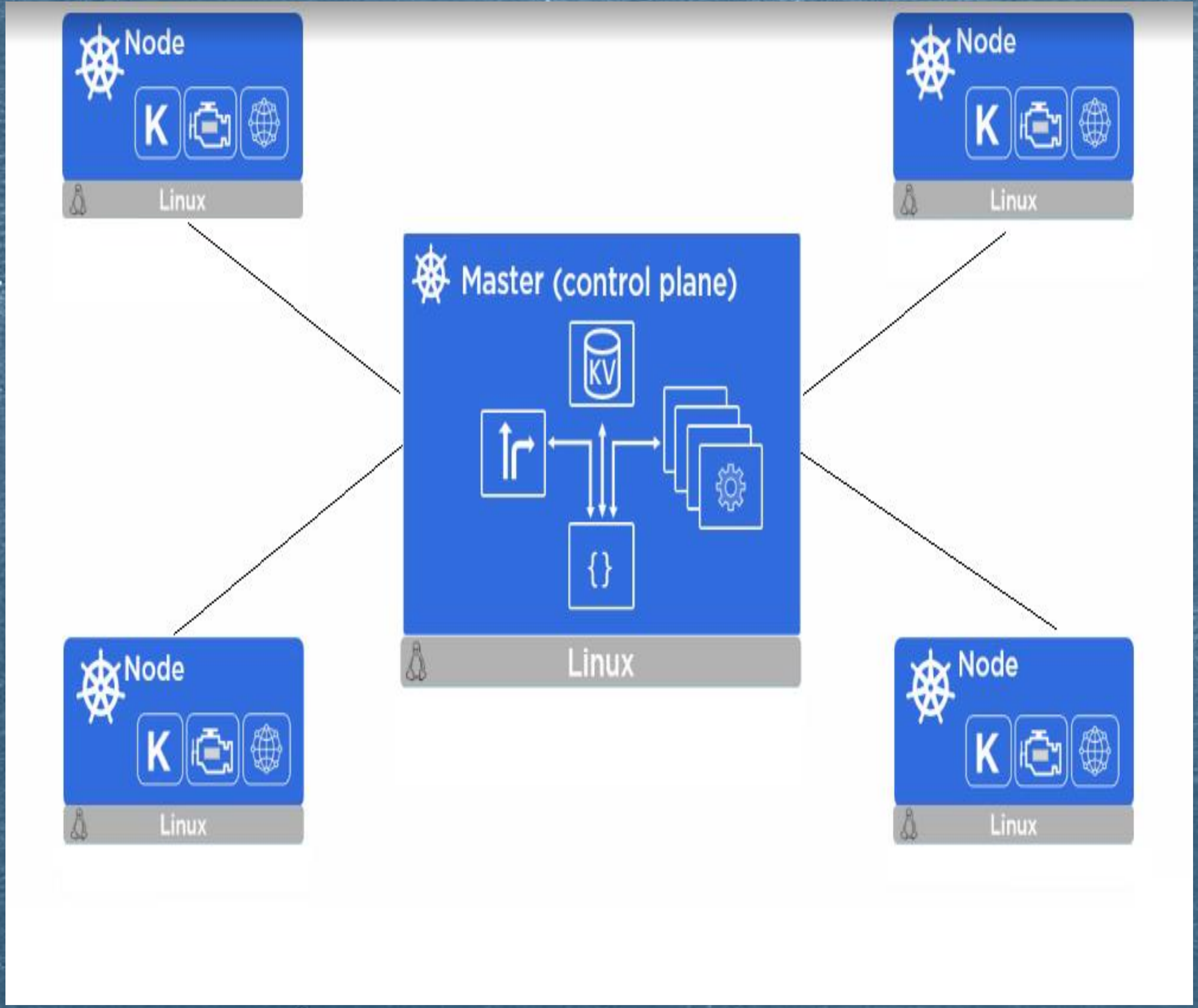
- Scenario : You need to start the right containers at the right time, figure out how they can talk to each other, handle storage considerations, and deal with failed containers or hardware.
- Kubernetes is an open source container orchestration platform, allowing large numbers of containers to work together in harmony, reducing operational burden. It helps with things like:
  - Running containers across many different machines
  - Scaling up or down by adding or removing containers when demand changes
  - Keeping storage consistent with multiple instances of an application
  - Distributing load between the containers
  - Launching new containers on different machines if something fails

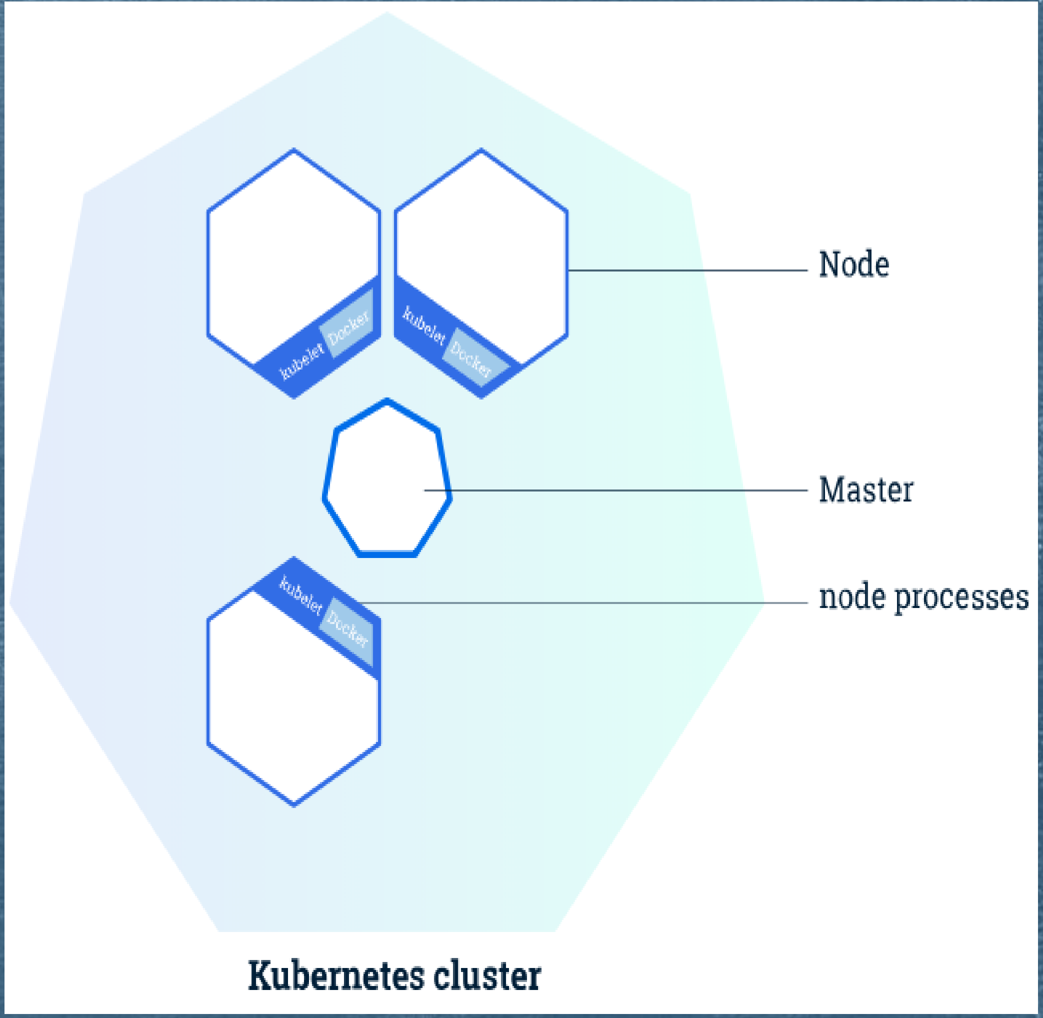
Kubernetes and other orchestration systems

- 2017 was the year Kubernetes conquered the container orchestration space.
- Docker Swarm and Mesos have been offering their own container orchestration tools.
- Now both have Kubernetes along with aws, azure, oracle in their ecosystem.

Docker Swarm / Pros	Kubernetes / Pros
Easy Installation & Setup, Integration Options with Docker, Less Expensive	Most Popular Solution with Strong community, Fully Open Source Platform High-Fault Tolerance, Built-in Logging & Monitoring Tools, Auto-Scaling Functionality
Docker Swarm / Cons	**Kubernetes / Cons**
Limited API, Smaller Community, Low-Fault Tolerance, Less Customizable, 3rd-Party Monitoring and Logging Tools, Scale Services Manually	Harder Installation & Setup, Less User-Friendly

