__/__/____

- Linux Foundation: Hub for open-source tech. projects

- CNCF: Cloud Native Computing Foundation
  ⇓
  under the Linux Foundation umbrella.

  vendor neutral home for many of the fastest growing projects on Github including Kubernetes
  Prometheus → open source monitoring system and was the 2nd project accepted into CNCF after Kubernetes.
  Envoy → High-performance proxy developed in C++ to mediate all inbound and outbound traffic for all services in the service mesh.

  * code hosting platform for version control and collaboration
  * commonly used to host open-source projects

- CLOUD NATIVE ARCHITECTURE:

Ⓐ Cloud Native Arch. Fundamentals:

HOW ⟺ cultural + technological + architectural design patterns → [Cloud Native Arch.] → WHAT (Objective) ⟺ optimize S/W for cost effectency reliability and faster time to market

Ⓑ Cloud-native tech. → [Orgⁿ] → Build and run scalable apps in modern dynamic environments such as

(Micro-service arch. + CICD + Containers + Cloud)

↓

Loosely coupled systems (resilient manageable observable)

↓

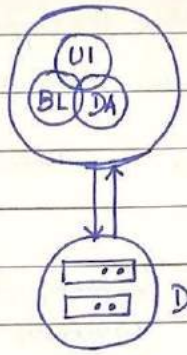
Robust automⁿ → Frequent changes predictably with min effort

APIs + ↓ S/W intermediary that allows 02 apps talk to each other.

e.g. Waiter → API (tells Kitchen what the customer wants and delevers after prepar^{n} of food)

__/__/____

- Monolithic Arch:



UI - User interface
BL - Biz. logic
DA - Data Access

* Traditional way of building appⁿs
* a monolithic applⁿ is built as a single and indivisible unit.
* It is a single logical executable.
* S/W built on this model operates with one single base of code. Whenever stakeholders want to make updates or changes, they access the same set of code.
* Monolith —meaning→ Object made from single large piece of material
* monolithic arch. —also known→ multi-tier arch.
* Orgⁿs:

.... ▶ .... Web Browser (Client)
⇓
could HTML, CSS and Javascript

LEGACY

Data Exchange Format : Enterprise Data Bus
Enterprise Data Interchange
Enterprise Data Exchange

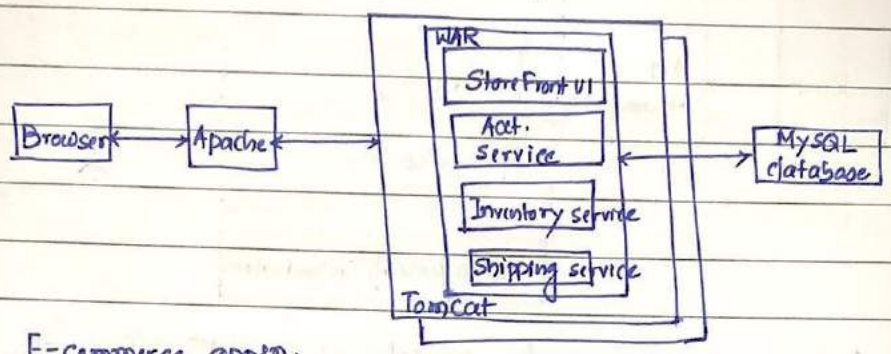
MODERN

→ Web Browser ..▶.. Mobile ..▶.. SmartPhones
→ Mobile applⁿ taking more priority than web applⁿ
⇓
orgⁿs ..▶.. need to expose data using APIs. because legacy data exchange formats are not compatible with mobile appⁿ.

→ Mobile applⁿ is nearly useless without internet connectivity and backend services.

__/__/____

- Traditional Web Appln Arch:



E-commerce appln:
takes order from customers
verifies inventory and available credit
and ships them.

