Cloud Computing

Virtualization

Zeinab Zali

References: (1) Mastering KVM Virtualization, Vedran Dakic, 2020
(2) Linux Containers and Virtualization A Kernel Perspective, Shashank Mohan Jain, 2020
(3) Core Kubernetes, Jay Vyas, 2022
(4) Cloud Computing Theory and Practice, second edition
(5) Kubernetes in Action, Marko Luksa, 2020
(6) Slides: Docker and Kubernetes: The Practical Guide, Maximilian schuvarzmuller

ECE Department, Isfahan University of Technology

Resource Sharing in clouds

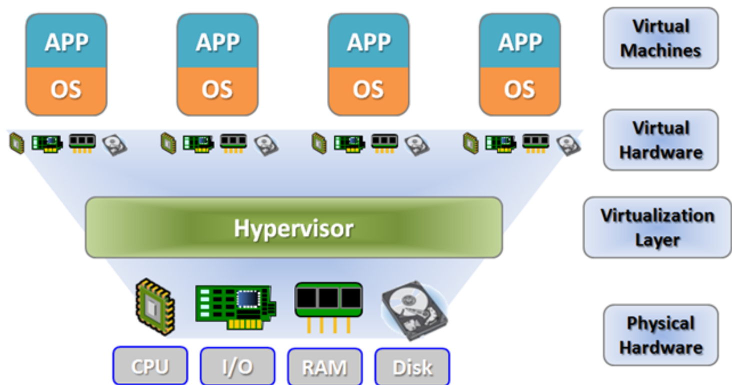
Economics of Clouds requires sharing resources How do we share a physical computer among multiple users?

- Answer: Abstraction
- Abstraction: what a generic computing resource should look like, Then providing this abstract model to many users
- Abstraction enables virtualization

What is virtualization?

Virtualization is a concept that creates virtualized resources and maps them to physical resources

- Virtualization can be done using specific software functionality (hypervisor)



Virtualization terms?

- **Virtual Machine(M):** an isolated environment with access to a subset of physical resources of the computer system
- **Host OS:** The OS that we are running on base physial system
- **Guest OS:** The OS that we are running in a virtual machine

Virtualization in Clouds

Clouds are based on Virtualization

- They offer services based mainly on virtual machines, remote procedure calls, and client/servers
- The instantaneous demands for resources of the applications running concurrently are likely to be different and complement each other

Virtualization benefits

- supporting portability, improve efficiency, increase reliability, and shield the user from the complexity of the system
- providing more freedom for the system resource management because VMs can be easily migrated
- allowing a good isolation of applications from one another

Types of Virtualization

If we are talking about how we're virtualizing a virtual machine as an object, there are different types of virtualization:

- **Full Virtualization:** a virtual machine is used to simulate regular hardware while not being aware of the fact that it's virtualized (we don't have to modify the guest OS)
- **Software-based:** using binary translation to virtualize the execution of sensitive instruction sets while emulating hardware using software
- **Hardware-based:** removing binary translation from the equation while interfacing with a CPU's virtualization features (AMD-V, Intel VT)

Types of Virtualization

- **Paravirtualization:** the guest OS understands the fact that it's being virtualized and needs to be modified, along with its drivers, so that it can run on top of the virtualization solution.
- **Hybrid virtualization:** the guest OS can be run unmodified (full), and the fact that we can insert additional paravirtualized drivers into the virtual machine to work with some specific aspects of virtual machine work (most often, I/O-intensive memory workloads)
- **Container-based virtualization:** a type of application virtualization that uses containers. A container is an object that packages an application and all its dependencies

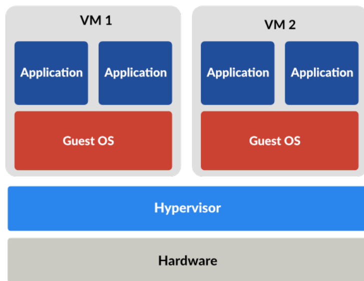
Hipervisor (Virtual Machine Manager)

Virtual Machine Manager (VMM) or hypervisor is a piece of software that is responsible for monitoring and controlling virtual machines or guest OSes.

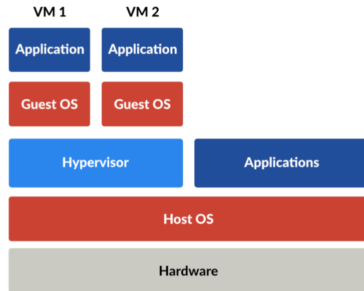
- providing virtual hardware
- virtual machine life cycle management
- migrating virtual machines
- allocating resources in real time
- responsible for efficiently controlling physical platform resources, such as memory translation and I/O mapping.

Hipervisor Types

- **Type 1:** If the VMM/hypervisor runs directly on top of the hardware, its generally considered to be a type 1 hypervisor
- **Type 2:** If there is an OS present, and if the VMM/hypervisor operates as a separate layer, it will be considered as a type 2 hypervisor



Type 1



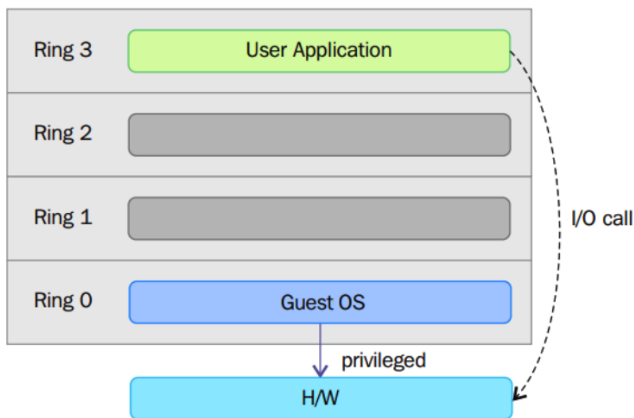
Type 2

Hardware facilities for virtualization

- **Second-Level Address Translation (SLAT) (EPT intel or RVI AMD):** a CPU feature for adding a translation lookaside buffer (TLB) that contains a real-time updated cache of virtualized memory addresses and their corresponding physical addresses
- **Intel VT or AMD-V support:** supporting hardware virtualization extensions and full virtualization
- **Long mode support:** 64-bit CPU
- **Input/Output Memory Management Unit (IOMMU) virtualization (Intel VT-d or AMD-Vi):** allowing virtual machines to access peripheral hardware directly (graphics cards, storage controllers, network devices, and so on)
- **others:** Single Root Input Output Virtualization (SR/IOV), PCI passthrough, Trusted Platform Module (TPM) support

Protection Rings

- All the applications run in ring 3 (user mode)
- The kernels run in ring 0 ,i.e. the privileged mode(kernel mode)
- 1 and 2 are mostly unused



Trap and Emulate

- Most of the time, the software runs exactly as it would on a real machine, but if the operating system (OS) attempts to perform a privileged operation, a hardware trap will occur
- Since the VMM executes in supervisor mode, it can catch this hardware exception, inspect the state of the OS that caused it
- Then VMM emulate the behavior that would have occurred on real hardware
- problem: executing some sensitive instructions may not causes a trap in nonprotected mode

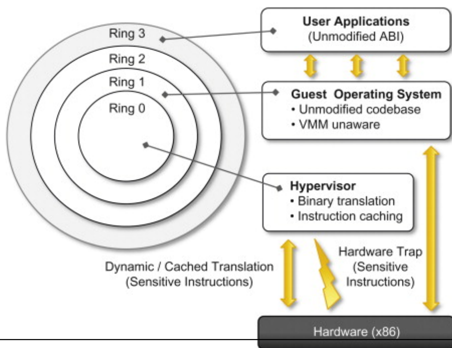
Binary Translation

- If guest vCPU is in user mode, guest can run instructions natively, whereas if guest vCPU is in kernel mode, VMM checks every instruction (and does not wait for a trap!)
- Non-sensitive instructions run normally but sensitive instructions are translated appropriately
- Performance of this method is worse than trap-and-emulate since all codes of the guest kernel is inspected by the VMM

Dynamic Binary Translation

For each block of codes, dynamic BT translates critical instructions, if any, into some privilege instructions, which will trap to VMM for further emulation

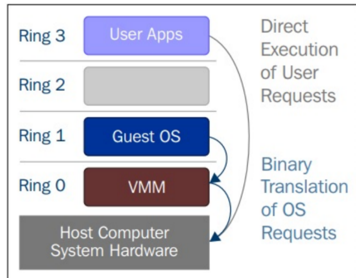
- To improve performance, the equivalent set of instruction is cached so that translation is no longer necessary for further occurrences of the same instructions



Full virtualization with Dynamic Binary Translation

In full virtualization, privileged instructions are emulated to overcome the limitations that arise from the guest OS running in ring 1 and the VMM (or host OS) running in ring 0

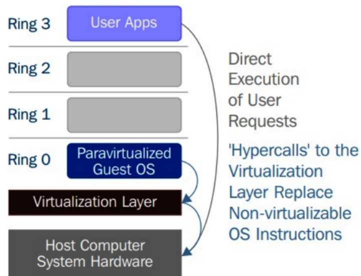
- It relies on techniques such as dynamic binary translation to trap and virtualize the execution of certain sensitive instructions



ParaVirtualization

In para virtualization, the guest OS is modified or patched for virtualization

- The modified kernel of the guest OS is able to communicate with the underlying hypervisor via special calls (hypercall)
  - These special calls are equivalent to system calls generated by an application to a non virtualized OS



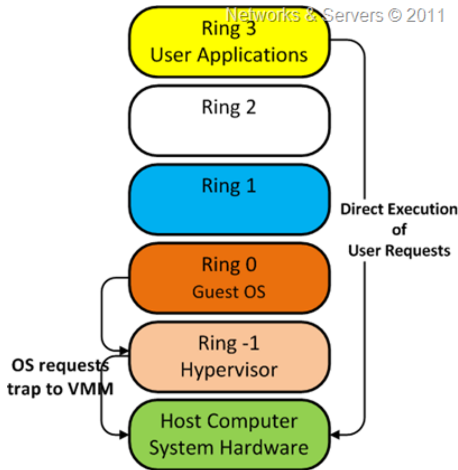
Hardware-assisted Virtualization (Native Virtualization)

Hardware-assisted virtualization is a platform virtualization method designed to efficiently use full virtualization with the hardware capabilities.