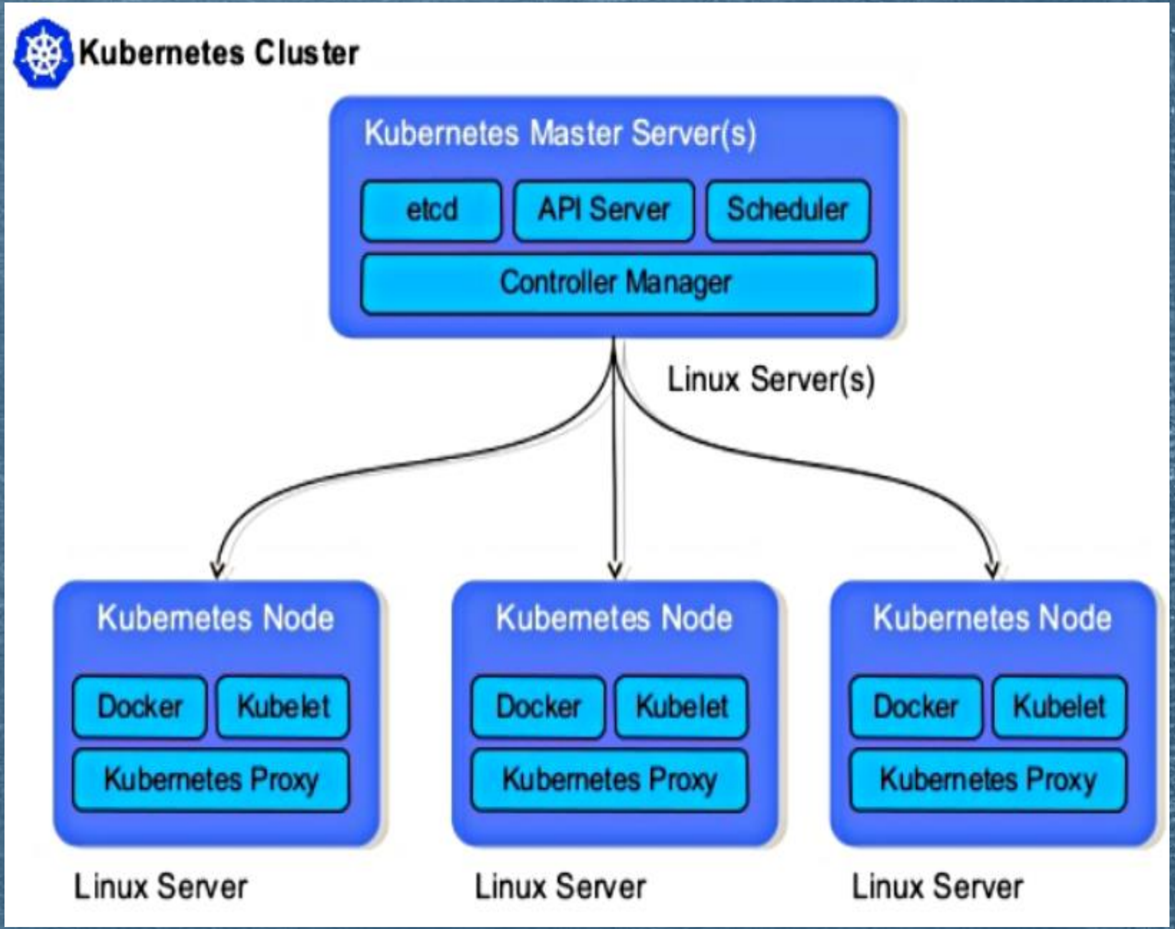
Kubernetes : components





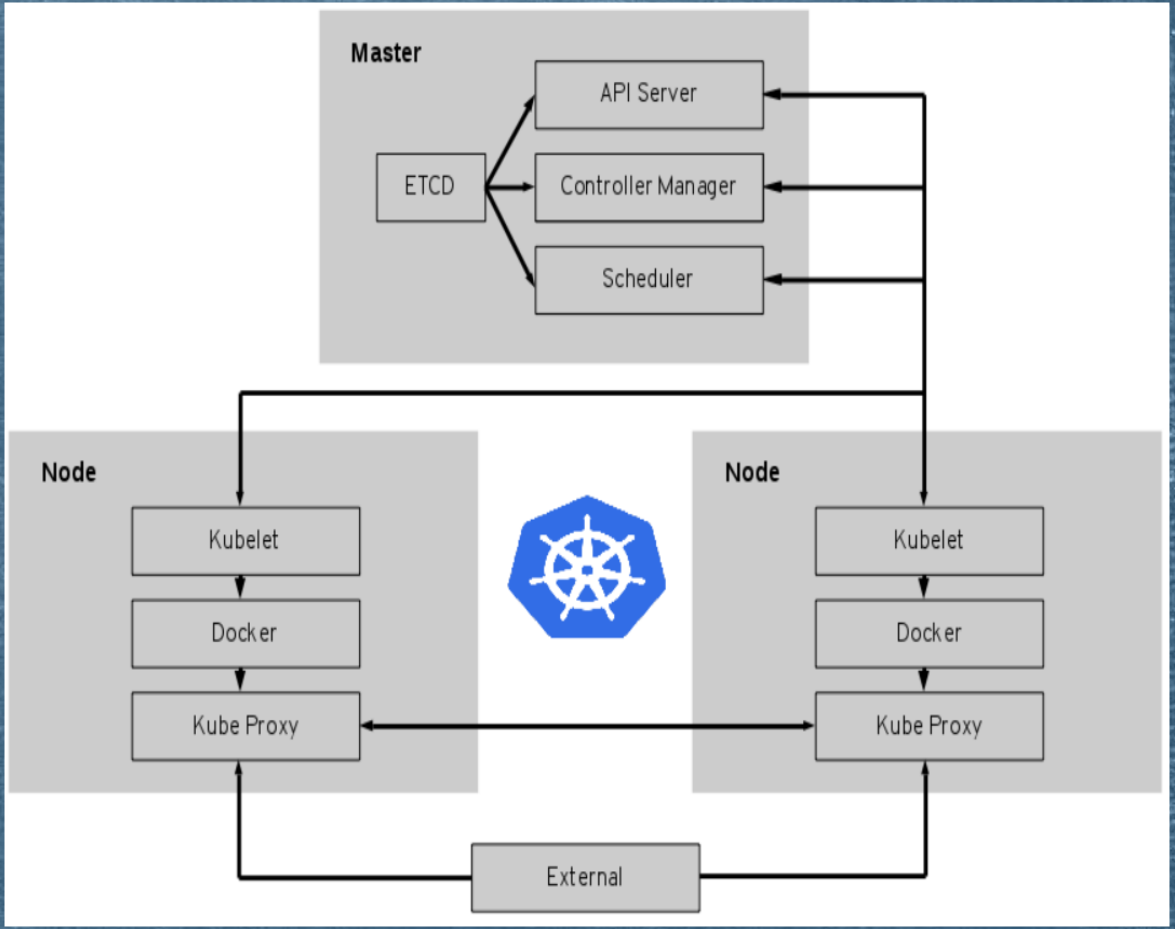
Kubernetes M.components : apiserver

- Kube-apiserver : It is the brain to the master and is front-end to the master or control plane. Kube-apiserver implements the RESTful API and consumes json via a manifest file. Manifest files declare the state of the app like a record of intent and are validated and deployed on the cluster. It exposes an endpoint (by default on port 443) so that kubectl (command line utility) can issue commands/queries and run on the master.



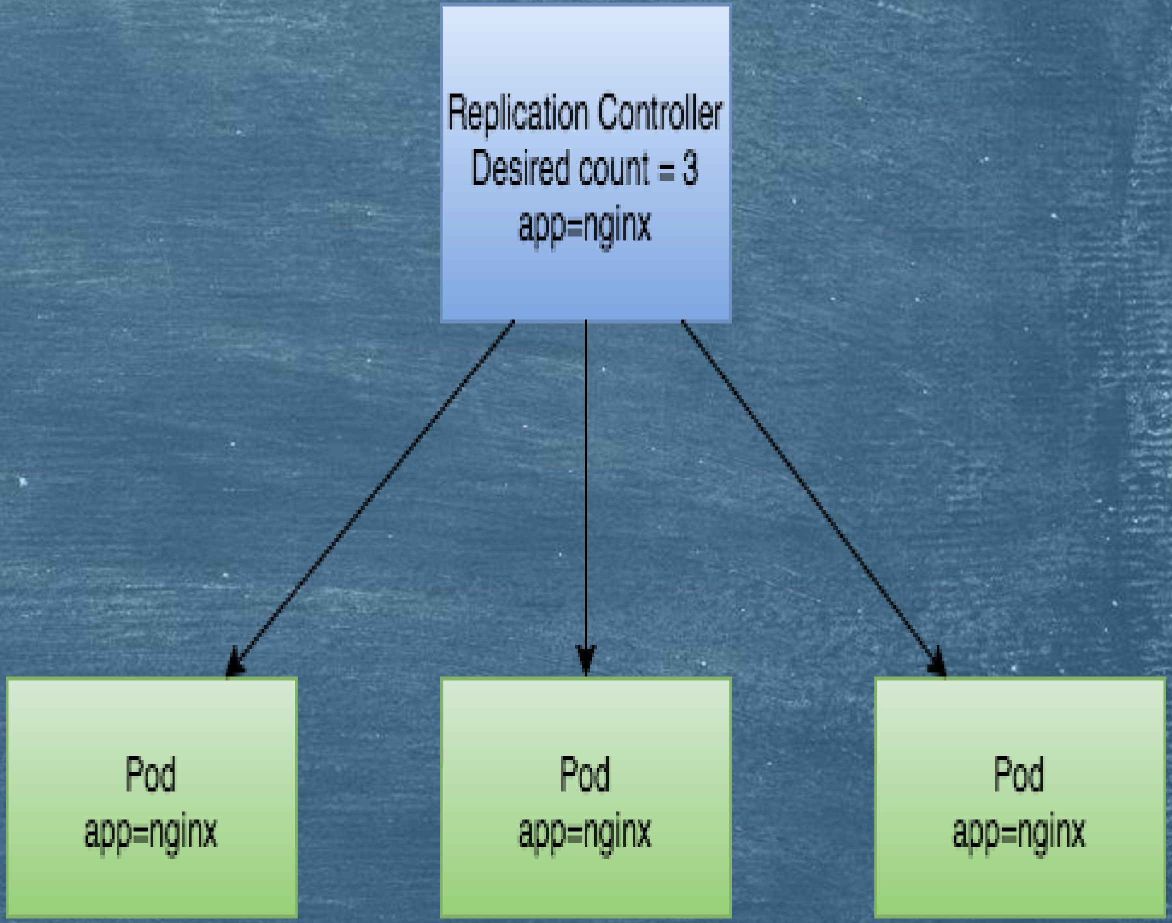
Kubernetes M.components : Cluster Store, etcd

- etcd : provides persistent sto rage and is stateful. Open source distributed key-value store that serves as the backbone of distributed systems by providing a canonical hub for cluster coordination and state management. It takes care of storing and replicating data used by Kubernetes across the entire cluster.



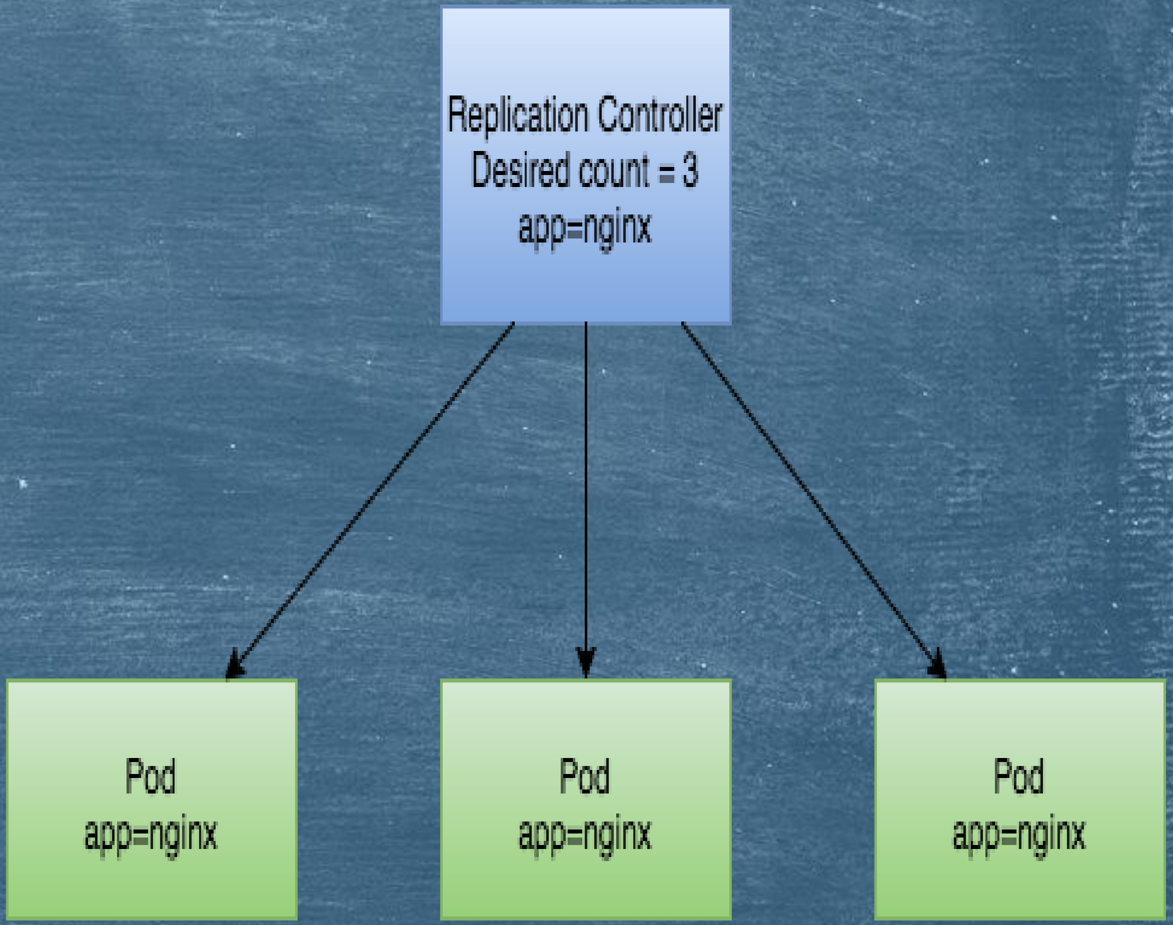
Kubernetes M.components : controller manager

- Control manager : is a daemon that implants the core control loops shipped with Kubernetes. It is the controller of controllers. It watches the shared state of the cluster through the API server and makes changes attempting to move the current state towards the desired state.



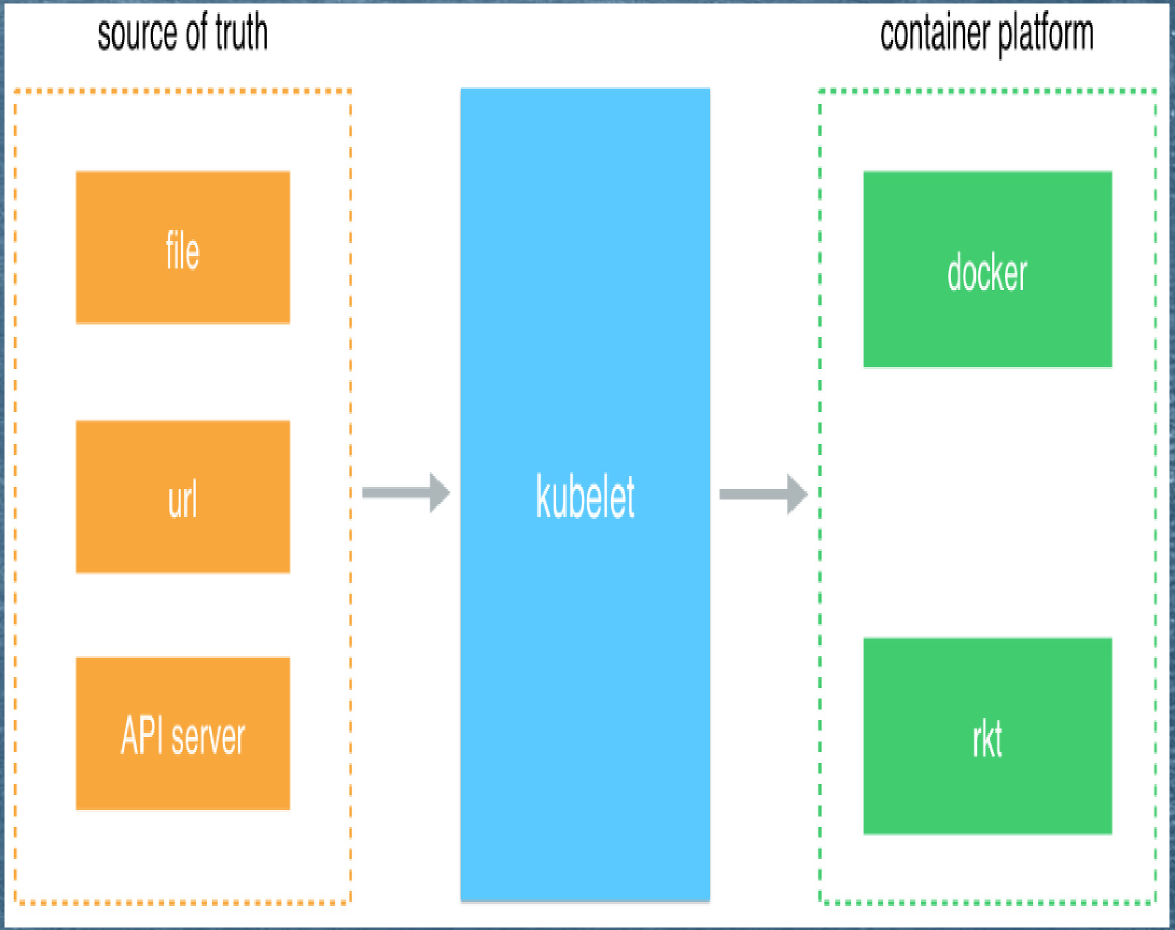
Kubernetes M.components : scheduler service

- Scheduler Service : This is the process that watches API-server for new pods and assigns workloads to specific nodes in the cluster. It is responsible for tracking resource utilization on each host to make sure that workloads are not scheduled in excess of the available resources.