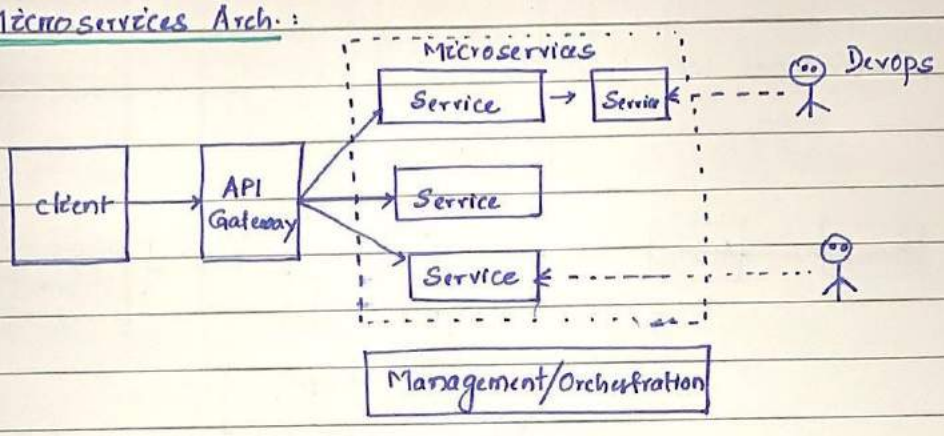
developed as Java-web appln → consists of a single WAR file → Runs on a web container such as Tomcat

- Advantages:
  * easier debugging and testing
  * simple to deploy
  * simple to develop

- Challenges:
  * Understanding - When a monolithic appln scales up, it becomes too complicated to understand.
  * Hard to implement changes in a large and complex appln with highly tight coupling
  * code becomes complicated when the appln grows.
  * Scalability: cannot scale components independently, only the whole appln.
  * Extremely difficult to change tech. or language or framework as everything is tightly coupled and depends upon one another
  * Reliability: Bug in any module can potentially bring down the entire appln.

__/__/____

- Microservices Arch.:



* A micro-services arch. consists of a coll^{n} of small, autonomous services.

Service → self-contained and should implement a single biz. capability within a bounded context.

→ independent and loosely coupled.

→ services communicate with each other by using well-defined APIs.

* API Gateway: Entry point for clients

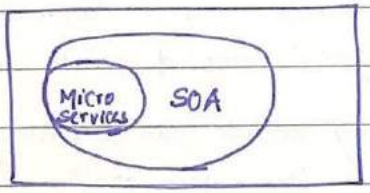
Clients call the API gateway, which forwards the call to the appropriate services on the back end.

decouples clients from services.

API layer also enables the micro-services to communicate with each other over HTTP, gRPC and TCP/UDP.

* The comm^{n} between microservices is a stateless comm^{n} where each pair of request and response is independent.

* Micro-services is an implementation of SOA. (service-oriented arch)



S/W components are exposed to the outer world for usage in the form of services.

SOA → Enterprise service exposure

Microservices → Appl^{n} scope

⭐ Micro services-based arch. is aligned with event-driven arch. and SOA principles.

⭐ One of the greatest benefits of micro-services is scalability.

- REST: Representational State Transfer
  ↓
  framework for developing APIs
  REST APIs work by feeding HTTP requests and returning responses in JSON (JavaScript Object Notation) format.

* With HTTP methods (such as GET, POST, PUT and DELETE), clients can access and manipulate appl^{n} resources by using a URL.

- Principles of Microservice Design:
  1. Reuse
  2. Loose coupling
  3. Autonomy
  4. Fault tolerance
  5. Composability
  6. Discoverability

- Advantages:
  1) Agility: easier to manage bug fixes and feature releases
     → update a service without redeploying the entire appl^{n}
     → rollback an update if something goes wrong
  2) Small code base
  3) Mix of technologies
  4) Fault isolation: If an individual micro-service becomes unavailable, it won't disrupt the entire appl^{n}.
  5) Scalability: Services can be scaled independently, letting you scale out subsystems that require more resources, without scaling out the entire appl^{n}.

- Drawbacks:
  1) Additional complexity of creating a distributed system
     → developers must implement the inter-service comm^{n} and deal with partial failure
     → implementing requests that spans multiple services is more difficult.

Transition from a legacy monolith to modern micro-services architecture:

i) Running monoliths as micro-services (X)

ii) Big-bang approach (Refactoring of the monolith, postponing the devp. of new features) (X)

iii) Incremental refactoring (New features – microservices, communicate with monolith through APIs
→ testing the interaction between services is difficult.