- These extensions allow the VMM/hypervisor to run a guest OS that expects to run in kernel mode, in lower privileged rings
- Hardware-assisted virtualization not only proposes new instructions but also introduces a new privileged access level, called ring -1, where the hypervisor/VMM can run
  - guest virtual machines can run in ring 0
- With hardware-assisted virtualization, the OS has direct access to resources without any emulation or OS modification

Hardware-assisted Virtualization (Native Virtualization)

With hardware-assisted virtualization, the OS has direct access to resources without any emulation or OS modification



libvirt-QEUME-KVM

- **QEUME:** it has two modes:
  - Emulator: QEMU emulates CPUs through dynamic binary translation techniques and provides a set of device models
  - Virtualizer: This is the mode where QEMU executes the guest code directly on the host CPU, thus achieving native performance
- **KVM :** a common kernel module called kvm.ko and also hardware-based kernel modules such as kvm-intel.ko (Intel-based systems)
  - KVM turns the Linux kernel into a hypervisor, thus achieving virtualization
  - KVM exposes a device file called /dev/kvm to applications to create, initialize, and manage the kernel-mode context of virtual machines.
- **libvirt:** It is a library to provide a common and stable layer for managing virtual machines running on a hypervisor through KVM

Xen

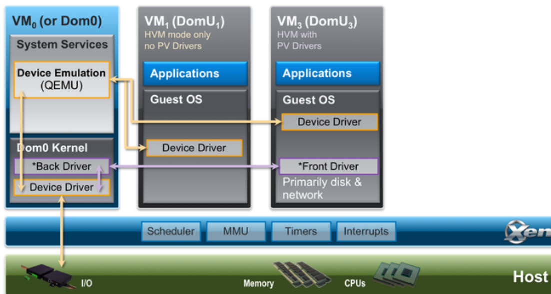
Xen is a free and open source type 1 hypervisor widely employed in virtualization environment.

- This hypervisor adopts paravirtualization and Hardware-assisted full virtualization
- Xen can scale up to 4095 host CPUs with 16Tb of RAM
- Using paravirtualization, Xen supports maximum of 512 VCPU with 512GB RAM per guest
- Using hardware Virtualization, Xen supports a maximum of 128 VCPUs with 1TB RAM per guest.

Xen Architecture

The whole environment is divided into domains or virtual machines:

- Dom 0 hosts the most important OS and is a privileged domain responsible for the creation of other new domains
- All other virtual machines are Dom U which is the guest operating system

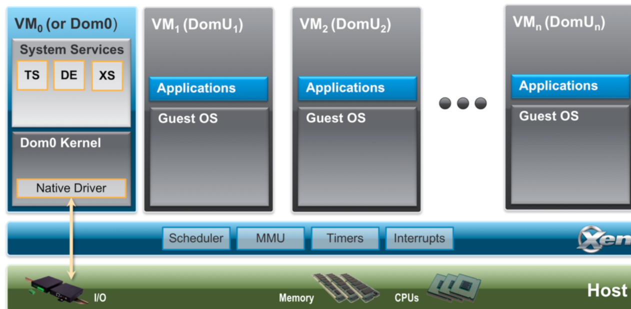


Xen Architecture

- domain0 is created at boot time which is permitted to use the control interface
- Guest VMs are totally isolated from the hardware: in other words, they have no privilege to access hardware or I/O functionality. Thus, they are also called unprivileged domain (or DomU).
- Linux distributions that are based on Linux kernels newer than Linux 3.0 are Xen Project-enabled and usually include packages that contain the hypervisor and Tools

Dom0 Functions

- **System Services:** such as XenStore/XenBus (XS) for managing settings, the Toolstack (TS) exposing a user interface to a Xen based system, Device Emulation (DE) which is based on QEMU in Xen based systems



Dom0 Functions

- **Native Device Drivers:** Dom0 is the source of physical device drivers and thus native hardware support for a Xen system
- **Virtual Device Drivers:** Dom0 contains virtual device drivers (also called backends).
- **Toolstack:** allows a user to manage virtual machine creation, destruction, and configuration. The toolstack exposes an interface that is either driven by a command line console, by a graphical interface or by a cloud orchestration stack such as OpenStack or CloudStack.

Xen Control Interactions

- synchronous calls from a domain to Xen may be made using a hypercall
  - A software trap into the hypervisor to perform a privileged operation
- notifications are delivered to domains from Xen using an asynchronous event mechanism
  - similar to traditional Unix signals, there are only a small number of events, each acting to flag a particular type of occurrence
  - Examples: events are used to indicate that new data has been received over the network, or that a virtual disk request has completed.

Xen Memory Management

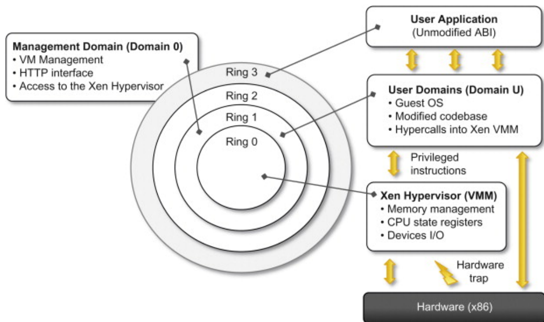
- Each time a guest OS requires a new page table it allocates and initializes a page from its own memory reservation and registers it with Xen
- At this point the OS must relinquish direct write privileges to the page-table memory
    - all subsequent updates must be validated by Xen

Xen CPU Management

- the insertion of a hypervisor below the operating system violates the usual assumption that the OS is the most privileged entity in the system.
- In order to protect the hypervisor from OS misbehavior (and domains from one another) guest OSes must be modified to run at a lower privilege level.
- Efficient virtualizion of privilege levels is possible on x86 because it supports four distinct privilege levels in hardware

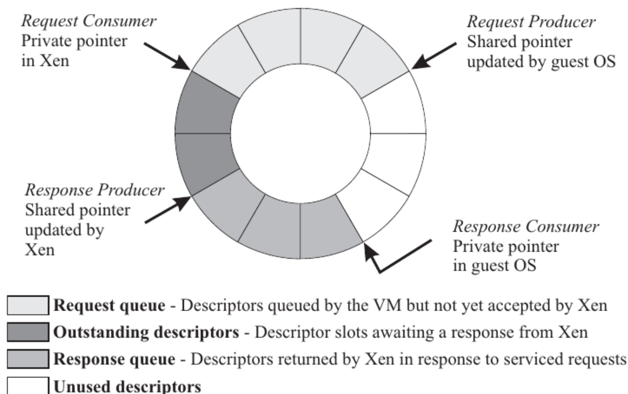
Xen CPU Management

- 4 distinct privilege levels
  - 0 for most privileged, 3 for least privileged
- Any guest OS can be ported to Xen by modifying it to execute in ring 1
  - This prevents the guest OS from directly executing privileged instructions, yet it remains safely isolated from applications running in ring 3



Xen IO Management

- I/O data is transferred to and from each domain via Xen, using shared-memory and asynchronous buffer descriptor rings.
  - an event-delivery mechanism instead of hardware interrupts for sending asynchronous notifications to a domain.



Other VMMs

- hyper-v
- VMware's ESXi Server
- proxmox

Containers

What is a Container

A standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another

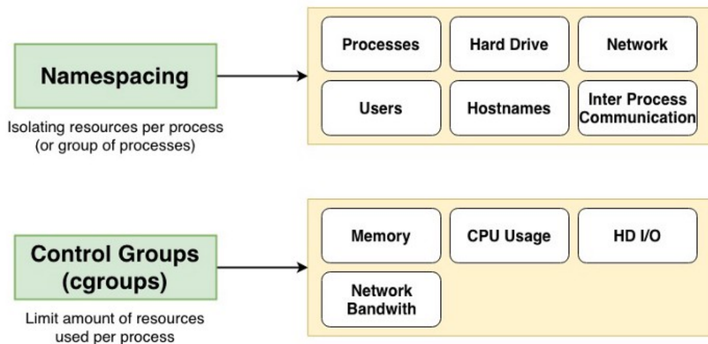
solving the “it works on my machine” headache

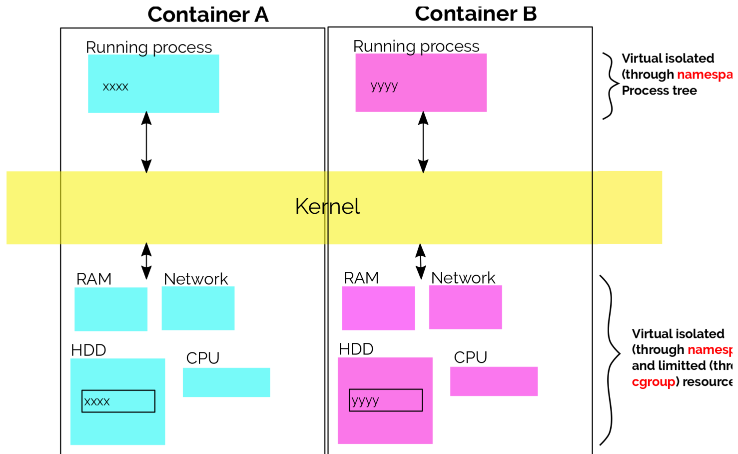
A Brief History of Containers

- (1979) **chroot** systemcall: the beginning of process isolation
- (2000) FreeBSD Jails
- (2001) Linux VServer
- (2004) Solaris Containers
- (2005) Open VZ
- (2006) Process Containers
- (2007) **cgroups**: merged to Linux kernel 2.6.24
- (2008) LXC (LinuX Containers): The first, most complete implementation of Linux container manager
- (2013) **Docker**

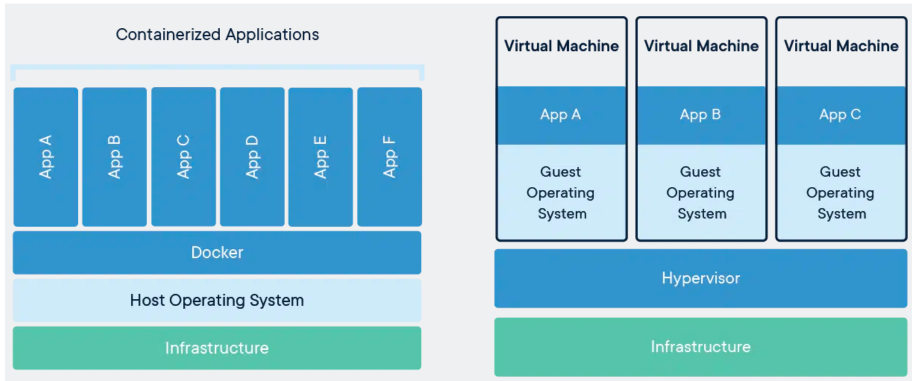
Basic blocks provided by kernel

- namespaces
- cgroups
- Copy-On-Write filesystem
- bind mounts (docker)




Container A
Container B
Running process
xxxx
Running process
yyyy
Virtual isolated
(through namespa
Process tree
Kernel
RAM
Network
HDD
CPU
xxxx
RAM
Network
HDD
CPU
yyyy
Virtual isolated
(through namesp
and limitted (thr
cgroup) resource

Container vs Virtual Machine



Namespaces

- A namespace is a logical isolation within the Linux kernel.
- A namespace controls visibility within the kernel.
  - All the controls are defined at the process level, that means a namespace controls which resources within the kernel a process can see
- By isolation, we mean that there should be a kind of sandboxing of the individual application, so that certain resources in the application are confined to that sandbox
- The technique to achieve such sandboxing is done by a specific data structure in the Linux kernel, called the namespace
- related system calls: clone, unshare, setns, netns

Namespace types

- **UTS:** This namespace allows a process to see a separate host name
- **PID:** The processes within the PID namespace have a different process tree (/proc)
- **Mount:** It controls which mount points a process should see
  - Apart from mount, there is a bind mount, which allows a directory (instead of a device) to be mounted at a mount point
- **Network:** the process within the network namespace will see different network interfaces, routes, and iptables