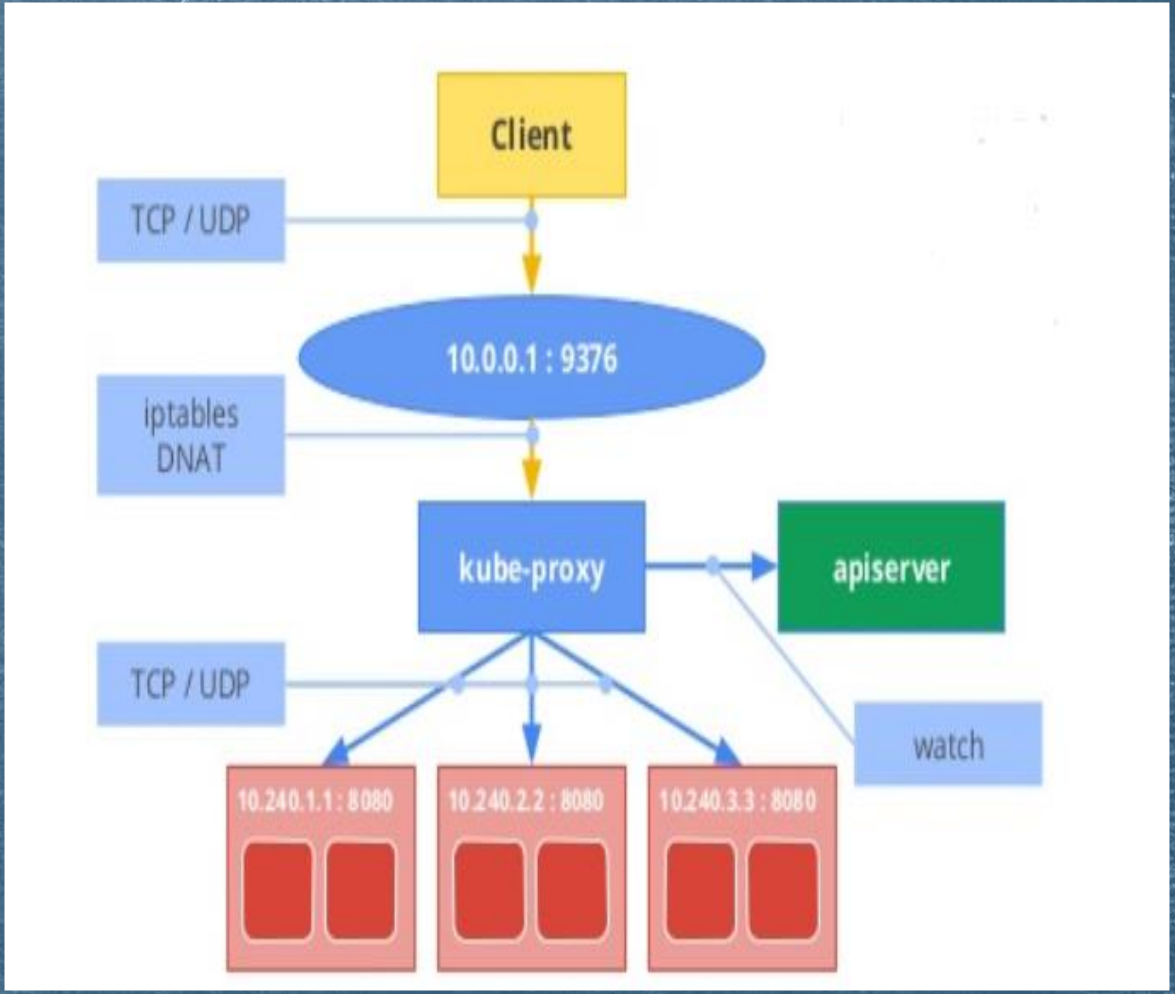
Kubernetes N.components : kubelet

- Scheduler Service : The main Kubernetes agent on the node. Registers node with the cluster. Watches API server for work assignment. Instantiate pods for carrying out the work reports back to master. Exposes endpoint on port-10255. It lets you inspect the specs of a Kubelet.



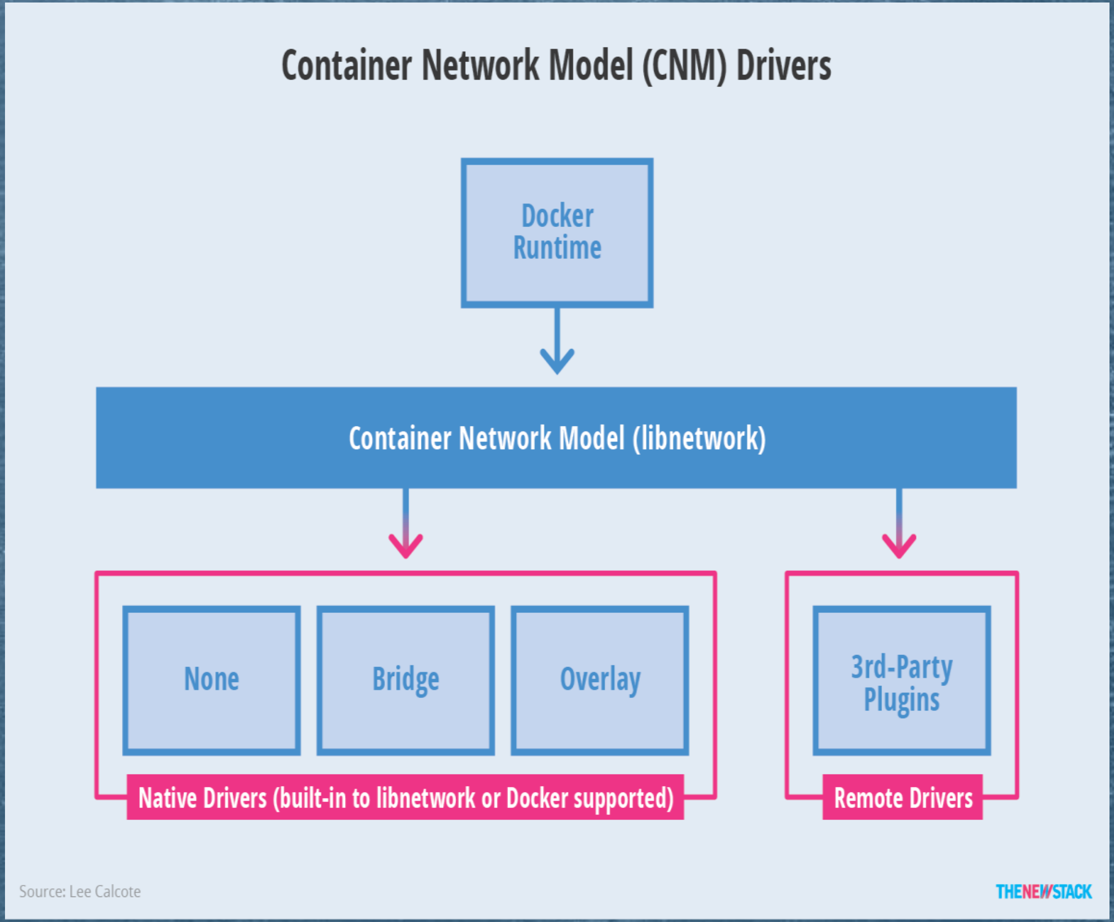
Kubernetes N.components : proxy

- Proxy : It is like the network brain of the node. It is a network proxy which reflects Kubernetes networking services on each node. It ensures every pod gets its own unique IP. If there are multiple containers in a pod, then they all will share same IP. It load balances across all pods in a service.



Kubernetes : network model

- Two ways to configuring network interfaces for Linux containers: the container network model (CNM) and the container network interface (CNI).
- The Container Network Model (CNM) is a specification proposed by Docker, adopted by projects such as libnetwork, with integrations from projects and companies such as Cisco Contiv, Kuryr, Open Virtual Networking (OVN), Project Calico, VMware and Weave.

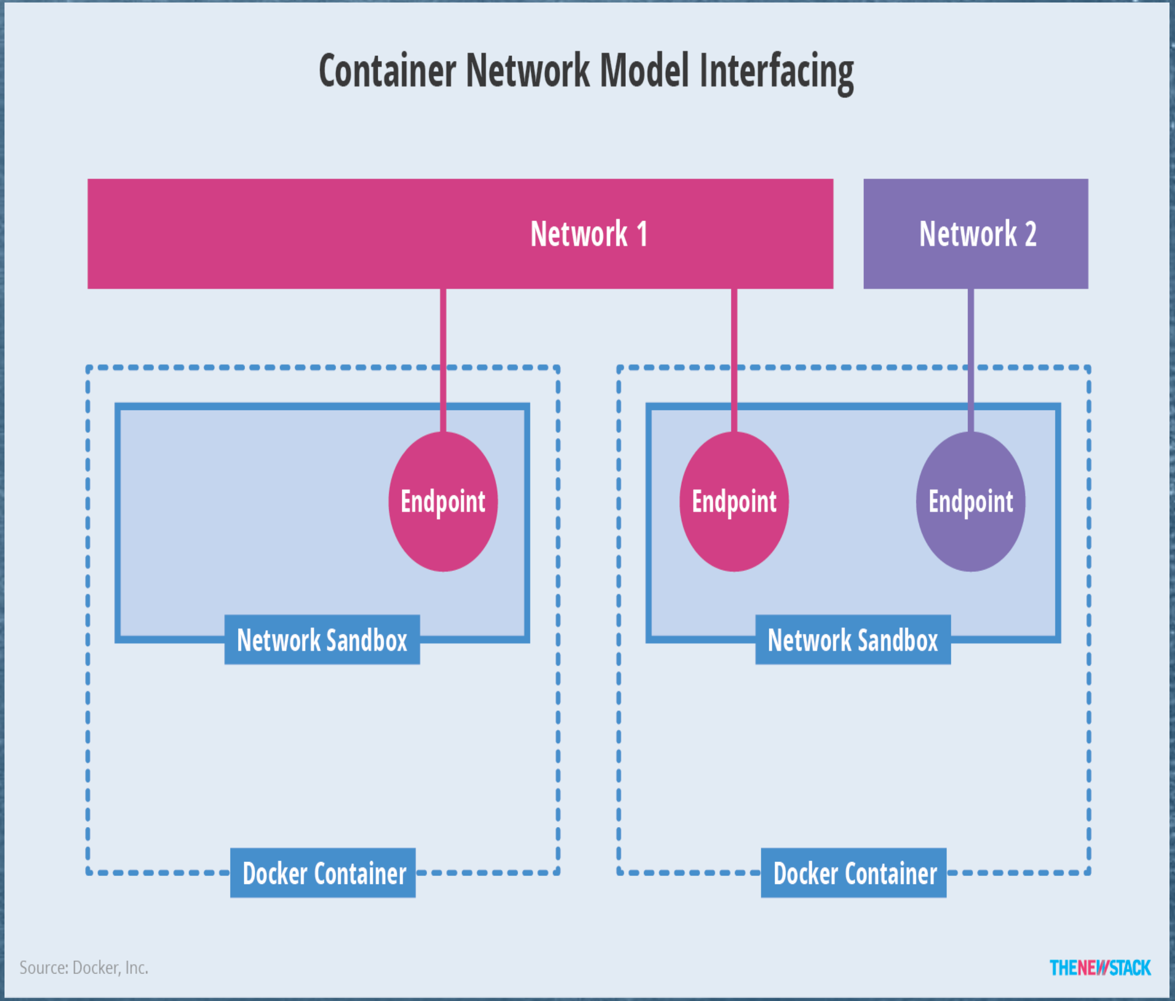

Container Network Model (CNM) Drivers
Docker Runtime
Container Network Model (libnetwork)
None
Bridge
Overlay
3rd-Party Plugins
Native Drivers (built-in to libnetwork or Docker supported)
Remote Drivers
Source: Lee Calcote
THENEWSTACK

Kubernetes : containers network model

- Libnetwork is the canonical implementation of the CNM specification. Libnetwork provides an interface between the Docker daemon and network drivers.
- The network controller is responsible for pairing a driver to a network. Each driver is responsible for managing the network it owns, including services provided to that network like IPAM.
- With one driver per network, multiple drivers can be used concurrently with containers connected to multiple networks.
- Drivers are also defined as having a local scope (single host) or global scope (multi-host).