2) Deployment complexity: operational complexity of deploying and managing a sys. comprising of many services

3) Increased memory consumption:

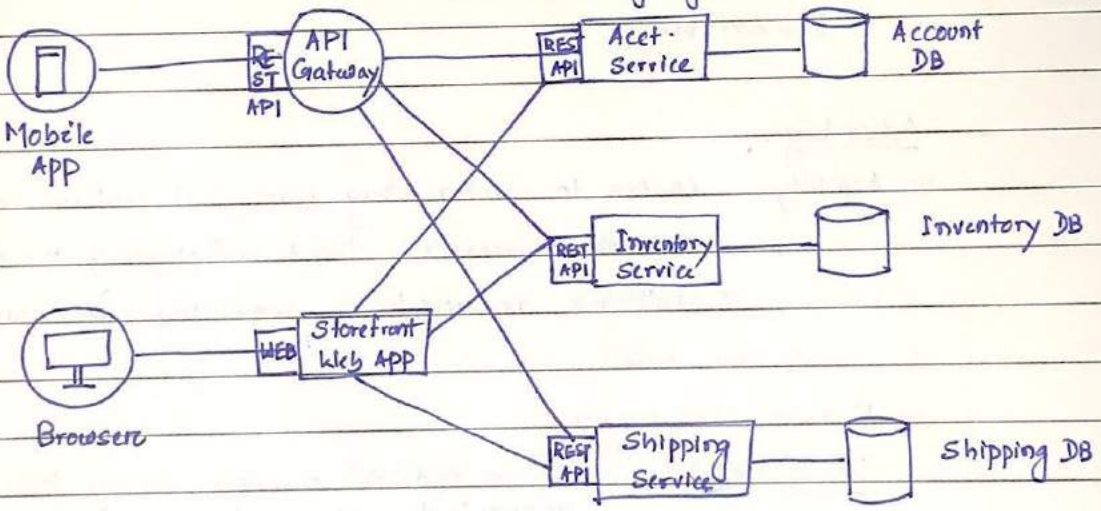
Monolithic appln → N instances

Microservice appln → NXM service instances

↓

overhead of M times as many JVM runtimes
(if each service runs in its own JVM)

4) Lack of governance: appln becomes hard to maintain with so many different languages and frameworks.



	Monolithic Arch.	Micro-services Arch.
Comm" with the appln	tightly ~~loosely~~ coupled ~~services~~ (Changes in one module of code affect the other) Internal procedures (function calls)	loosely ~~tightly~~ coupled services (Changes made in one doesn't affect the other) REST API calls
	~~De~~centralized data mgmt. (entire app. may use one or more DBs)	Decentralized data mgmt. (Each microservice may use its own DB) ↓ federated

__/__/____

Entire appln must be scaled.

Each micro-service can be scaled independently.

Fault isolation - difficult
↳ If any specific feature is not working, the complete sys. goes down.

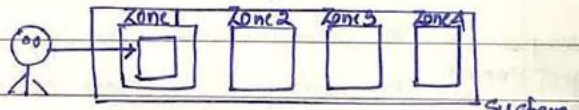
Fault isolation - easy

- Characteristics of Cloud Native Architecture:

1. High level of automation → CI/CD pipeline → helps in easier DR
2. Self healing → regular automated health check of your appln ⇒ automatic restart of services
3. Scalable → scaling your appln describes the process of handling more load while producing the same cost. exp. One way of scaling can be starting multiple instances of the same appln and distributing the load across them. Automating this process based on appln metrics like CPU or memory can ensure availability and perf. of the services.
4. Cost-efficient
5. Easy to maintain → (Microservices)
6. Secure by default

Scaling down your appln and usage based pricing models (CSPs) can save cost if traffic is low.

Kubernetes: optimize of infrastructure usage
↓
efficient and denser placing of applns



Users who are inside a zone can access every system inside.

- Zero-trust Computing:

* known as zero trust security model or zero trust arch. (ZTA) or zero trust network arch. (ZTNA)

* Approach to the design and impln of IT systems where the main concept is 'never trust, always verify'.
↓
devices shouldn't be trusted by default, even if they are connected to LAN or managed network and previously verified.

__/__/____

* Zero trust is designed to protect modern digital envs by leveraging network segn, preventing lateral movement, providing Layer 7 threat prevention and simplifying granular user-access control.

* Trust is a vulnerability.

* security framework requiring all users, whether in or outside the org's network to be authenticated, authorized and continuously validated for security config and posture before being granted or keeping access to appl^ns and data.

* Also known as perimeterless security.