Namespace types

- **IPC:** This namespace scopes IPC constructs such as POSIX message queues
  - Between two processes within the same namespace, IPC is enabled, but it will be restricted if two processes in two different namespaces try to communicate over IPC
- **Cgroup:** Without this restriction, a process could peek at the global cgroups via the /proc/self/cgroup hierarchy.
  - This namespace effectively virtualizes the cgroup itself
- **Time:** The time namespace has two main use cases:
  - Changes the date and time inside a container Adjusts the clocks for a container restored from a checkpoint

cgroups

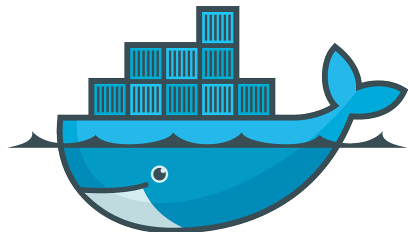
We need a way to introduce resource controls for processes within the namespace; this is achieved using a mechanism called control groups (cgroups), that provides:

- Resource limiting: groups can be set to not exceed a configured limit for CPU, memory, I/O, network
- Prioritization: some groups may get a larger share of CPU utilization[14] or disk I/O throughput
- Accounting: measures a group's resource usage, which may be used, for example, for billing purposes
- Control freezing groups of processes, their checkpointing and restarting

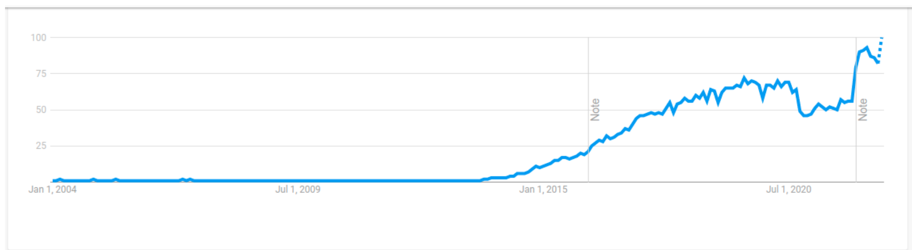
Linux Isolation Demo

What is Docker?

- Docker is a human-friendly container implementation developed and popularized by Docker Inc. in 2013



Interest over Time

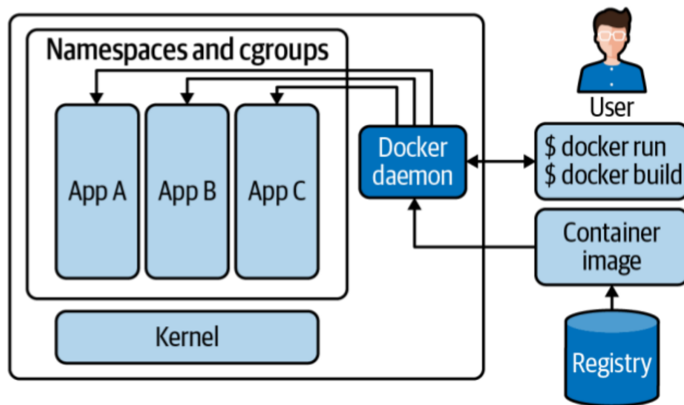


https://trends.google.com

Install on Ubuntu

Install from here: https://docs.docker.com/engine/install/ubuntu/

A closer look



Docker Components

- **Docker daemon (dockerd):** It listens for Docker API requests and manages Docker objects such as images, containers, networks, and volumes
- **Docker API:** APIs specify interfaces that programs can use to talk to and instruct the Docker daemon (a RESTful API accessed by an HTTP client)
- **Docker client (docker):** It uses Docker APIs to control or interact with the Docker daemon through scripting or direct CLI commands

Docker Container

What is a docker container? It's a running instance of the image

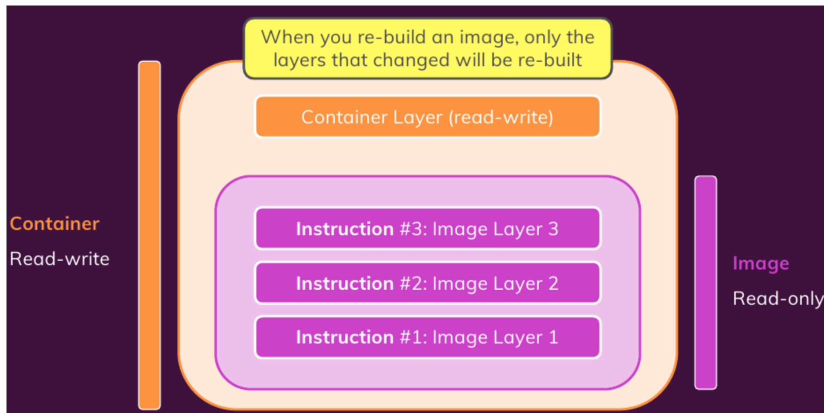
- Isolated namespace
- (Possibly) Limited resources
- Could share resources with host

Docker Image

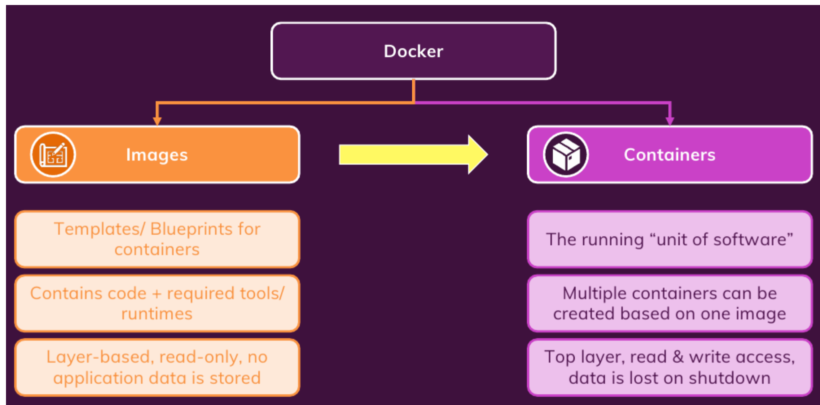
What is a docker image? It is a read-only template with instructions for creating a Docker container

- Each container is a running OCI (OpenContainers/Image-spec) image
- The OCI specification is a standard way to define an image that can be executed by a program such as Docker
- It is ultimately a tarball with various layers
- Each of the tarballs inside an image contains such things as Linux binaries and application files.
- When we run a container, the container runtime (such as Docker, containerd, or CRI-O) takes the image, unpacks it, and starts a process on the host system that runs the image contents.

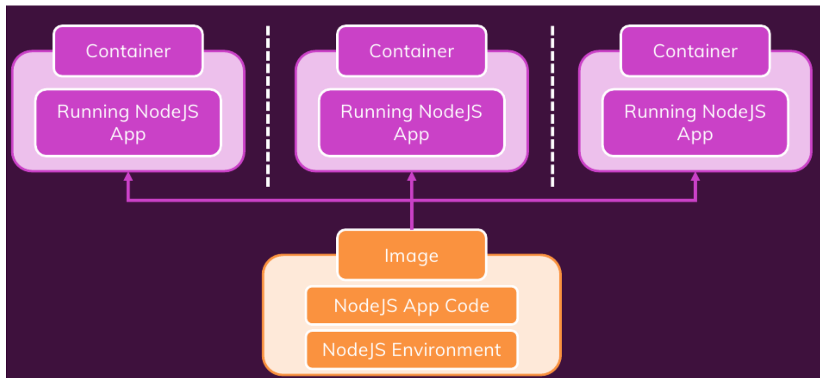
Docker Image Layers



Container vs Images



Multiple Container from One Images

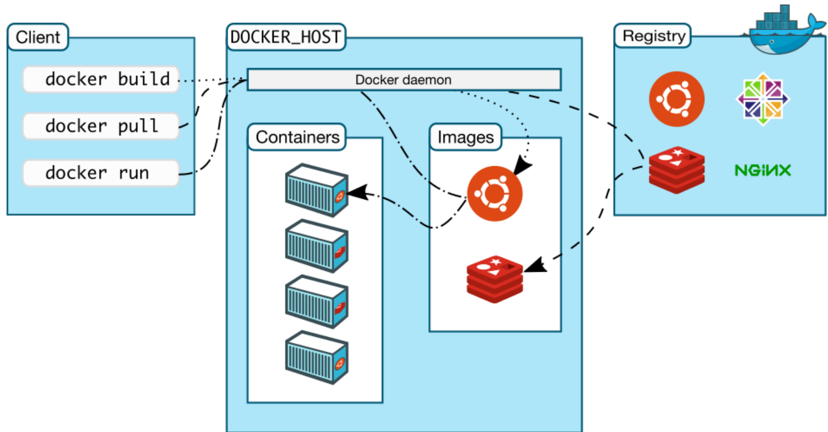


Dockerfile

Dockerfile describes the application and tells Docker how to build it into an image

- FROM: To specify the base image
  - All Dockerfiles start with the FROM instruction. This will be the base layer of the image, and the rest of the app will be added on top as additional layers
- COPY: To copy a file/folder into the image
- RUN: To run a command during image build proccess
- CMD: The command that runs the container

Docker Architecture



Docker commands

Some useful docker commands:

- docker build
- docker run
- docker images
- docker ps
- docker rmi/rm

IR repository

- Add
  - "registry-mirrors": ["https://registry.docker.ir"]
- Restart:
  - sudo systemctl daemon reload
  - sudo systemctl restart docker

Demo

What is Environment Variables

An environment variable is a named object that contains data used by one or more applications

- a simple way to share configuration settings between multiple applications and processes
- useful commands:
  - echo $VAR_NAME
  - export VAR_NAME="value"
  - VAR_NAME="value"

Environment Variables Example

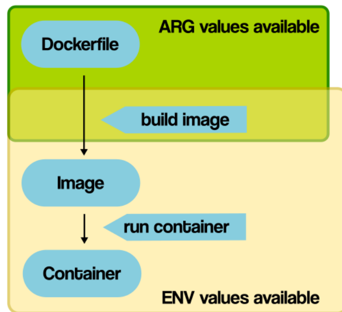
The **PATH** variable is an environment variable containing a list of paths that Linux search for executable when running a command

```
zeinabzali@HadoopMaster:~
File Edit View Search Terminal Help
➜  ~ echo $PATH
/usr/local/sbin:/usr/bin:/sbin:/usr/games:/usr/local/games:/snap/bin:/usr/local/bin:/usr/sbin:/bin
➜  ~ cp
cp: missing file operand
Try 'cp --help' for more information.
➜  ~ export PATH=/usr/local/sbin:/usr/bin:/sbin:/usr/games:/usr/local/games:/snap/bin:/usr/local/bin:/usr/sbin
➜  ~ cp
zsh: command not found: cp
➜  ~ export PATH=$PATH:/bin
➜  ~ cp
cp: missing file operand
Try 'cp --help' for more information.
➜  ~
```

Environment variables in docker

Environment variables are a convenient way to externalize application configuration in building Docker containers