Kubernetes : containers network model

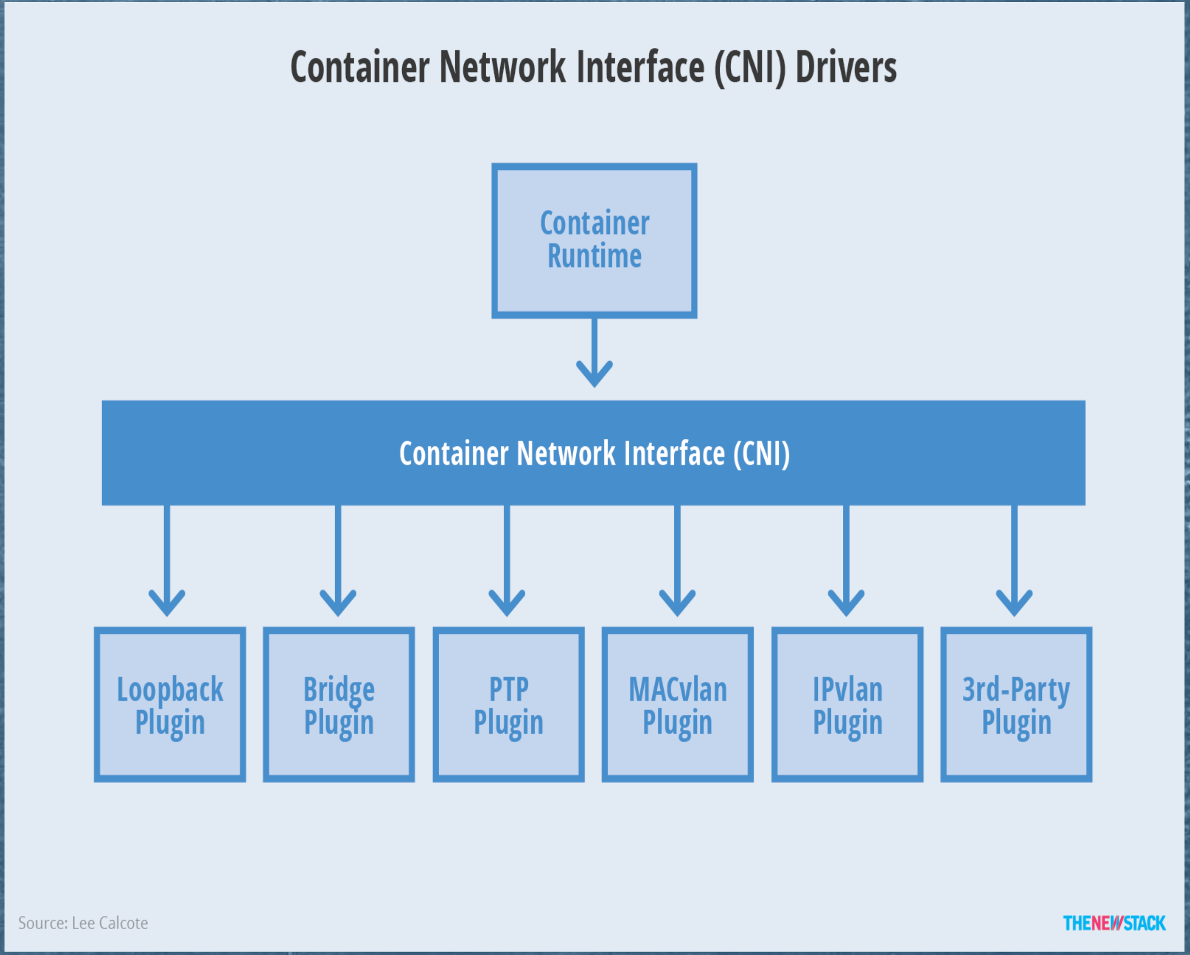
- **Network Sandbox:** Essentially the networking stack within a container.
- **Endpoint:** A network interface that typically comes in pairs. One end of the pair sits in the network sandbox, while the other sits in a designated network.
- **Network**: A group of endpoints.



Kubernetes : CNI

Container Network Interface

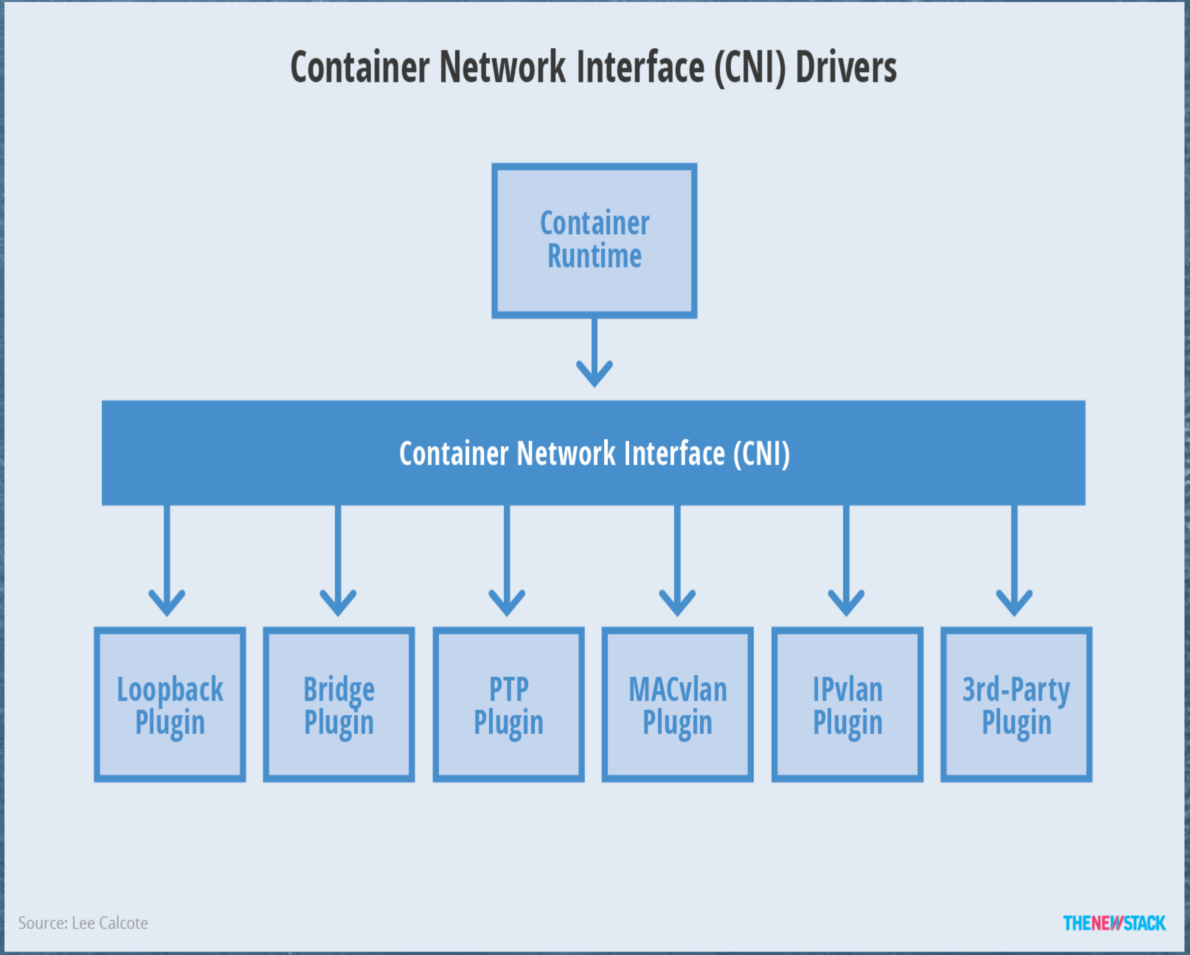
- The Container Network Interface (CNI) is a container networking specification proposed by CoreOS and adopted by projects such as Apache Mesos, Cloud Foundry, Kubernetes, Kurma and rkt.
- Multiple plugins may be run at one time with a container joining networks driven by different plugins. Networks are described in configuration files, in JSON format, and instantiated as new namespaces when CNI plugins are invoked.



Kubernetes : CNI

Container Network Interface

- The container runtime needs to first allocate a network namespace to the container and assign it a container ID, then pass along a number of parameters (CNI config) to the network driver. The network driver then attaches the container to a network and reports the assigned IP address back to the container runtime via JSON.



Kubernetes : CNM

- CNM does not provide network drivers access to the container's network namespace.
- The benefit here is that libnetwork acts as a broker for conflict resolution.
- Conflict being when two independent network drivers provide the same static route, using the same route prefix, but point to different next-hop IP addresses.
- CNM is designed to support the Docker runtime engine only.

Kubernetes : CNI

- CNI does provide drivers with access to the container network namespace.
- CNI supports integration with third-party IPAM and can be used with any container runtime.
- With CNI's simplistic approach, it's been argued that it's comparatively easier to create a CNI plugin than a CNM plugin.

Kubernetes : Service Discovery

- Service discovery is the process of figuring out how to connect to a service.
- While there is a service discovery option based on environment variables available, the DNS-based service discovery is preferable.
- To connect to a service from within the cluster from another service. You need to create a jump pod in the same namespace.
- The DNS add-on will make sure that this service is available via the FQDN thesvc.default.svc.cluster.local from other pods in the cluster.

Kubernetes : scaling and load balancing

- Internal Load Balancer
  - kube-proxy : userspace(round robin) and iptables(random)
    - When a request comes in, it assigns the request to the next destination on the list, then permutes the list, so the next request goes to the following destination on the list.
    - Iptables mode (the current default), the incoming requests are assigned to pods within a service by random selection.
- External Load Balancer
  - LoadBalancer : sets a service to use from a cloud service provider.
    - will only work with specified providers (AWS, Azure, OpenStack, CloudStack, GCP)
  - The external load balancer directs requests to the pods represented by the service.

Kubernetes : Ingress controller and reverse proxy

- API object that manages external access to the services in a cluster, typically HTTP.
- Collection of rules that allow inbound connections to reach the cluster services.
- Users request ingress by POSTing the Ingress resource to the API server.
- Kubernetes Engine deploys an ingress controller on the master.
- We can deploy any number of custom ingress controllers in a pod.

Kubernetes : Ingress controller and reverse proxy

- The kubectl proxy:
  - runs on a user's desktop or in a pod
  - proxies from a localhost address to the Kubernetes apiserver
  - client to proxy uses HTTP
  - proxy to apiserver uses HTTPS
  - locates apiserver
  - adds authentication headers

- The apiserver proxy:
  - is a bastion built into the apiserver
  - connects a user outside of the cluster to cluster IPs which otherwise might not be reachable
  - runs in the apiserver processes
  - client to proxy uses HTTPS (or http if apiserver so configured)
  - proxy to target may use HTTP or HTTPS as chosen by proxy using available information
  - can be used to reach a Node, Pod, or Service
  - load balancing when used to reach a Service

- The kube proxy:
  - runs on each node
  - proxies UDP and TCP
  - does not understand HTTP
  - provides load balancing
  - is just used to reach services

- Proxy/Load-balancer in front of apiserver(s):
  - existence and implementation varies from cluster to cluster (e.g. nginx)
  - sits between all clients and one or more apiservers
  - acts as load balancer if there are several apiserver

Kubernetes : storage backend

- PV : It is a resource in the cluster just like a node is a cluster resource.
- Have a lifecycle independent of any individual pod that uses the PV.
- PersistentVolume api object captures the details of the implementation of the storage, be that NFS, iSCSI, or a cloud-provider-specific storage system.
- PersistentVolumeClaim is a request for storage by a user. It is similar to a pod. Pods consume node resources and PVCs consume PV resources.

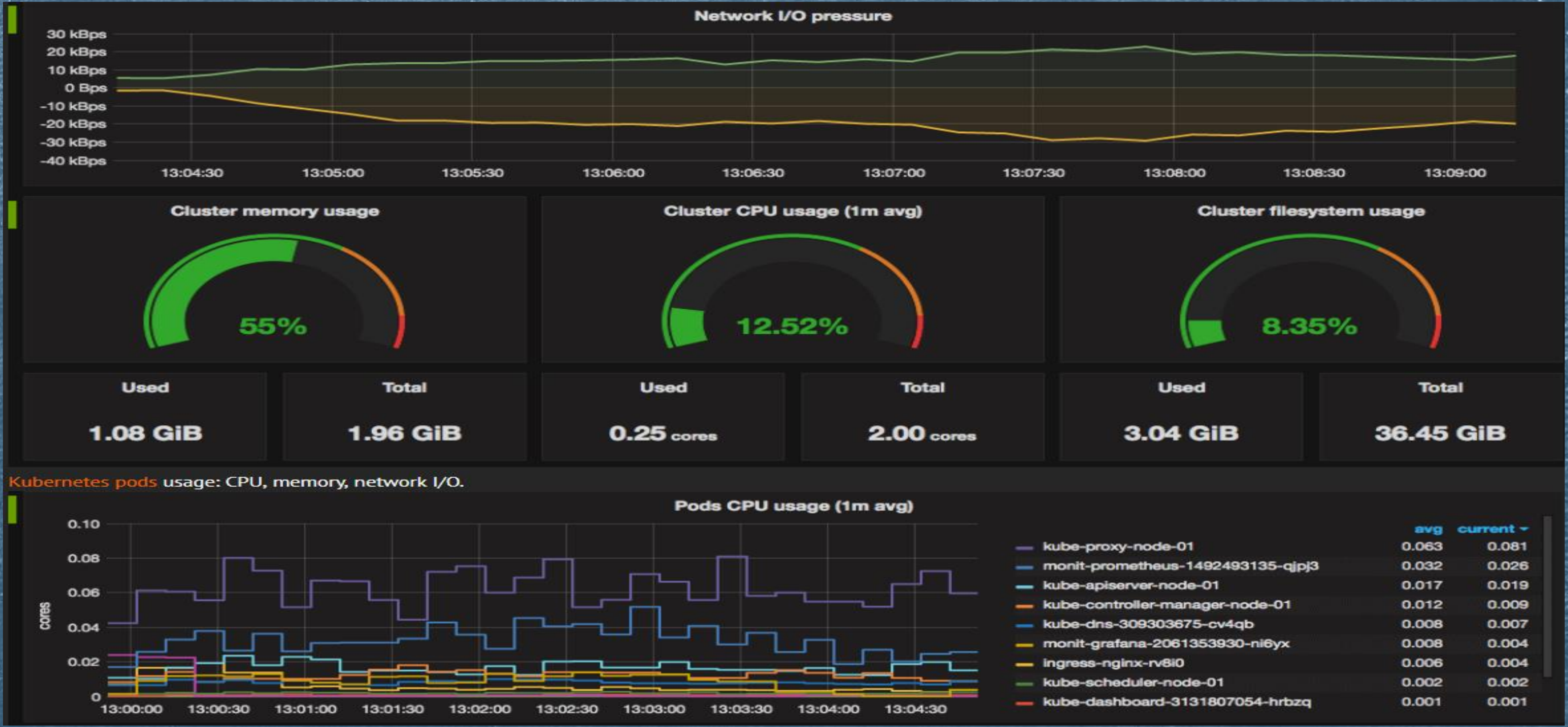
Kubernetes : Securing

- All API communication in the cluster is encrypted by default with TLS.
- It allows the necessary certificates to be created and distributed to the cluster components.
- API Authentication : clients are typically service accounts or use x509 client certificates, and they are created automatically at cluster startup
- API Authorization : integrated Role-Based Access Control (RBAC) component that matches an incoming user or group to a set of permissions bundled into roles.

Kubernetes : cluster monitoring via Prometheus

- You only need to have running Kubernetes cluster with deployed Prometheus. Prometheus will use metrics provided by cAdvisor via kubelet service (runs on each node of Kubernetes cluster by default) and via kube-apiserver service only.
- Monitors Kubernetes cluster using Prometheus. Shows overall cluster CPU / Memory / Filesystem usage as well as individual pod, containers, systemd services statistics. Uses cAdvisor metrics only.

Kubernetes : cluster monitoring via Prometheus



Network I/O pressure

Cluster memory usage

55%

Used	Total
1.08 GiB	1.96 GiB

Cluster CPU usage (1m avg)

12.52%

Used	Total
0.25 cores	2.00 cores

Cluster filesystem usage

8.35%

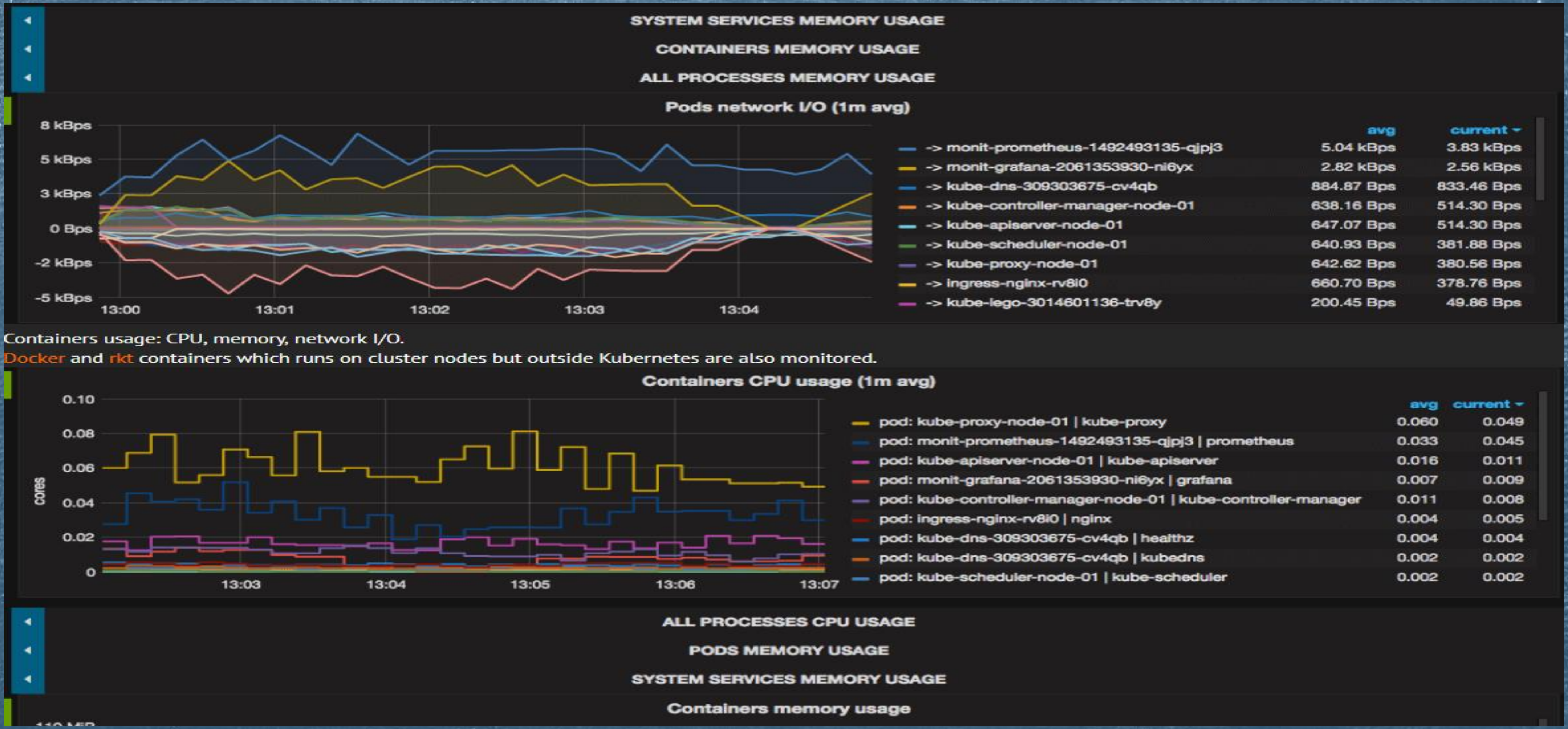
Used	Total
3.04 GiB	36.45 GiB

Kubernetes pods usage: CPU, memory, network I/O.

Pods CPU usage (1m avg)

	avg	current ▾
kube-proxy-node-01	0.063	0.081
monit-prometheus-1492493135-qjpj3	0.032	0.026
kube-apiserver-node-01	0.017	0.019
kube-controller-manager-node-01	0.012	0.009
kube-dns-309303675-cv4qb	0.008	0.007
monit-grafana-2061353930-ni6yx	0.008	0.004
ingress-nginx-rv8i0	0.006	0.004
kube-scheduler-node-01	0.002	0.002
kube-dashboard-3131807054-hrbzq	0.001	0.001

Kubernetes : cluster monitoring via Prometheus



SYSTEM SERVICES MEMORY USAGE

CONTAINERS MEMORY USAGE

ALL PROCESSES MEMORY USAGE

Pods network I/O (1m avg)

	avg	current
-> monit-prometheus-1492493135-qjpj3	5.04 kBps	3.83 kBps
-> monit-grafana-2061353930-ni6yx	2.82 kBps	2.56 kBps
-> kube-dns-309303675-cv4qb	884.87 Bps	833.46 Bps
-> kube-controller-manager-node-01	638.16 Bps	514.30 Bps
-> kube-apiserver-node-01	647.07 Bps	514.30 Bps
-> kube-scheduler-node-01	640.93 Bps	381.88 Bps
-> kube-proxy-node-01	642.62 Bps	380.56 Bps
-> ingress-nginx-rv8i0	660.70 Bps	378.76 Bps
-> kube-lego-3014601136-trv8y	200.45 Bps	49.86 Bps

Containers usage: CPU, memory, network I/O.
Docker and rkt containers which runs on cluster nodes but outside Kubernetes are also monitored.

Containers CPU usage (1m avg)

	avg	current
pod: kube-proxy-node-01 | kube-proxy	0.060	0.049
pod: monit-prometheus-1492493135-qjpj3 | prometheus	0.033	0.045
pod: kube-apiserver-node-01 | kube-apiserver	0.016	0.011
pod: monit-grafana-2061353930-ni6yx | grafana	0.007	0.009
pod: kube-controller-manager-node-01 | kube-controller-manager	0.011	0.008
pod: ingress-nginx-rv8i0 | nginx	0.004	0.005
pod: kube-dns-309303675-cv4qb | healthz	0.004	0.004
pod: kube-dns-309303675-cv4qb | kubedns	0.002	0.002
pod: kube-scheduler-node-01 | kube-scheduler	0.002	0.002