- Two kinds of environment variables in docker:
  - build-time
  - run-time



Build Time Variables in Docker

These variables are used in docker file, so it will be configured differently depending on the environment that was used to build a container

- in Dockerfile:

```
ARG some_variable_name
# or with a hard-coded default:
#ARG some_variable_name=default_value

RUN echo "Oh dang look at that $some_variable_name"
# you could also use braces - ${some_variable_name}
```

- in build time:

```
$ docker build --build-arg some_variable_name=a_value
```

Runtime Variables in Docker

Setting variables for Docker containers can be done in three main ways:

- CLI arguments
- .env config files
- through docker-compose

Runtime variables in docker

Setting variables for Docker containers through CLI arguments:

- CLI arguments

```
sudo docker run
-e POSTGRES_USER='postgres'
-e POSTGRES_PASSWORD='password'
...
```

```
// set variable
POSTGRES_PASSWORD='password'

// use it later
docker run -e POSTGRES_PASSWORD -e POSTGRES_USER ...
```

Runtime variables in docker

Setting variables for Docker containers through .env files:

- .env config files
  - creating .env file

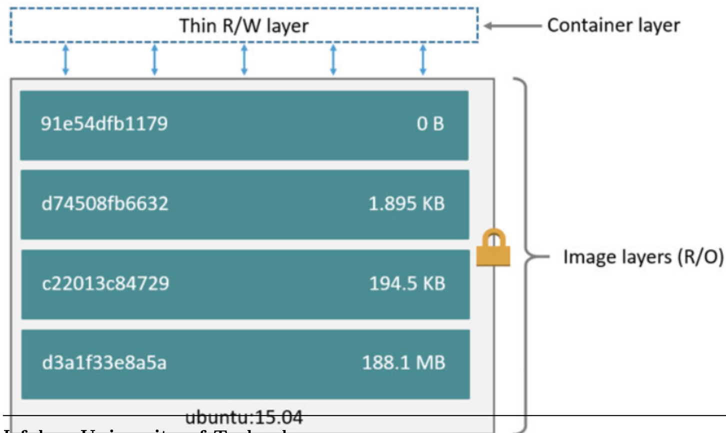
```
POSTGRES_PASSWORD='password'
POSTGRES_USER='postgres'
APPLICATION_URL='example.com'
```

  - pass it to docker run with the –env-file flag

```
docker run --env-file ./envfile ...
```

Persisting data in docker

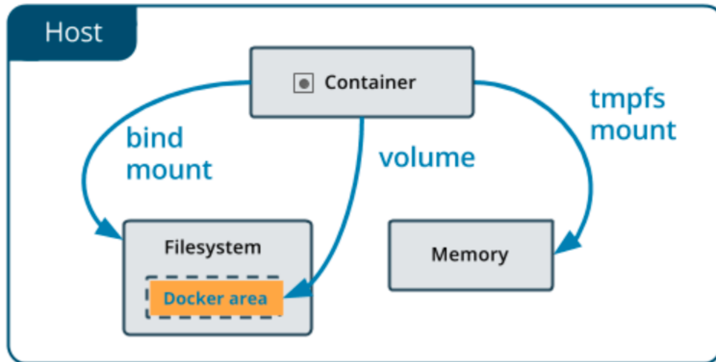
The container's writable layer does not persist after the container is removed (but is persist after restarting a container, i.e. stop ans start again)



Persisting data in docker

Different methods for persisting data of containers

- bind mount: managed by OS (in anywhere)
- volumes: managed by docker (in /var/lib/docker/volumes/)
    - anonymous volumes
    - named volumes



bind mount

```
$ docker run -d \
  -it \
  --name devtest \
  --mount type=bind,source="$(pwd)"/target,target=/app \
  nginx:latest
```

Anonymous Volumes

Anonymous Volumes persist data temporarily; the data will be visible after restarting the container, but not after removing it

- Note: Each anonymouse volume has an ID ('docker volume ls' to see it) through which it is accessible by other containers like named volumes)

```
$ docker run -d \
  --name devtest \
  -v          /app \
  nginx:latest
```

Named Volumes

Named Volumes can persist data after we restart or remove a container and it's accessible by other containers.

```
$ docker volume create my-vol
```

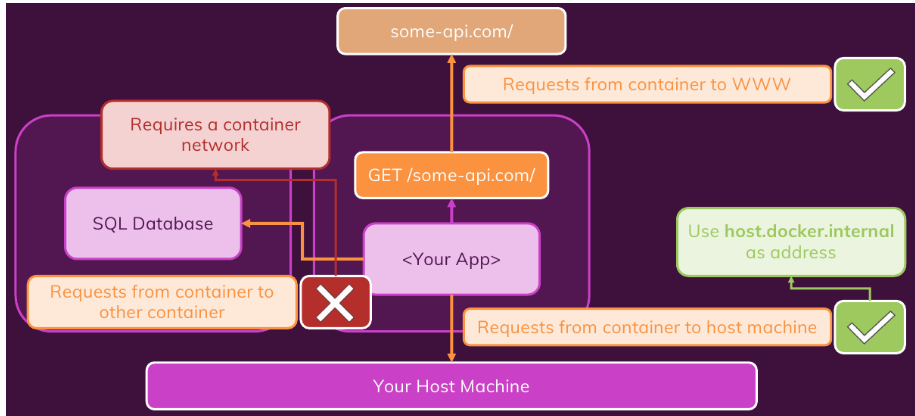
```
$ docker run -d \
  --name devtest \
  -v myvol2:/app \
  nginx:latest
```

Usefull Commands

- docker volume create
- docker volume rm
- docker volume prune
- docker volume ls
- docker volume inspect

Possible communications in a dockerized application

Container to www and container to host are achived without extra work



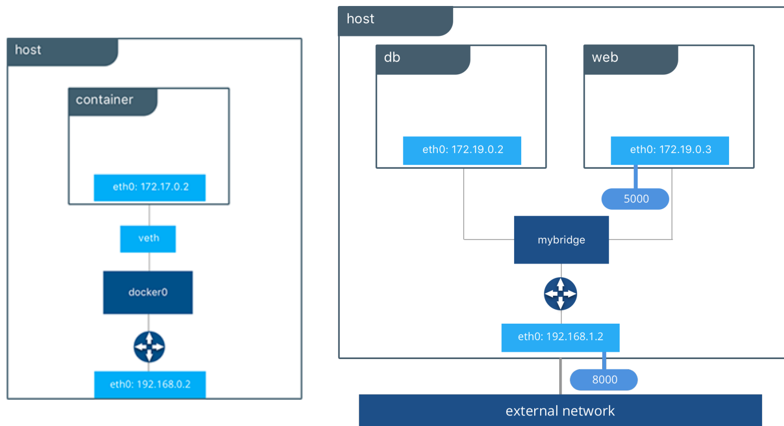
Different methods

How to connect containers together, or connect them to non-Docker workloads?

- **Bridge:** when you need multiple containers to communicate on the same Docker host
- **Host:** when the network stack should not be isolated from the Docker host
- **Overlay**: when you need containers running on different Docker hosts to communicate
- **Macvlan:** when you need your containers to look like physical hosts on your network, each with a unique MAC address
- **Network plugins**

Bridge

The bridge network creates a private internal isolated network to the host so containers on this network can communicate



Bridge

- Bridge mode is the default mode of networking when the –net option is not specified.
- Communication with other containers in the network is open
- Within a Docker network, all containers can communicate with each other and IPs are automatically resolved (through containers' names)
- Communication with services outside the host goes through NAT

User-defined Bridge

```
$ docker network create -d bridge my_bridge
```

```
$ docker network ls

NETWORK ID          NAME                DRIVER
7b369448dccb        bridge              bridge
615d565d498c        my_bridge           bridge
18a2866682b8        none                null
c288470c46f6        host                host
```

```
$ docker run -d --net=my_bridge --name db training/postgres
```

Useful commands

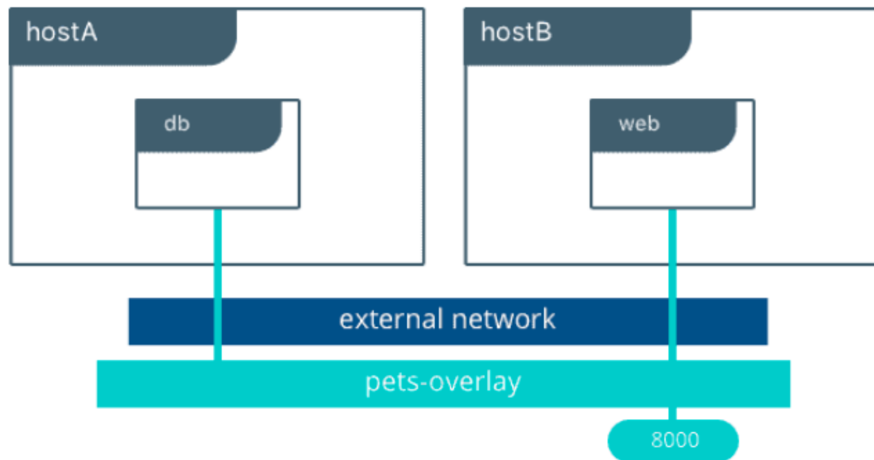
- docker network ls
- docker network inspect bridge
- ip address show dev docker0
- brctl show

Host Networking

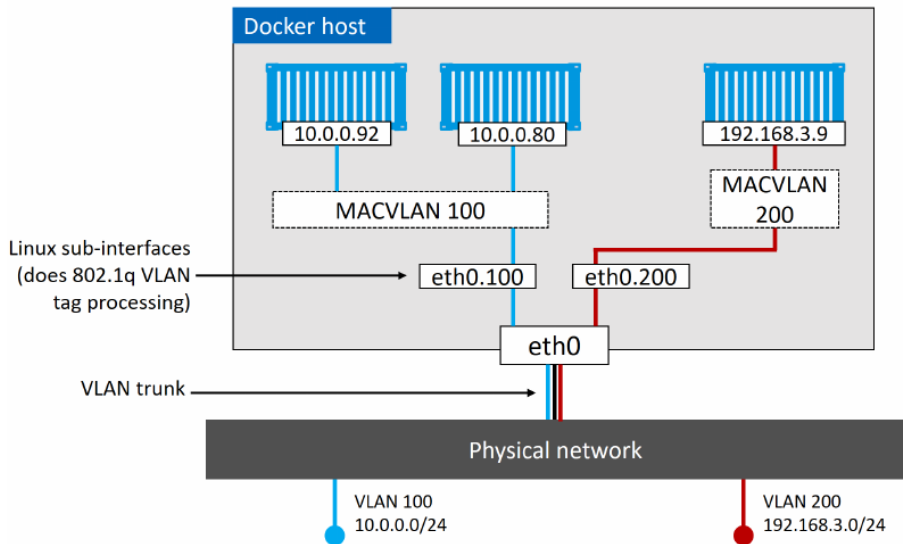
Host mode networking can be useful to optimize performance, and in situations where a container needs to handle a large range of ports

- The container shares the same IP address and the network namespace as that of the host
- Processes running inside this container have the same network capabilities as services running directly on the host

Overlay Networking



Macvlan Networking



Docker Compose

multi-container Docker applications

We can write a shell script for building all the images, creating required networks, exporting required variables, running containers and making the connections, ...