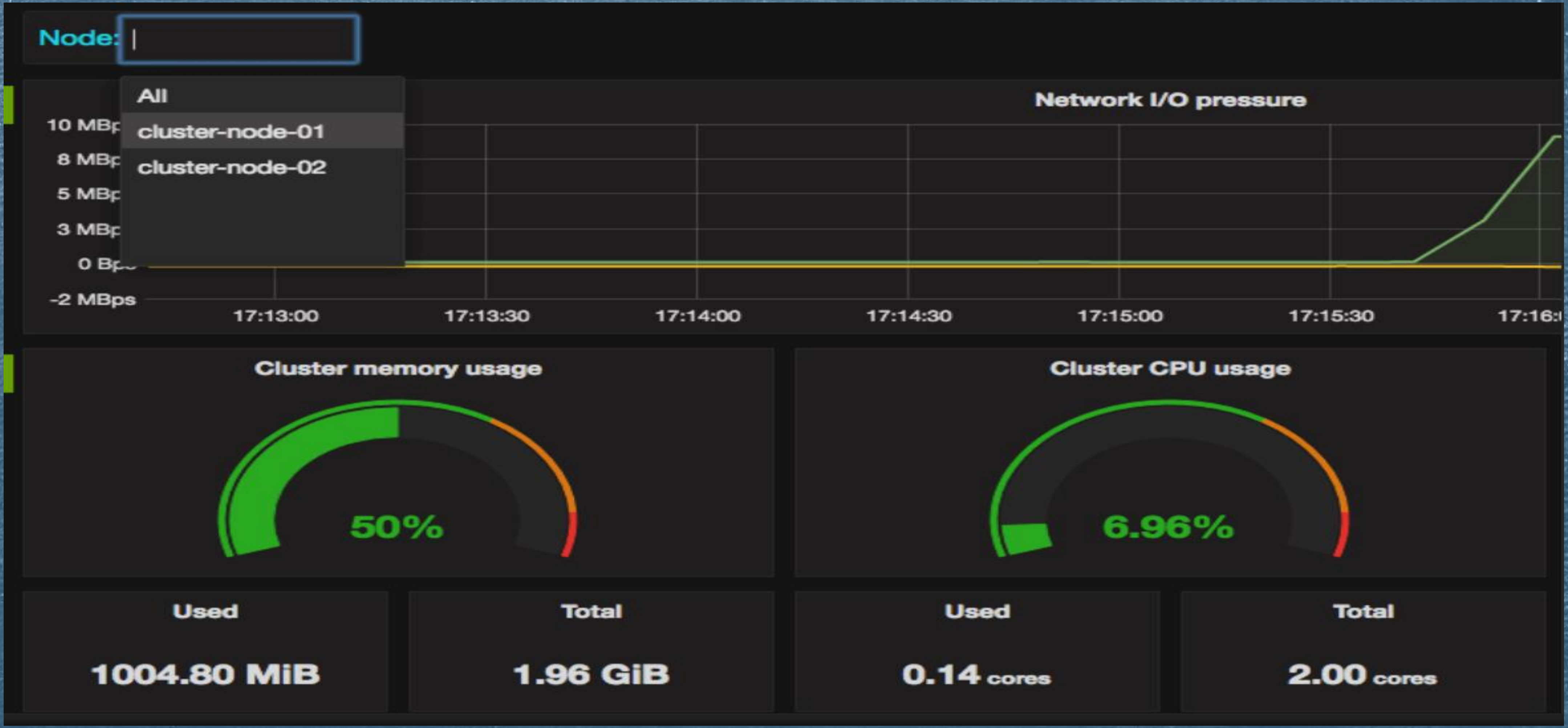
ALL PROCESSES CPU USAGE

PODS MEMORY USAGE

SYSTEM SERVICES MEMORY USAGE

Containers memory usage

Kubernetes : cluster monitoring via Prometheus



Cluster memory usage

50%

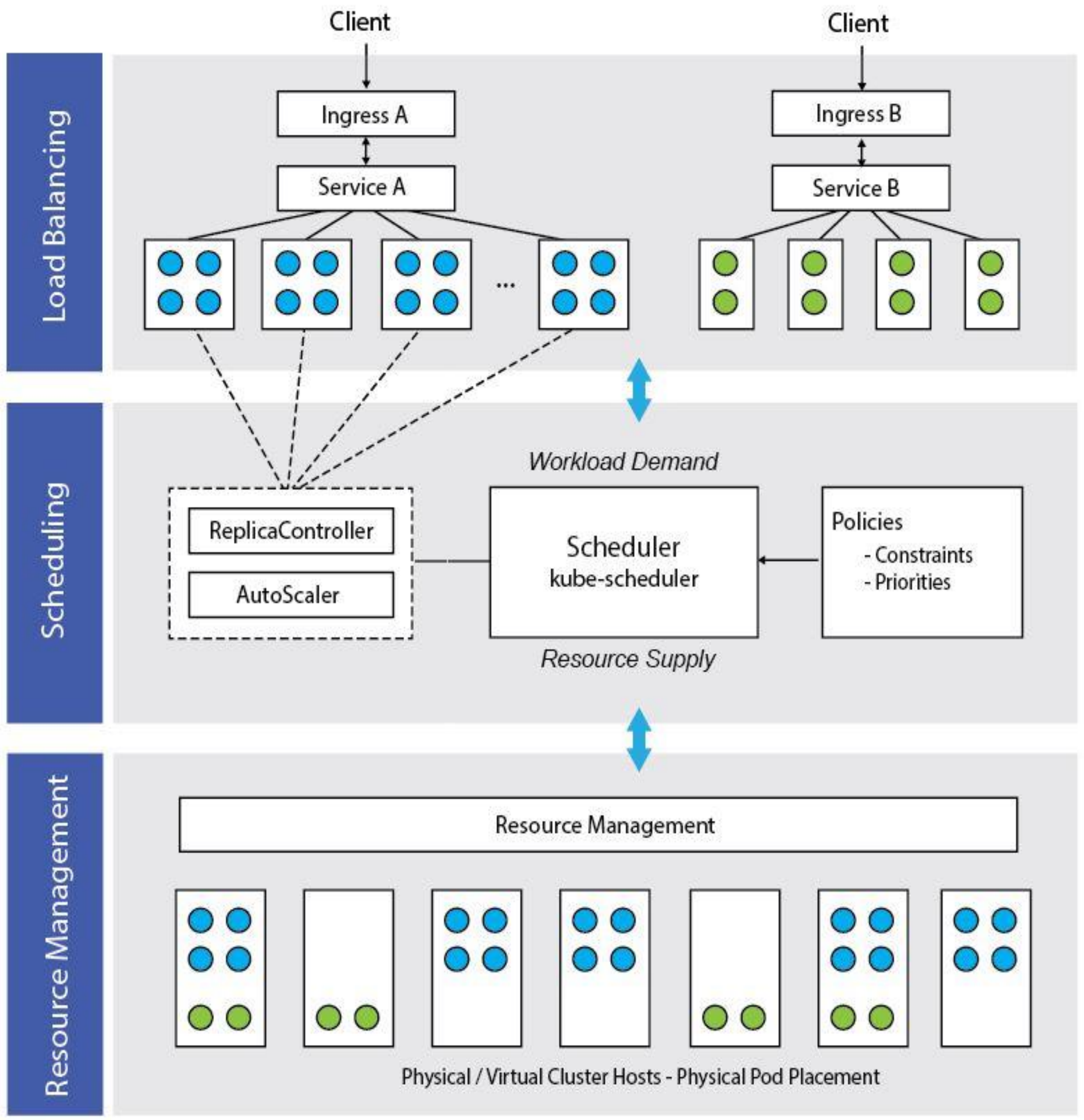
Cluster CPU usage

6.96%

Used	Total	Used	Total
1004.80 MiB	1.96 GiB	0.14 cores	2.00 cores

Kubernetes : pillars

- Resource management
  - ResourceRequest
  - ResourceLimit
  - ResourceCapacity
- Scheduling
  - role is to match resource "supply" to workload "demand"
- Load Balancing
  - Replication controllers
  - Service



Kubernetes : service discovery

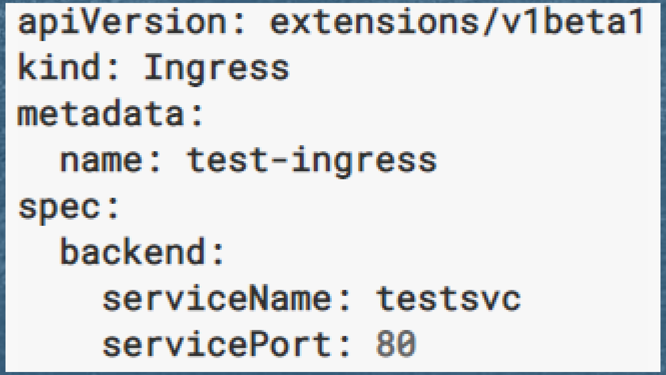
- An abstraction which defines a logical set of Pods and a policy by which to access them
- Service discovery is the process of figuring out how to connect to a service
- DNS based service discovery is a cluster add-on
- Flexible and generic way to connect to services across the cluster
- Endpoints API that is updated whenever the set of Pods in a Service changes

Kubernetes : scaling

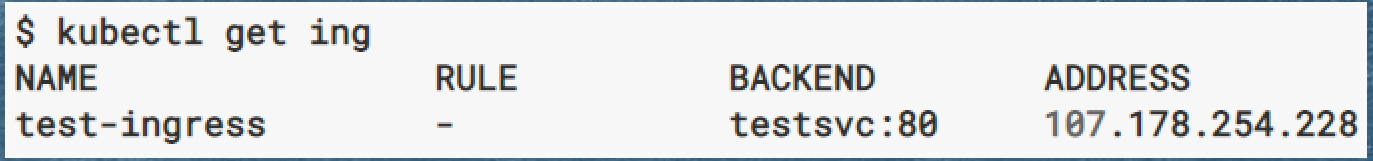
- Scaling will increase the number of Pods to the new desired state
- Scaling to zero is also possible, and it will terminate all Pods of the specified Deployment
- Services will monitor continuously the running Pods using endpoints, to ensure the traffic is sent only to available Pods
- Rolling updates are used to versioned deployments and they are reversible to last known working state

Kubernetes : Ingress

- API object that manages external access to the services in a cluster, typically HTTP
- Ingress is a collection of rules that allow inbound connections to reach the cluster services
- An Ingress controller is responsible for fulfilling the Ingress, usually with a loadbalancer
- Currently supports : GCE and nginx controllers



```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: test-ingress
spec:
  backend:
    serviceName: testsvc
    servicePort: 80
```



```
$ kubectl get ing
NAME                RULE          BACKEND        ADDRESS
test-ingress        -             testsvc:80     107.178.254.228
```

Kubernetes : reverse proxy

- kubectl proxy
  - Runs in a pod, pod to proxy on http, proxy to apiserver on https, & authentication
- apiserver proxy
  - Bastion server, user to cluster ip connector, can connect node, pod or service(lb)
- kube proxy
  - Runs on all node, udp & tcp, only used to reach services
- Apiserver
  - Differs from cluster to cluster
- cloud lb’s
  - provided by some cloud providers, created auto by service type ‘LoadBalancer’

Kubernetes : Persistence of application state

- Statefull
  - A stateful application, has several other parameters it is supposed to look after in the cluster. There are dynamic databases which, even when the app is offline or deleted, persist on the disk.
  - Persistent storage can be dynamically provisioned on demand. In Kubernetes, dynamic provisioning is configured by creating a StorageClass
- Stateless
  - A stateless application is one which depends on no persistent storage. The only thing your cluster is responsible for is the code, and other static content, being hosted on it. That's it, no changing databases, no writes and no left over files when the pod is deleted
  - Kubernetes uses the Deployment controller to deploy stateless applications as uniform, non-unique Pods. Deployments manage the desired state of your application: how many Pods should run your application, what version of the container image should run

Kubernetes : volumes

- When a Container crashes, kubelet will restart it, but the files will be lost - the Container starts with a clean state
- When running Containers together in a Pod it is often necessary to share files between those Containers. The Kubernetes Volume abstraction addresses these problems
- Mount propagation allows for sharing volumes mounted by a Container to other Containers in the same Pod, or even to other Pods on the same node.
- None - default mode
- HostToContainer - This volume mount will receive all subsequent mounts that are mounted to this volume or any of its subdirectories.
- Bidirectional - This volume mount behaves the same the HostToContainer mount. In addition, all volume mounts created by the Container will be propagated back to the host and to all Containers of all Pods that use the same volume

Kubernetes : storage backend

- Kubernetes manages these volumes and containers using a collection of metadata to describe what our intended system should be
- Metadata is essentially our desired state – Kubernetes will constantly converge to this state
- Metadata is created in the form of manifests which are small YAML documents each with a focus on a specific type of object
- Stateful container objects are :
    - PersistentVolume – the low level representation of a storage volume
    - Volume Driver – the code used to communicate with the backend storage provider
    - Pod – a running container that will consume a PersistentVolume
    - PersistentVolumeClaim – the binding between a Pod and PersistentVolume
    - StorageClass – allows for dynamic provisioning of PersistentVolumes

Kubernetes : storage backend

- We create a persistent volume manifest that kubernetes needs to know about our ebs volume
- Than we use kubectl command to submit this to kubernetes master
- Kubelet:
  - Mount and format new PersistentVolumes that are scheduled to this host
  - Start containers with PersistentVolume hostpath mounted inside the container
  - Stop containers and unmount the associated PersistentVolume
  - Constantly send metrics to the controllers about container & PersistentVolume state
- Controller:
  - to match a PersistentVolumeClaim to a PersistentVolume
  - to dynamically provision a new PersistentVolume if a claim cannot be met (if enabled)
  - in the case of EBS this is done via the AWS api from the masters
  - to attach the backend storage to a specific node if needed
  - to instruct the kubelet for a node to mount (and potentially format) the volume
  - this is done on the actual node
  - to instruct the kubelet to start a container that uses the volume

Kubernetes : cluster management

- AKS : Azure Kubernetes Service
- Google Compute Engine clusters
- Google Kubernetes Engine clusters

Cluster autoscaling

- Kubernetes scheduler to find a place to run the pod. If there is no node that has enough free capacity (or doesn't match other pod requirements) then the pod has to wait until some pods are terminated or a new node is added.
- Cluster autoscaler looks for the pods that cannot be scheduled and checks if adding a new node, similar to the other in the cluster, would help. If yes, then it resizes the cluster to accommodate the waiting pods.
- Cluster autoscaler also scales down the cluster if it notices that one or more nodes are not needed anymore for an extended period of time 10min by default
- Node management & stopping new pod creation use the drain command

Kubernetes : deployment

- You need to have a Kubernetes cluster, and the kubectl command-line tool must be configured to communicate with your cluster
- You can run an application by creating a Kubernetes Deployment object, and you can describe a Deployment in a YAML file
- Create a Deployment based on the YAML file
- Display information about the Deployment
- List the pods created by the deployment
- Display information about a pod
- You can update the deployment by applying a new YAML file

Kubernetes : deployment

- Apply the new YAML file
- Watch the deployment create pods with new names and delete the old pods
- You can increase the number of pods in your Deployment by applying a new YAML file
- Apply the new YAML file
- Verify that the Deployment has four pods
- Delete the deployment by name

Kubernetes : controllers

- ReplicaSet
- Deployment(new)

A ReplicaSet ensures that a specified number of pod replicas are running at any given time.

However, a Deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to pods along with a lot of other useful features.

Thus, It's recommended to use

Deployments instead of directly using ReplicaSets, unless you require custom update orchestration or don't require updates at all.