Introducing Docker Compose

Compose is a tool for defining and running multi-container Docker applications in an easy way

- Compose is an external Python binary that we have to install on a host running the Docker Engine.
- We define multi-container (multi-service) apps in a YAML file, pass the YAML file to the docker-compose binary, and Compose deploys it via the Docker Engine API.
  - Manage multiple services
  - Easy volumes and networking
  - Alternative controlling commands

docker-compose.yml

```
version: "3.9"  # optional since v1.27.0
services:
  web:
    build: .
    ports:
      - "8000:5000"
    volumes:
      - .:/code
      - logvolume01:/var/log
    depends_on:
      - redis
  redis:
    image: redis
volumes:
  logvolume01: {}
```

top-level keys in compose.yaml (docker-compose.yml)

- **version**: This defines the version of the Compose file format (basically the API)
  - It is always the first line at the root of the file.
- **services:** Each container represents as a service
- **networks**
- **volumes**
- **configs:** Allow you to store non-sensitive information, such as configuration files, outside a service's image or running containers.
- **secrets:** A secret is a blob of data, such as a password, SSH private key, SSL certificate, or another piece of data that should not be transmitted over a network or stored unencrypted in a Dockerfile or in your application's source code

Some other Options

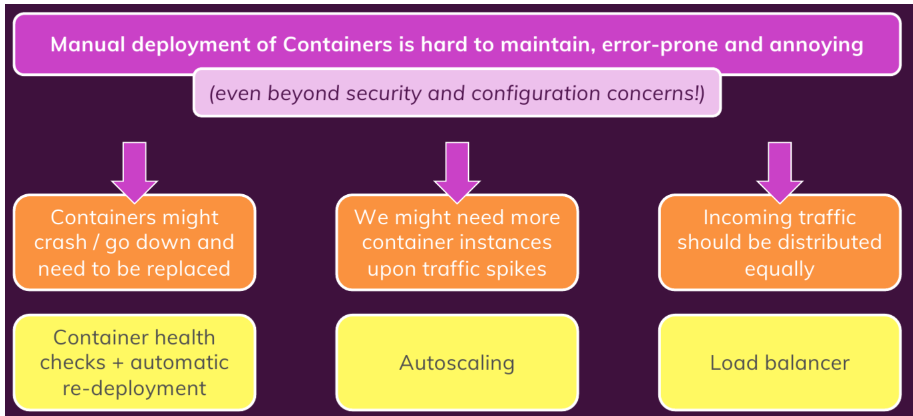
- Ordering deployment of services (through depends on)
- Environment variables for each service (env_file or key/values through environment)
- building an image from a docker file (through build)

docker-compose Commands

- docker-compose –version
- docker-compose up (running the project according to the service specifications in yaml file)
- docker-compose down

Kubernetes

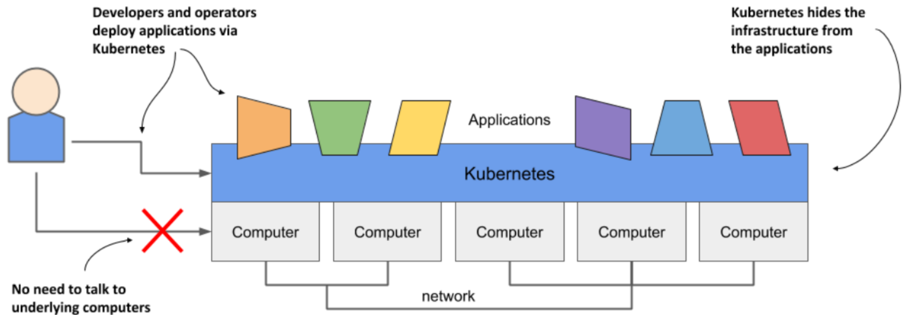
What is the problem?



What is Kubernetes?

Kubernetes is a Greek word meaning the helmsman of a ship. The name is fitting since the entire container ecosystem is following the shipping naming convention.

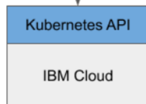
Infrustructure Abstraction

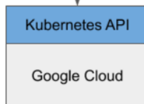


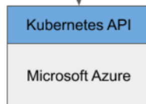
Standardizing the Application

No need to use each provider's proprietary API to deploy and manage applications.

Users can now deploy applications on any cloud provider using a standard set of APIs provided by Kubernetes.



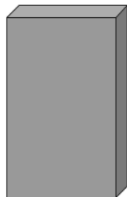




Microservices vs Monolithic

A monolithic application consists of one or a very small number of applications.

Monolithic applications can be managed manually.



Monolithic application

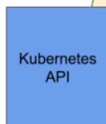
Microservices

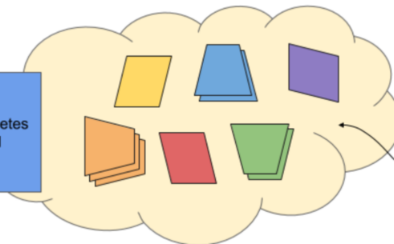
Each microservice is an application in its own right. It must be installed, configured and managed separately.

Microservices require automated management due to their multiplicity and distributed nature.

Uniform Deployment Area

Applications are deployed via the Kubernetes API

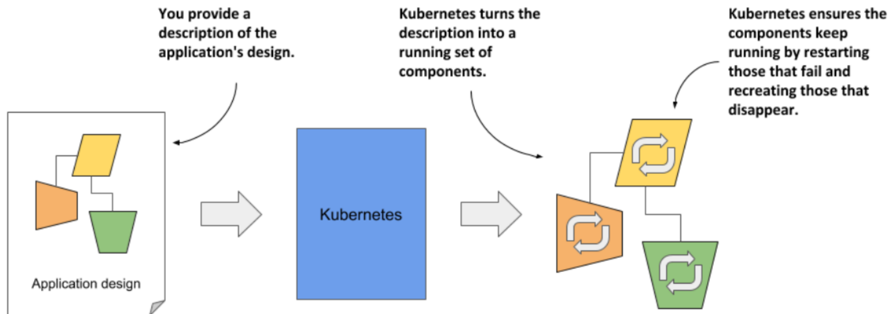




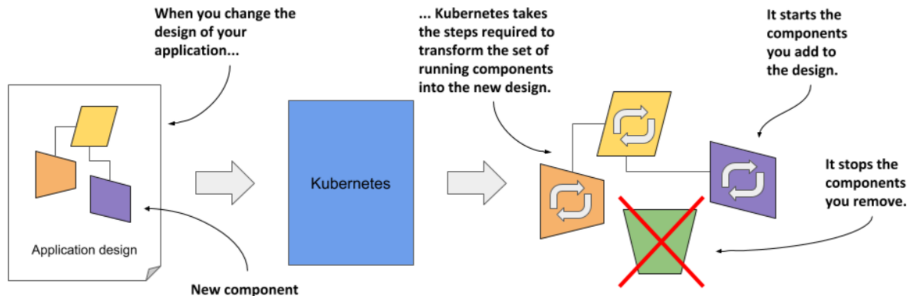
All the worker nodes together form a single place you deploy your applications to.

The applications run across the whole cluster of machines. Individual applications may be relocated whenever necessary.

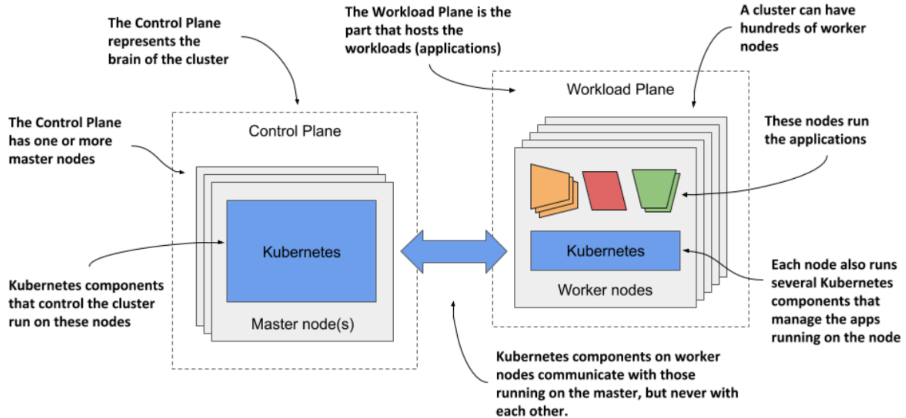
Health Checking and automatic re-deployment



Reconfiging the Running Application



Cluster Management

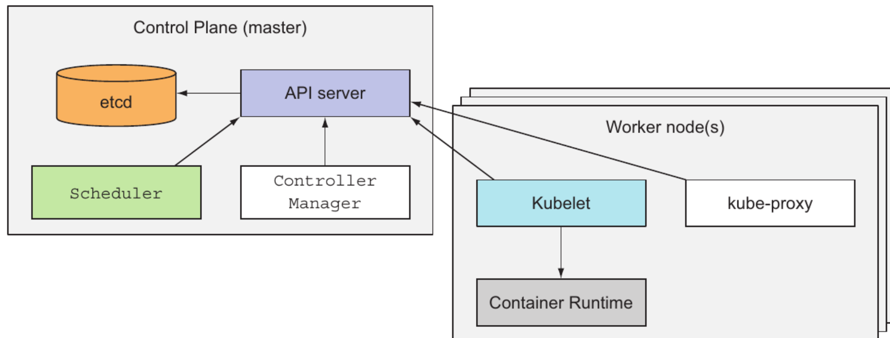


What Kubernetes can do?

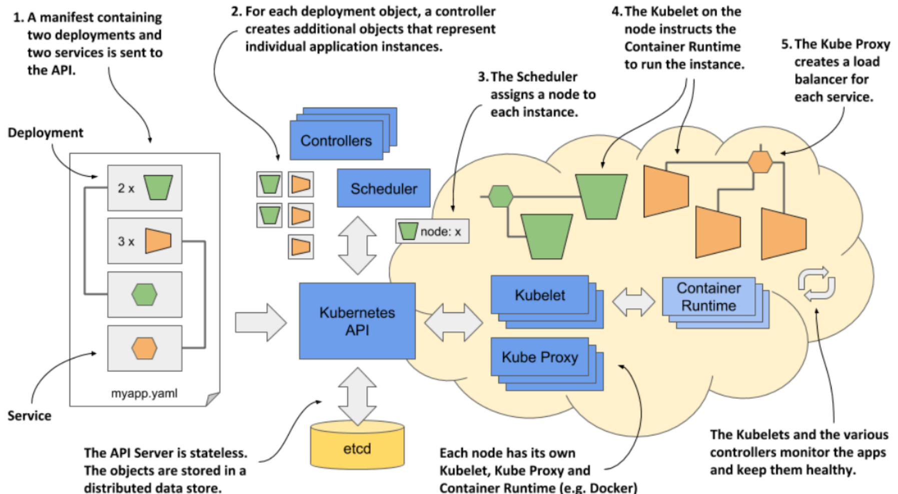
In a production environment, you need to manage the containers that run the applications and ensure that there is no downtime

- Service discovery and load balancing
- Storage orchestration
- Automated rollouts and rollbacks
- Automatic bin packing
- Self-healing
- Secret and configuration management

Kubernetes Architecture



Kubernetes Components

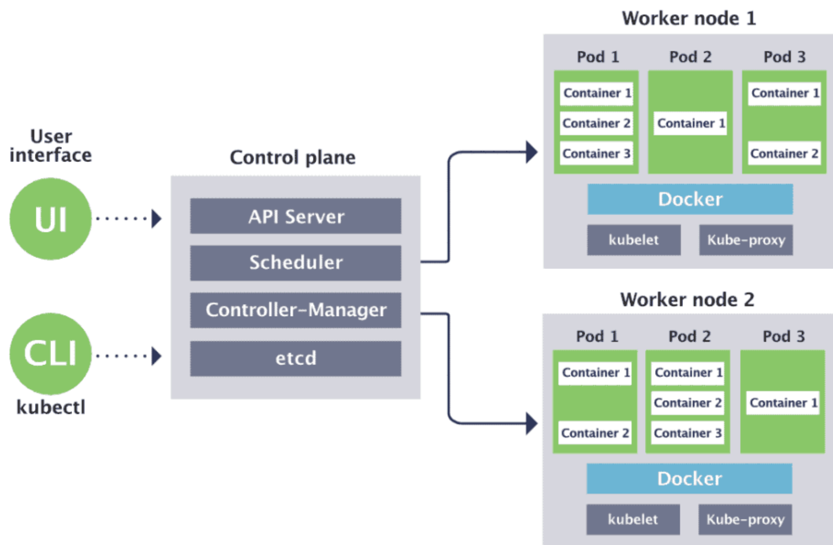


Kubernetes Architecture

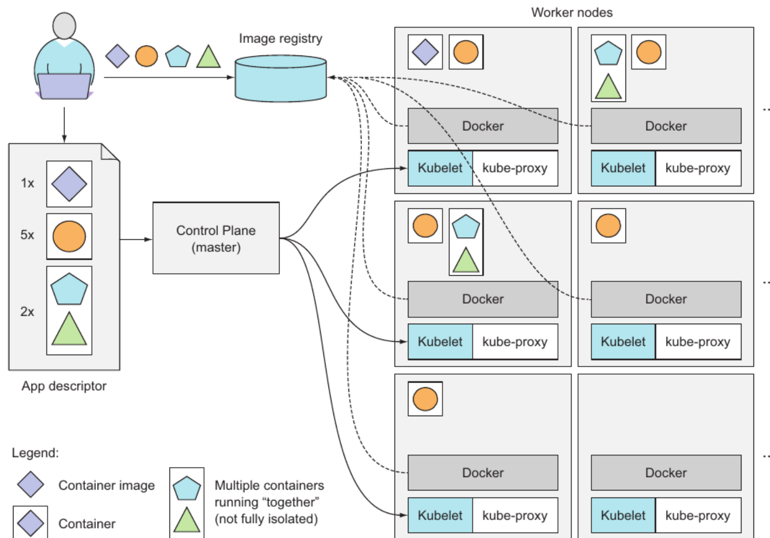
A Kubernetes cluster consists of a set of worker machines (called nodes), controlled by a control plane (called master node(s))

- The worker node(s) host the Pods that are the components of the containerized application workload.
- The control plane manages the worker nodes and the Pods in the cluster.

A Closer look



A Running Application on Kubernetes



Control plane components

The control plane's components make global decisions about the cluster (for example, scheduling), as well as detecting and responding to cluster events.

- API server
- Cluster store (etcd)
- kube-scheduler
- kube-controller-manager
- cloud-controller-manager

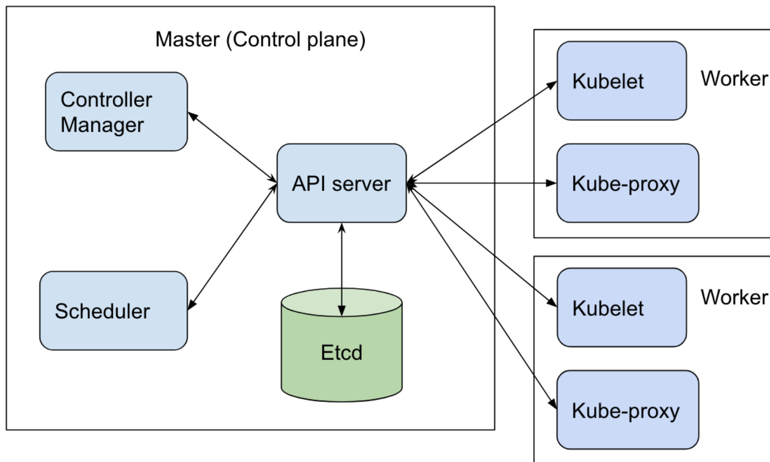
API server

The API server is the front end for the Kubernetes control plane

- the gateway to Kubernetes
- All requests to list, create, modify, or delete any resources in the cluster must go through this service
- It exposes a REST interface that tools such as kubectl use to manage the cluster and applications in the cluster

etcd

Consistent and highly-available key value store used as Kubernetes' backing store for all cluster state.



etcd

- It stores all the information on the topology of the cluster: nodes, pods, controllers, configs, secrets, accounts, and others
  - Everything information we see from running **kubectl** get command
- does not necessarily have to be installed on the same node as the other Kubernetes services
- we need more than one etcd server in a production environment or any environment that needs to be highly available.

kube scheduler

Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on based on:

- individual and collective resource requirements, hardware/software/policy constraints
- affinity and anti-affinity specifications
- quality of service requirements
- data locality

kube controller manager

It watches and controls worker nodes, correct number of Pods and more. It includes some different modules:

- Node controller: Responsible for noticing and responding when nodes go down
- Replication Controllers: monitors replication controllers (replicaset, deployment, job , ...) and the number of alive and healthy pods, then creates Pods if required to run those tasks to completion.
- Endpoints controller: Populates the Endpoints object (that is, joins Services and Pods).
- Service Account and Token controllers: Create default accounts and API access tokens for new namespaces.

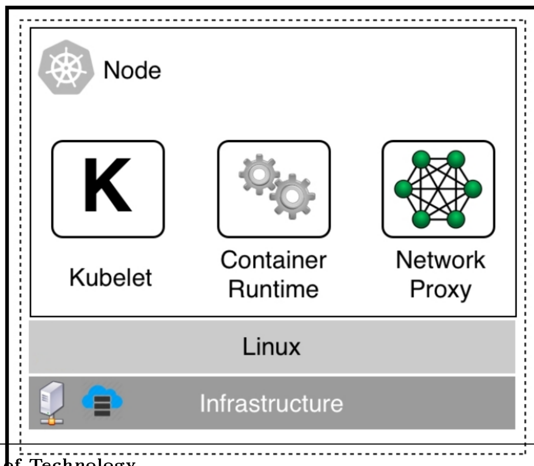
cloud controller manager

Kubernetes control plane component that embeds cloud-specific control logic, So knows how to interact with cloud provider resources

- If you are running Kubernetes on your own premises, or in a learning environment inside your own PC, the cluster does not have a cloud controller manager.
- The cloud controller manager lets you link your cluster into your cloud provider's API, and separates out the components that interact with that cloud platform from components that only interact with your cluster

Node components

- kubelet
- kubeproxy
- container runtime



kubelet

This is responsible for the communication between Master and worker nodes

- Pod Specs are provided to kubelet primarily through the API server.
- This is the first and foremost service which uses pod specifications to make sure all of the containers of the corresponding pods are running and healthy

kubeproxy

It runs as a daemon and is a simple network proxy and load balancer for all application services running on that particular node.

- kube-proxy maintains network rules which allow network communication to your Pods from network sessions inside or outside of your cluster.
  - If the host OS has a network stack that includes packet filtering rules, in that case, kube-proxy utilizes them to perform packet filtering and routing.
  - In the absence of these rules, it relies on itself to accomplish filtering and routing of packets.

container run-time

The container runtime is responsible for managing and running the individual containers of a pod.

- variouse options: containerd since version 1.9 (by default), Docker daemon, Other container runtimes, such as rkt or CRI-O

kubectl

A command line tool for communicating with a Kubernetes cluster's control plane, using the Kubernetes API.

- kubectl run hello-minikube
- kubectl cluster-info
- kubectl get nodes

Cluster

A set of node machines which are running the containerized application (worker nodes) or control other nodes (master nodes)

- Nodes are physical or virtual machines
- Setting up a full-fledged, multi-node Kubernetes cluster isn't a simple task, some tools for creating a kubernetes cluster are:
  - minikube, kubeadm, kind, rancher, Google kubernetes engine, Amazon Elastic Kubernetes Service

Kind

Kind is an open-source tool for running a Kubernetes cluster locally, using Docker containers as cluster nodes.

- kind create cluster –name=iut-cluster –config=iut_cluster.yaml

```
1  # three node (two workers) cluster config
2  kind: Cluster
3  apiVersion: kind.x-k8s.io/v1alpha4
4  nodes:
5  - role: control-plane
6  - role: worker
7  - role: worker
```

Pods

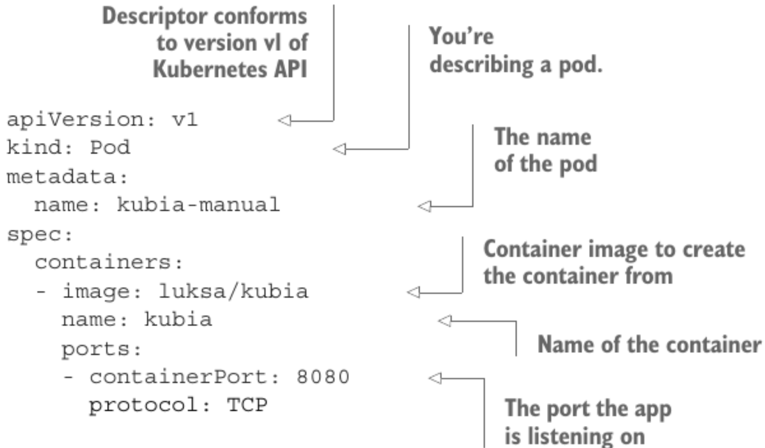
Pods are the smallest deployable units of computing that you can create and manage in Kubernetes.