Kubernetes : advanced controllers

- **Workloads:** Manage containers and their lifetimes
- **Discovery & Load Balancing:** Make your applications accessible to each other or external world
- **Config & Storage:** Bind data to your containers
- **Metadata:** Adjust the behavioural data for other objects
- **Cluster:** Managing cluster state and configurations

Kubernetes : advanced controllers

- ResourceQuota : When a pod has memory requests set, your pod's QoS (Quality of Service) class is Guaranteed, and when your limit is higher than requests, QoS class is Burstable > implemented via a yaml file
- PriorityClass : It allows the higher priority pods to be scheduled earlier than the lower priority ones
- LimitRange : forget specifying CPU & Memory requests and limits for your pods, they will be in BestEffort QoS class, segregation between rogue vs healthy pods
- PodSecurityPolicy : modify both grants and restrictions in a centralized manner with Kubernetes, configure the SELinux and AppArmor rules, drop and add Linux capabilities, modify namespace sharing for PID, network, IPC, enforce the user and group of the containers and even make the container read-only

Kubernetes : crons, *time-based schedule tasks*

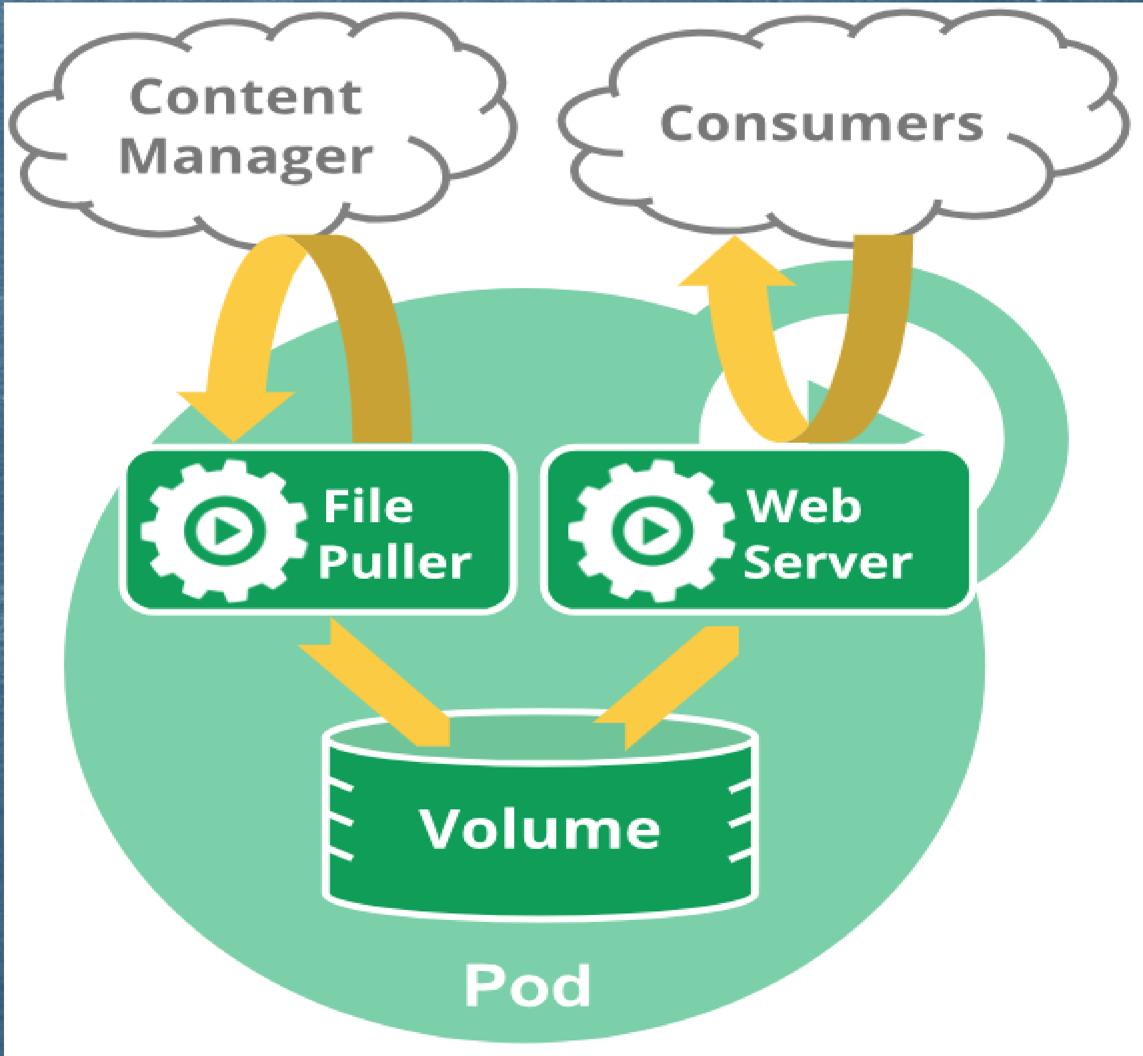
- One CronJob object is like one line of a *crontab* (cron table) file, times are denoted in UTC
- CronJob controller checks how many schedules it missed in the duration from its last scheduled time until now. If there are more than 100 missed schedules, then it does not start the job and logs the error
- if startingDeadlineSeconds is 200, the controller counts how many missed jobs occurred in the last 200 seconds

Kubernetes : pods

- Pods that run a single container. The "one-container-per-Pod" model is the most common Kubernetes use case; in this case, you can think of a Pod as a wrapper around a single container, and Kubernetes manages the Pods rather than the containers directly.

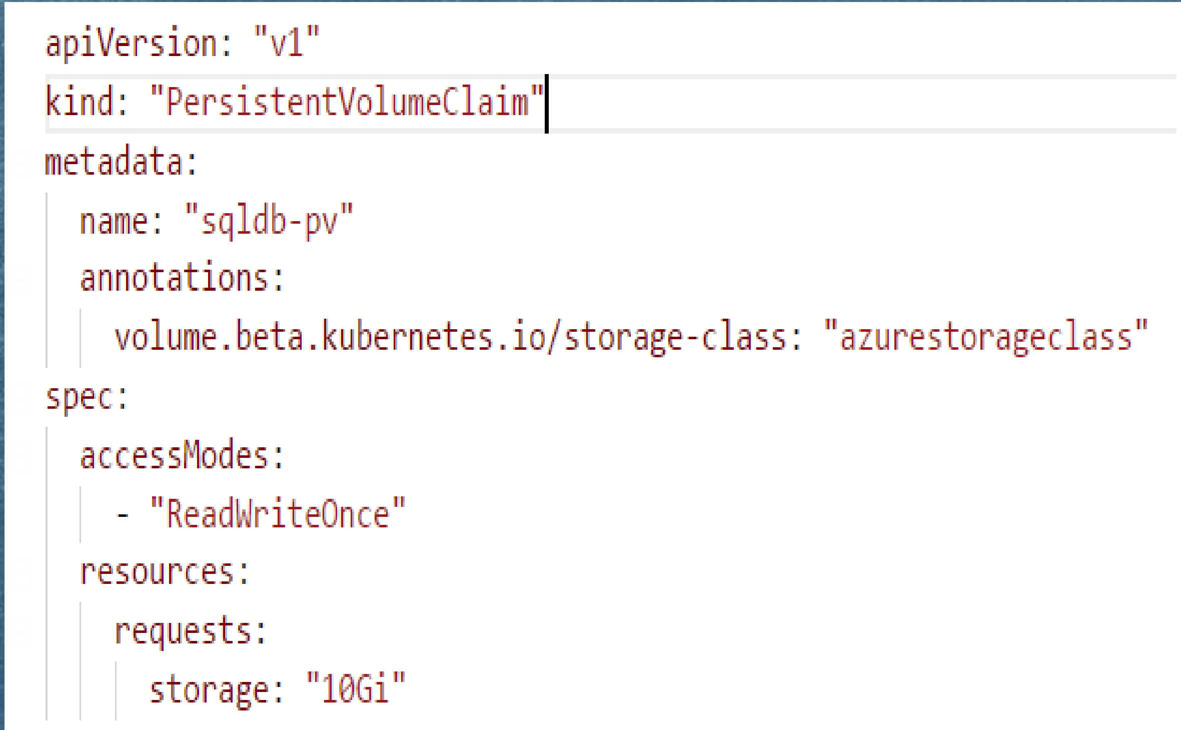
- Pods that run multiple containers that need to work together. A Pod might encapsulate an application composed of multiple co-located containers that are tightly coupled and need to share resources.

- These co-located containers might form a single cohesive unit of service–one container serving files from a shared volume to the public, while a separate "sidecar" container refreshes or updates those files. The Pod wraps these containers and storage resources together as a single manageable entity

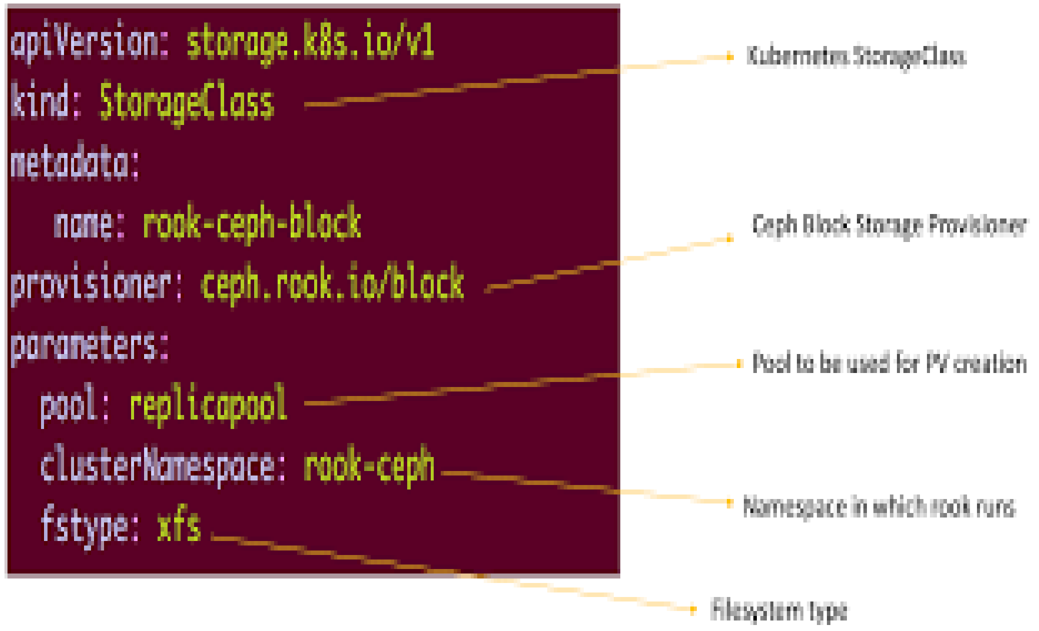


Kubernetes : storage classes

- StorageClass contains the fields provisioner, parameters, and reclaimPolicy, which are used when a PersistentVolume belonging to the class needs to be dynamically provisioned



```yaml
apiVersion: "v1"
kind: "PersistentVolumeClaim"
metadata:
  name: "sqldb-pv"
  annotations:
    volume.beta.kubernetes.io/storage-class: "azurestorageclass"
spec:
  accessModes:
    - "ReadWriteOnce"
  resources:
    requests:
      storage: "10Gi"
```

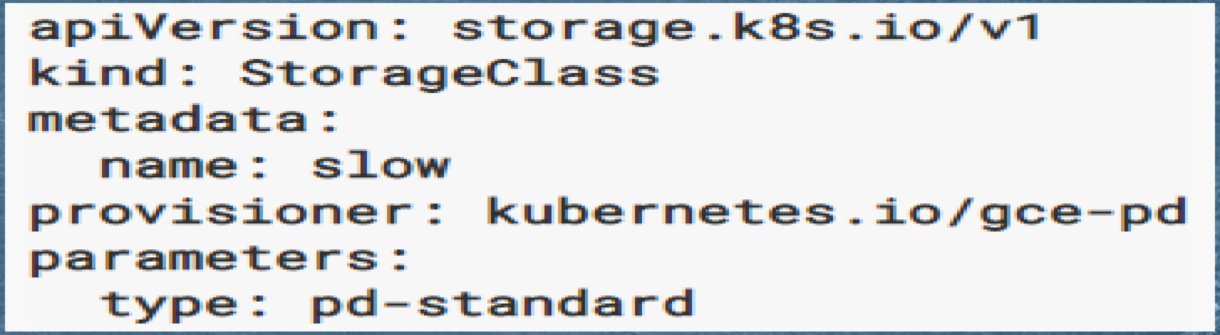


```yaml
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: rook-ceph-block
provisioner: ceph.rook.io/block
parameters:
  pool: replicapool
  clusterNamespace: rook-ceph
  fstype: xfs
```

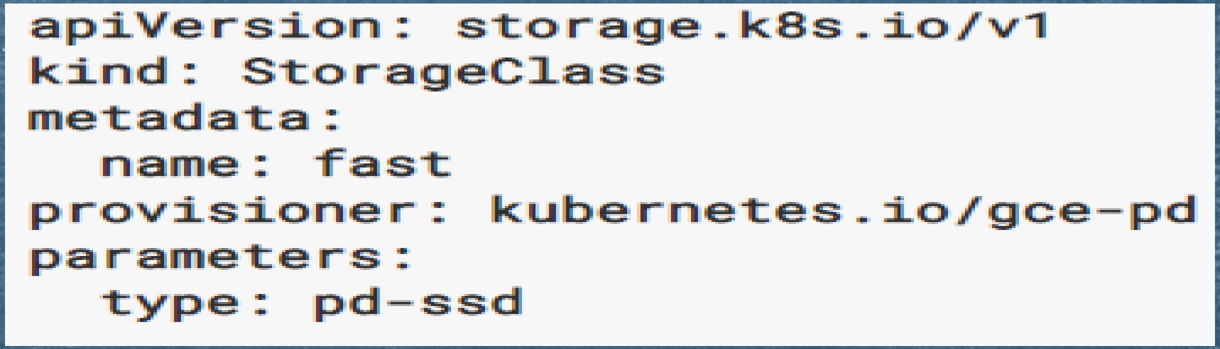
Kubernetes StorageClass
Ceph Block Storage Provisioner
Pool to be used for PV creation
Namespace in which rook runs
Filesystem type

Kubernetes : dynamic storage provisioning

- storage volumes to be created on-demand
- eliminates the need for cluster administrators to pre-provision storage
- API object StorageClass from the API group storage.k8s.io
- cluster administrator needs to pre-create one or more StorageClass objects for users. StorageClass objects define which provisioner should be used and what parameters should be passed to that provisioner when dynamic provisioning is invoked



```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: slow
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-standard
```



```
apiVersion: storage.k8s.io/v1
kind: StorageClass
metadata:
  name: fast
provisioner: kubernetes.io/gce-pd
parameters:
  type: pd-ssd
```

Kubernetes : network policies

- This is how groups of pods are allowed to communicate with each other and other network endpoints
- NetworkPolicy resources use labels to select pods and define rules which specify what traffic is allowed to the selected pods
- Network policies are implemented by the network plugin, so you must be using a networking solution which supports NetworkPolicy
- By default, pods are non-isolated; they accept traffic from any source
- Mandatory Fields: As with all other Kubernetes config, a NetworkPolicy needs apiVersion, kind, and metadata

Kubernetes : network policies

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Ingress
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector: {}
  egress:
  - {}
  policyTypes:
  - Egress
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: allow-all
spec:
  podSelector: {}
  ingress:
  - {}
```

```yaml
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: default-deny
spec:
  podSelector: {}
  policyTypes:
  - Egress
```

Kubernetes : network policies

- Mandatory Fields: NetworkPolicy needs apiVersion, kind, and metadata fields. For general information about working with config files,.
- spec: NetworkPolicy spec has all the information needed to define a particular network policy in the given namespace.
- podSelector: Each NetworkPolicy includes a podSelector which selects the grouping of pods to which the policy applies. The example policy selects pods with the label "role=db". An empty podSelector selects all pods in the namespace.
- policyTypes: Each NetworkPolicy includes a policyTypes list which may include either Ingress, Egress, or both. The policyTypes field indicates whether or not the given policy applies to ingress traffic to selected pod, egress traffic from selected pods, or both. If no policyTypes are specified on a NetworkPolicy then by default Ingress will always be set and Egress will be set if the NetworkPolicy has any egress rules.
- ingress: Each NetworkPolicy may include a list of whitelist ingress rules. Each rule allows traffic which matches both the from and ports sections. The example policy contains a single rule, which matches traffic on a single port, from one of three sources, the first specified via an ipBlock, the second via a namespaceSelector and the third via a podSelector.
- egress: Each NetworkPolicy includes a list of whitelist egress rules. Each rule allows traffic which matches both the to and ports sections. Example policy contains a single rule, which matches traffic on a single port to any destination in 10.0.0.0/24

```
apiVersion: networking.k8s.io/v1
kind: NetworkPolicy
metadata:
  name: test-network-policy
  namespace: default
spec:
  podSelector:
    matchLabels:
      role: db
  policyTypes:
  - Ingress
  - Egress
  ingress:
  - from:
    - ipBlock:
        cidr: 172.17.0.0/16
        except:
        - 172.17.1.0/24
    - namespaceSelector:
        matchLabels:
          project: myproject
    - podSelector:
        matchLabels:
          role: frontend
    ports:
    - protocol: TCP
      port: 6379
  egress:
  - to:
    - ipBlock:
        cidr: 10.0.0.0/24
    ports:
    - protocol: TCP
      port: 5978
```

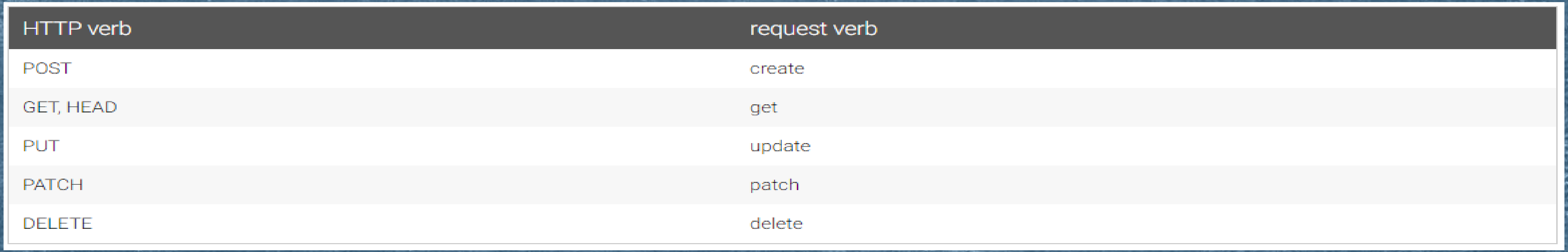
Kubernetes : Securing a cluster

Kubernetes is entirely API driven

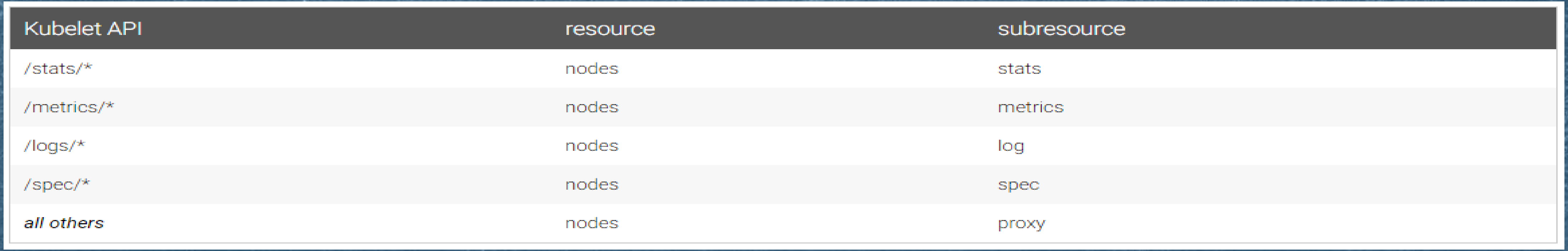
- Transport Level Security (TLS) for all API traffic
  - API communication in the cluster is encrypted by default with TLS, and the majority of installation methods will allow the necessary certificates to be created and distributed to the cluster components
- API Authentication
  - Typically service accounts or use x509 client certificates, and they are created automatically at cluster startup or are setup as part of the cluster installation
- API Authorization
  - Role-Based Access Control (RBAC) component that matches an incoming user or group to a set of permissions bundled into roles. These permissions combine verbs (get, create, delete) with resources (pods, services, nodes) and can be namespace or cluster scoped

Kubernetes : Controlling access to the Kubelet

- Kubelet expose HTTPS endpoints which grant powerful control over the node and containers. By default Kubelets allow unauthenticated access to this API



HTTP verb	request verb
POST	create
GET, HEAD	get
PUT	update
PATCH	patch
DELETE	delete



Kubelet API	resource	subresource
/stats/*	nodes	stats
/metrics/*	nodes	metrics
/logs/*	nodes	log
/spec/*	nodes	spec
all others	nodes	proxy

Kubernetes : Controlling capabilities of workload / user

- Limiting resource usage on a cluster
  - Resource quota limits the number or capacity of resources granted to a namespace. This is most often used to limit the amount of CPU, memory, or persistent disk a namespace can allocate
  - Limit ranges restrict the maximum or minimum size of some of the resources above, to prevent users from requesting unreasonably high or low values for commonly reserved resources like memory

- A pod definition contains a security context that allows it to request access to running as a specific Linux user on a node (like root), access to run privileged or access the host network
- The network policies for a namespace allows application authors to restrict which pods in other namespaces may access pods and ports within their namespaces
- When running Kubernetes on a cloud platform limit permissions given to instance credentials, use network policies to restrict pod access to the metadata API, and avoid using provisioning data to deliver secrets

Kubernetes : Protecting cluster components from compromise

- Restrict access to etcd
  - Write access to the etcd backend for the API is equivalent to gaining root on the entire cluster
  - It is recommended to isolate the etcd servers behind a firewall that only the API servers may access
  - Using separate etcd instances for non-master components or using etcd ACLs to restrict read and write access to a subset of the keyspace is strongly recommended
- Enable audit logging
- Restrict access to alpha or beta features
- Rotate infrastructure credentials frequently
- Review third party integrations before enabling them
- Encrypt secrets at rest
  - Encryption at rest, an alpha feature that will encrypt Secret resources in etcd, preventing parties that gain access to your etcd backups from viewing the content of those secrets

Kubernetes : Authentication, Authorization & Access Control

- Kubernetes uses client certificates, bearer tokens, an authenticating proxy, or HTTP basic auth to authenticate API requests through authentication plugins. As HTTP requests are made to the API server, plugins attempt to associate the following attributes with the request
  - Username: a string which identifies the end user. Common values might be kube-admin or jane@example.com.
  - UID: a string which identifies the end user and attempts to be more consistent and unique than username.
  - Groups: a set of strings which associate users with a set of commonly grouped users.
  - Extra fields: a map of strings to list of strings which holds additional information authorizers may find useful.

Kubernetes : Authentication, Authorization & Access Control

- Kubernetes authorizes API requests using the API server. It evaluates all of the request attributes against all policies and allows or denies the request. All parts of an API request must be allowed by some policy in order to proceed. This means that permissions are denied by default.
  - user - The user string provided during authentication.
  - group - The list of group names to which the authenticated user belongs.
  - extra - A map of arbitrary string keys to string values, provided by the authentication layer.
  - API - Indicates whether the request is for an API resource.
  - Request path - Path to miscellaneous non-resource endpoints like /api or /healthz.
  - API request verb - API verbs get, list, create, update, patch, watch, proxy, redirect, delete, and deletecollection are used for resource requests.
  - HTTP request verb - HTTP verbs get, post, put, and delete are used for non-resource requests.
  - Resource - The ID or name of the resource that is being accessed
  - Subresource - The subresource that is being accessed
  - Namespace - The namespace of the object that is being accessed
  - API group - The API group being accessed. An empty string designates the core API group

Kubernetes : Authentication, Authorization & Access Control

- RBAC uses the rbac.authorization.k8s.io API group to drive authorization decisions, allowing admins to dynamically configure policies through the Kubernetes API
  - Role can only be used to grant access to resources within a single namespace. Here’s an example Role in the “default” namespace that can be used to grant read access to pods
  - ClusterRole can be used to grant the same permissions as a Role, but because they are cluster-scoped, they can also be used to grant access to :
    - cluster-scoped resources (like nodes)
    - non-resource endpoints (like “/healthz”)
    - namespaced resources (like pods) across all namespaces (needed to run kubectl get pods --all-namespaces, for example)

Kubernetes : Control plane high availability

- Different approaches to setting up a highly available Kubernetes cluster using kubeadm
  - With stacked masters. This approach requires less infrastructure. etcd members and control plane nodes are co-located.
  - With an external etcd cluster. This approach requires more infrastructure. The control plane nodes and etcd members are separated

- Infrastructure:
  - Three machines that meet kubeadm’s minimum requirements for the masters
  - Three machines that meet kubeadm’s minimum requirements for the workers
  - Full network connectivity between all machines in the cluster
  - SSH access from one device to all nodes in the system
  - sudo privileges on all machines

- For the external etcd cluster only, you also need:
  - Three additional machines for etcd members

Kubernetes : Cluster monitoring

Resource metrics pipeline

limited set of metrics related to cluster components such as the HorizontalPodAutoscaler controller, as well as the kubectl top utility. These metrics are collected by metrics-server and are exposed via the metrics.k8s.io API. metrics-server discovers all nodes on the cluster and queries each node's Kubelet for CPU and memory usage. The Kubelet fetches the data from cAdvisor. metrics-server is a lightweight short-term in-memory store

Full metrics pipelines

such as Prometheus, gives you access to richer metrics. In addition, Kubernetes can respond to these metrics by automatically scaling or adapting the cluster based on its current state, using mechanisms such as the Horizontal Pod Autoscaler. The monitoring pipeline fetches metrics from the Kubelet, and then exposes them to Kubernetes via an adapter by implementing either the custom.metrics.k8s.io or external.metrics.k8s.io API

Kubernetes : Cluster monitoring

cAdvisor

- ✓ cAdvisor is an open source container resource usage and performance analysis agent
- ✓ cAdvisor is integrated into the Kubelet binary
- ✓ cAdvisor auto-discovers all containers in the machine and collects CPU, memory, filesystem, and network usage statistics
- ✓ cAdvisor also provides the overall machine usage by analyzing the 'root' container on the machine
- ✓ cAdvisor exposes a simple UI for on-machine containers on port 4194

Also monitoring possible using google stackdriver

prometheus

- ✓ Prometheus Operator simplifies Prometheus setup on Kubernetes, and allows you to serve the custom metrics API using the Prometheus adapter
- ✓ Prometheus provides a robust query language and a built-in dashboard for querying and visualizing your data
- ✓ Prometheus is also a supported data source for Grafana

Also monitoring possible using dynatrace

Kubernetes : troubleshooting

- https://stackoverflow.com/questions/tagged/kubernetes
- https://stackoverflow.com/questions/tagged/google-container-engine
- https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux_atomic_host/7/html/getting_started_with_kubernetes/troubleshooting_kubernetes
- https://www.weave.works/blog/debugging-and-troubleshooting-microservices-in-kubernetes
- https://vmware.github.io/vsphere-storage-for-kubernetes/documentation/troubleshooting.html
- https://docs.bitnami.com/kubernetes/how-to/troubleshoot-kubernetes-deployments
- https://docs.projectcalico.org/v2.0/getting-started/kubernetes/troubleshooting
- https://console.bluemix.net/docs/containers/cs_troubleshoot.html#cs_troubleshoot
- https://coreos.com/tectonic/docs/latest/troubleshooting/troubleshooting.html
- https://rancher.com/docs/rancher/v2.x/en/installation/ha/rke-add-on/troubleshooting/generic-troubleshooting
- https://cloud.google.com/kubernetes-engine/docs/support
- https://docs.microsoft.com/en-us/virtualization/windowscontainers/kubernetes/common-problems

Thank you and do let me know if you need

further digs

☺

godspeed to your success