- a group of one or more containers, with shared storage and network resources, and a specification for how to run the containers
- A Pod's contents are always co-located and co-scheduled, and run in a shared context
- A Pod is similar to a set of containers with shared namespaces and shared filesystem volumes.
- A Pod has a cluster internal IP by default and the containers in a Pod can communicate via localhost

Creating Pods

Directly: create the pod from its yaml file

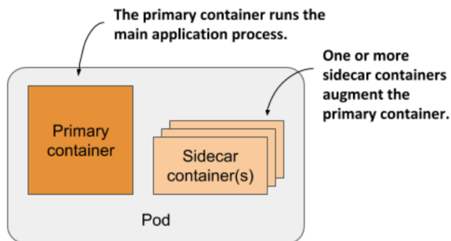
- kubectl create -f kubia-manual.yaml

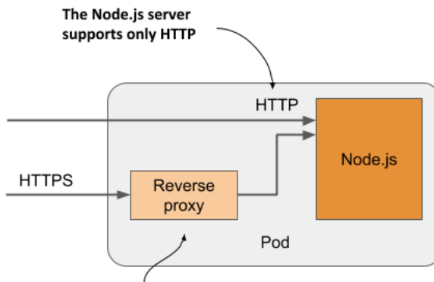


Creating Pods

- Directly (rarely)
- Kubernetes Controllers: Creates pods from templates and ensures the pods are always kept running.
    - ReplicaSet, Deployment, Job, DaemonSet, StatefulSet

Different containers in a single pod?



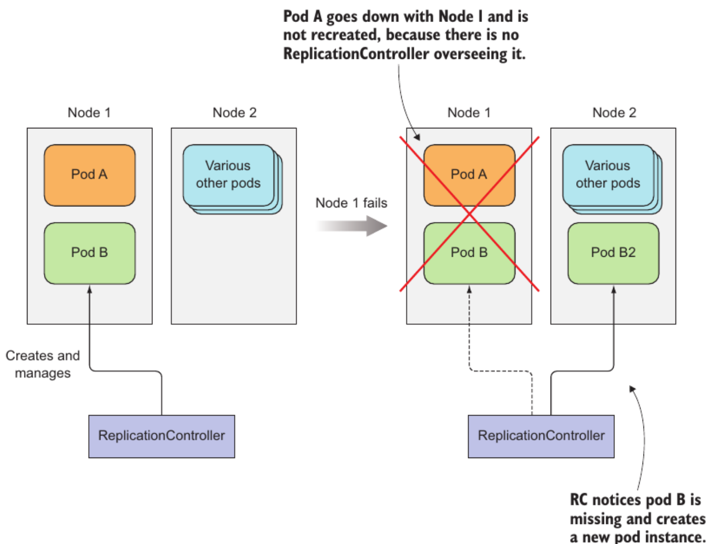


How to decide wether to split containers into multiple pods?

Always place containers in separate pods unless a specific reason requires them to be part of the same pod:

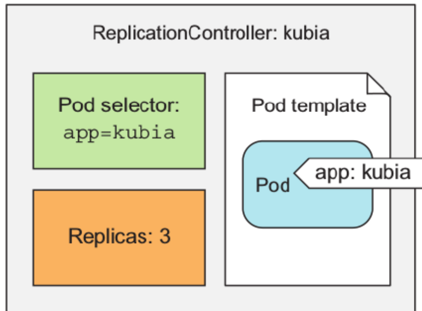
- Do these containers have to run on the same host?
- Do I want to manage them as a single unit?
- Do they form a unified whole instead of being independent components?
- Do they have to be scaled together?
- Can a single node meet their combined resource needs?

Controllers Operation



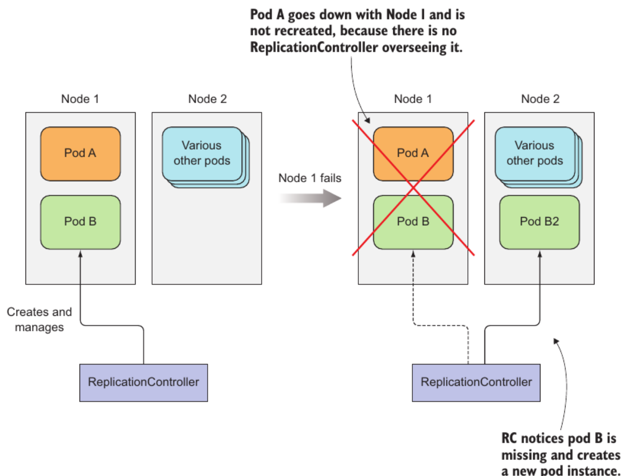
Three key parts of Controllers

- **Label selector**: It determines what pods are in the controller's scope
- **replica count**: It specifies the desired number of pods that should be running
- **Pod template**: It is used when creating new pod replicas



ReplicaSet

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



ReplicaSet yaml file

Create an object with type replicaset and set the pod specification from metadata below the template section

```yaml
apiVersion: apps/v1
kind: ReplicaSet
metadata:
  name: iut-replicaset
  labels:
    goal: learning
    type: server
spec:
  template:
    metadata:
      name: iut-pod
      labels:
        goal: learning
        type: web-server
    spec:
      containers:
        - name: node-app
          image: nginx
          ports:
            - containerPort: 80
  replicas: 3
  selector:
    matchLabels:
      type: web-server
```

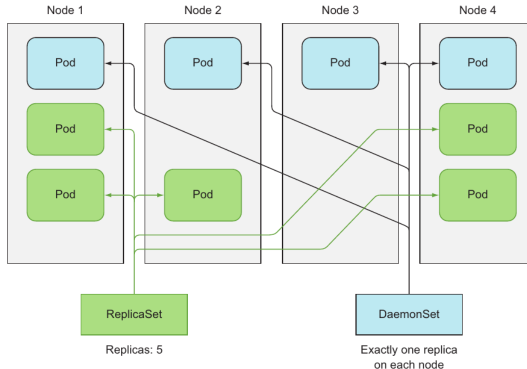
Usefull commands

- kubectl create -f replicaset-definition.yaml
- kubectl apply -f replicaset-definition.yaml
- kubectl replace -f replicaset-definition.yaml
- kubectl get replicaset
- kubectl delete replicaset iut-replicaset
- kubectl scale replicas=6 -f replicaset-definition.yaml

DaemonSet

A DaemonSet makes sure it creates as many pods as there are nodes and deploys each one on its own node

- DaemonSets run only a single pod replica on each node, whereas ReplicaSets scatter them around the whole cluster randomly

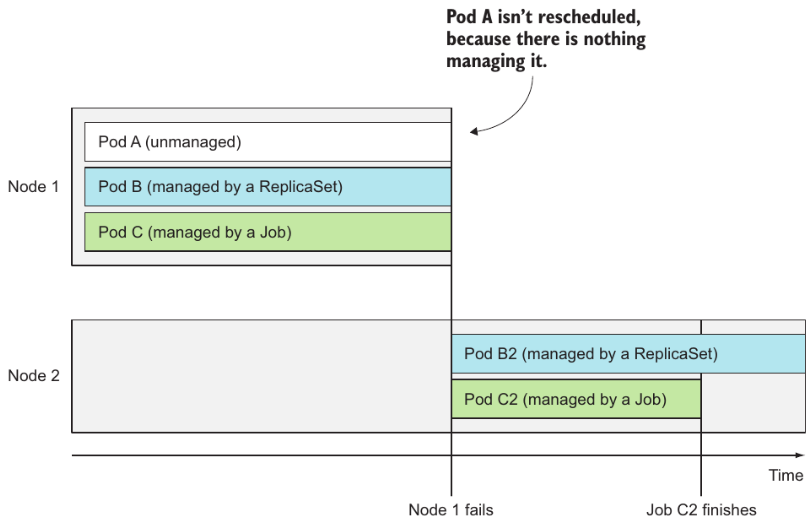


Job

You'll have cases where you only want to run a task that terminates after completing its work. Job allows you to run a pod whose container isn't restarted when the process running inside finishes successfully

- In the event of a node failure, the pods on that node that are managed by a Job will be rescheduled to other nodes the way ReplicaSet pods are.
- In the event of a failure of the process itself (when the process returns an error exit code), the Job can be configured to either restart the container or not.

Job



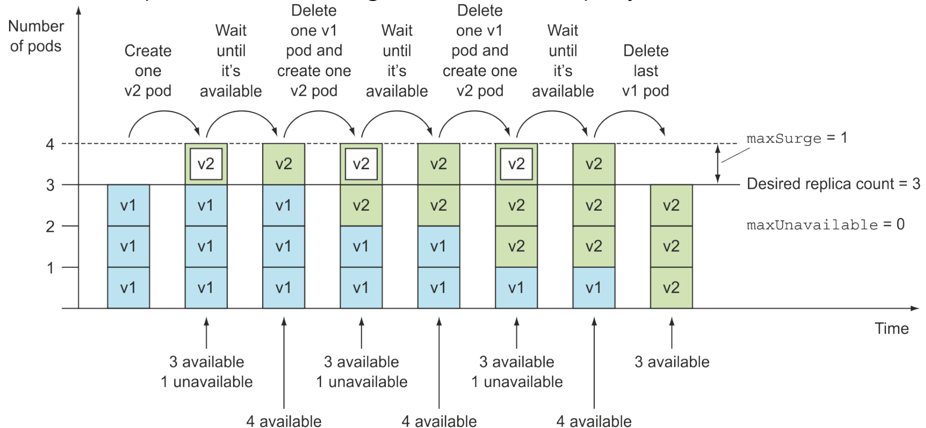
Deployement

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

- Create a Deployment to rollout a ReplicaSet
- Rollback to an earlier Deployment revision
- Scale up the Deployment to facilitate more load

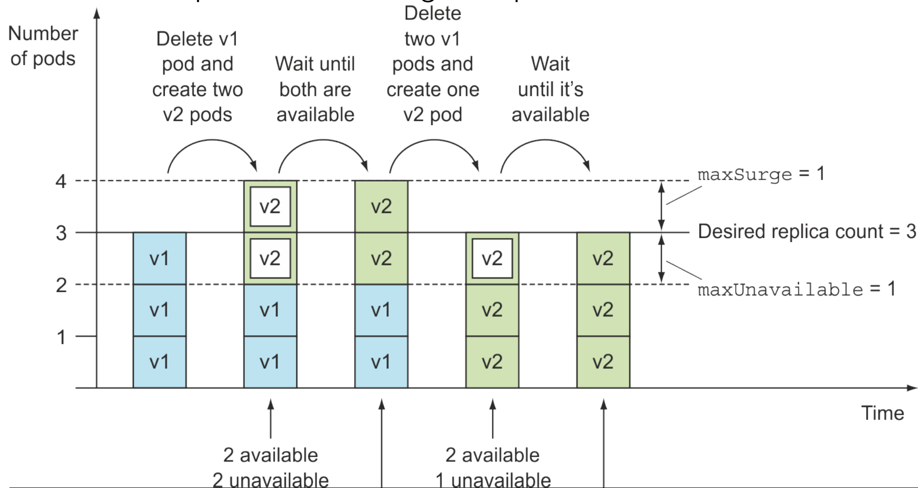
Deployment maxSurge

Determines how many pod instances you allow to exist above the desired replica count configured on the Deploymen



Deployment maxUnavailable

Determines how many pod instances can be unavailable relative to the desired replica count during the update

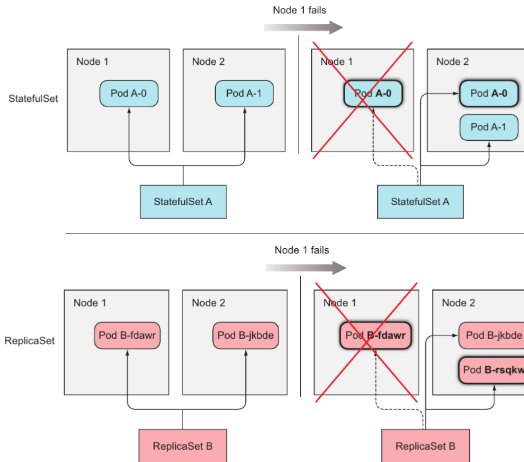


StatefulSets

How does one run multiple replicas of a pod and have each pod use its own storage volume? ReplicaSets create exact copies (replicas) of a pod; therefore you can't use them for these types of pods. What can you use?

- The Answer is using **StatefulSets**
- A StatefulSet replaces a lost pod with a new one with the same identity, whereas a ReplicaSet replaces it with a completely new unrelated pod

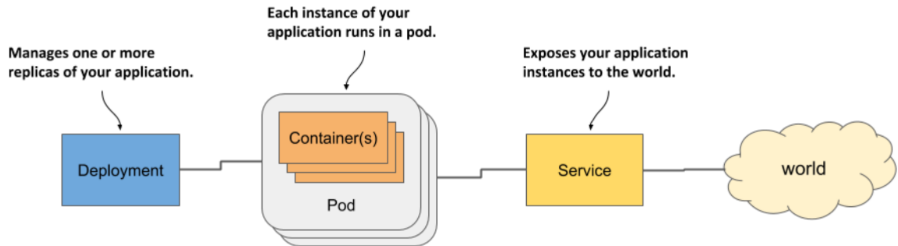
StatefulSets vs Replicaset



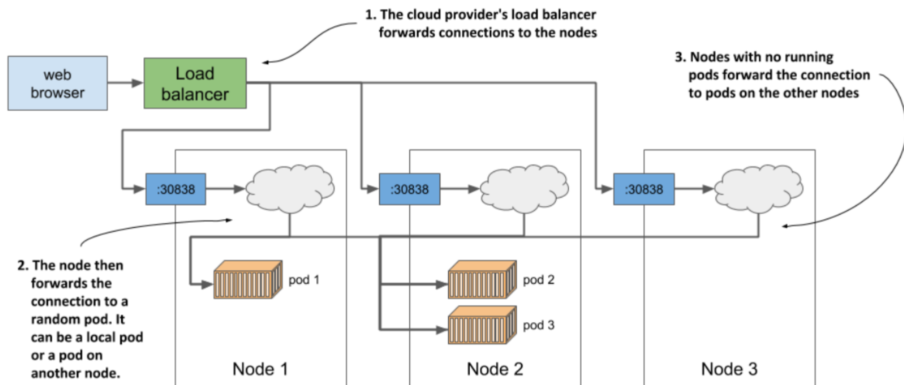
Service

When your application is running, so the next question to answer is how to access it.

- each pod gets its own IP address, but this address is internal to the cluster and not accessible from the outside
- To make the pod accessible externally, you'll expose it by creating a Service object.
- Several types of Service objects exist such as LoadBalancer



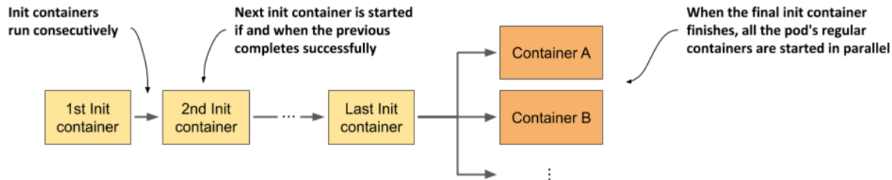
Load Balancer Service



Init Containers

Init containers are a list of containers to run when the pod starts and before the pod's normal containers are started.

- When a pod contains more than one container, all the containers are started in parallel
- Init containers run one after the other and must all finish successfully before the main containers of the pod are started



Init Containers Yaml File

```yaml
apiVersion: v1
kind: Pod
metadata:
  name: kubia-init
spec:
  initContainers:
  - name: init-demo
    image: luksa/init-demo:1.0
  - name: network-check
    image: luksa/network-connectivity-checker:1.0
  containers:
  - name: kubia
    image: luksa/kubia:1.0
    ports:
    - name: http
      containerPort: 8080
```

Lifecyle hooks

These lifecycle hooks are specified per container, unlike init containers, which apply to the whole pod. As their names suggest, they're executed when the container starts and before it stops.

- Post-start hooks: A post-start hook is executed immediately after the container's main process is started. The hook is run in parallel with the main process.
- Pre-start hooks: A pre-stop hook is executed immediately before a container is terminated

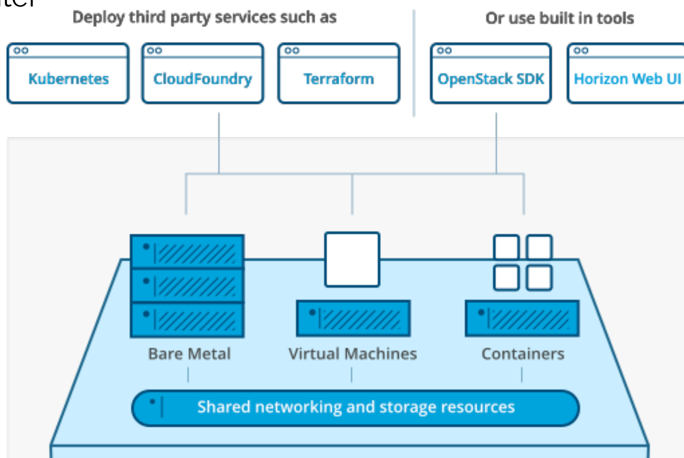
Web Dashboard

A graphical web user interface that its functionality may lag significantly behind kubectl, which is the primary tool for interacting with Kubernetes.

OpenStack

Introduction

OpenStack is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter



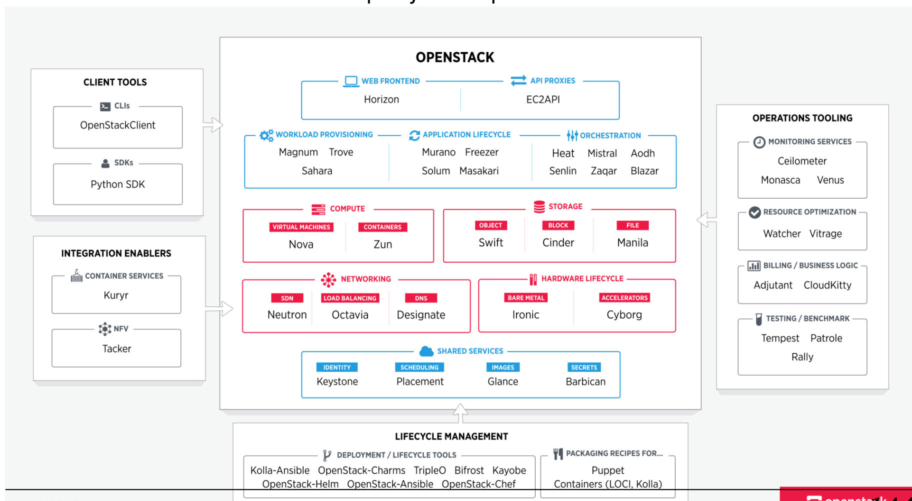
Introduction

OpenStack is an open source software project originally created by NASA and Rackspace (now RedHat, IBM, and HP as well as companies which are dedicated entirely to OpenStack such as Mirantis, and CloudBase)

- It is written in the Python programming language and is usually deployed on the Linux operating system
- Management and provisioning methods:
    - through APIs: SDK
    - through dashboard: giving administrators control while empowering their users to provision resources through a web interface

Architecture

OpenStack is a modular system and is broken up into services (plug and play components).



Compute

- Nova: It provides a way to provision compute instances (aka virtual servers).
- Zun: It provides an OpenStack API for launching and managing containers backed by different container technologies

Compute Service: Nova

- Nova supports creating virtual machines, baremetal servers (through the use of ironic), and has limited support for system containers.
- Nova runs as a set of daemons on top of existing Linux servers to provide that service.
- we refer to provisioned compute nodes as instances and not virtual machines

Availability Zones

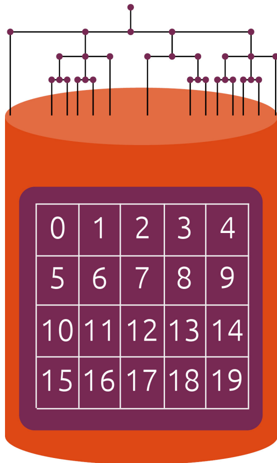
Availability Zones are an end-user visible logical abstraction for partitioning a cloud without knowing the physical infrastructure

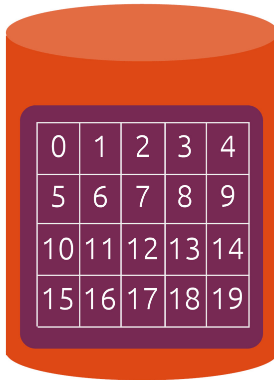
- Availability zones can be used to partition a cloud on arbitrary factors, such as location (country, datacenter, rack), network layout and/or power source
- Availability zones can only be created and configured by an admin but they can be used by an end-user when creating an instance

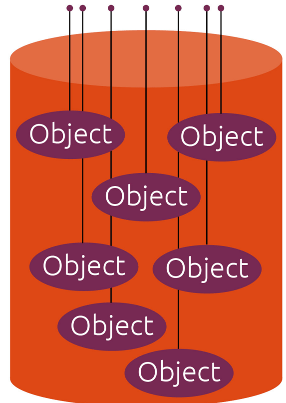
Persistent Storages

- Object storage (SWIFT): It is a highly available, distributed, eventually consistent object store based on the S3 service available in the Amazon Web Service environment.
- Block storage (CINDER): It virtualizes the management of block storage devices and provides end users with a self service API to request and consume those resources without requiring any knowledge of where their storage is actually deployed or on what type of device
  - The life cycle of Cinder volumes is maintained independent of compute instances
  - volumes may be attached or detached to one or more compute instances to provide a backing store for filesystem-based storage.

Block vs File vs Object Storage







Networking

- Networking (Neutron): Neutron is an SDN networking project that provides an API for creating ports, subnets, networks, and routers
- Load Balancer (Octavia)
- DNS Service (Designate)

Shared Services

- Keystone (Idendity Service)
- Placement (Placement Service): The Compute scheduler service uses the Placement service to filter the set of candidate Compute nodes based on specific attributes
- Glance: Image Service

Horizon

Horizon is the canonical implementation of OpenStack's dashboard, which is extensible and provides a web based user interface to OpenStack services.

Heat

A Heat template describes the infrastructure for a cloud application in a text file that is readable and writable by humans, and can be checked into version control, diffed.

- Infrastructure resources that can be described include: servers, floating ips, volumes, security groups, users, etc.

Flavor

The flavor describes the characteristics of the instantiated image-it normally represents a number of cores of compute with a given amount of memory and storage.

- all compute in OpenStack is the instantiation of a Glance image with a specified hardware template, the flavor