Twelve-Factor App:

* Modern Era: S/W delivered as a service (webapps or SaaS)

* Twelve-Factor App: ~~methodology for building software as a service.~~ guideline for developing cloud-native appl^ns

1) Codebase – one codebase tracked in revision control (CI/CD)
2) Dependencies – explicitly declare and isolate dependencies
   Appl^n packages → managed through sbt, maven
3) Config – store configs in an env. (no hard coding of config^n values)
4) Backing services – treat backing services as attached resources
   (database, message broker or any ext. sys that the app communicates)
5) Build, Release, Run – strictly separate build and run stages
6) Processes – Execute the app as one or more stateless processes
   (Store the data in store/DB instead of in-memory)
7) Port Binding – export services via port binding
8) Concurrency – scale out via the process model
9) Disposability – maximize robustness with fast startup and graceful shutdown
10) Dev/Prod parity – keep the env's as similar as possible
11) Logs – treat logs as event streams
12) Admin Processes – run admin tasks as one-off processes

__/__/____

- Concurrency: Scaling the appⁿ
  ⇓
  running the appⁿ as multiple processes/instances instead of running as one large sys.

(scaling up) vertical scaling → add additional H/W to the sys. [e.g. CPU, RAM]
(scaling out) horizontal scaling → add additional instances of the applⁿ
(adoption of containers) ⇒ adding more machines to your pool of resources.

- System Disposability: sys. should not be impacted when new instances are added or taken from the existing instances as per need.

- Logs: observability: achieved → APM tools (ELK)
  Log aggⁿ tools (Splunk)

- Auto Scaling:

* Auto Scaling: - cloud computing tech. for dynamically allocating computational resources. (compute, memory or networking resources)
  - Instance: single server or machine
  - Advantages:
    cost
    security
    availability
    Resilience

* Load Balancer: distribute the incoming traffic across multiple targets (VM instances)
  ↓
  scales your app
  supports heavy traffic
  detect and automatically remove unhealthy VM instances
  route traffic to the closest VM.

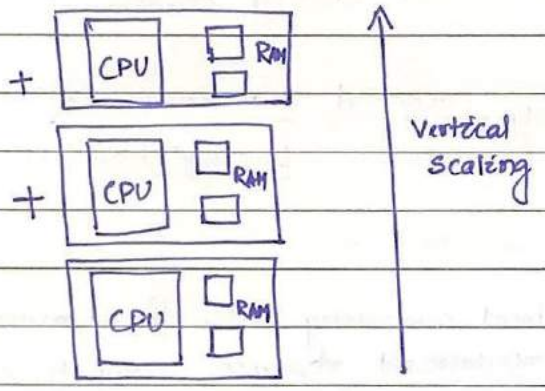
* Horizontal Auto Scaling: (Process of spawning new compute resources)
  ⇓
  new copies of your appⁿ process
  VMs
  new racks of servers
  new servers or machines

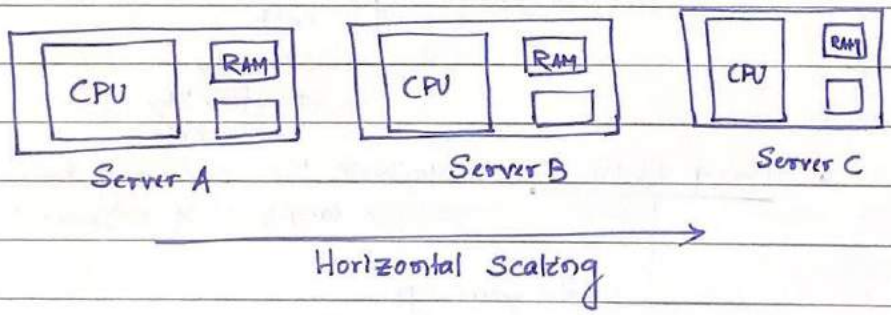
__/__/____

* Vertical Auto Scaling: describes the change in size of (CPU & memory) underlying H/W which only works within hardware limits for BareMetal servers, VMs

The H/W can be scaled up e.g. by adding more RAM but only until all RAM slots are occupied.

* Phy. server dedicated to a single tenant (single customer)
* known as managed dedicated servers





- Serverless:

* Serverless computing → arch. where code execution is managed by a cloud provider instead of the traditional method of developing appl^ns and deploying them on servers.

* Developers don't have to worry about managing, provisioning and maintaining servers when deploying code.

__/__/____

* Serverless relies on functions or more specifically functions-as-a-service (Faas) where developers break down their appl^n's into small, stateless chunks, meaning that they can execute without any context regarding the underlying server.

Faas offerings:

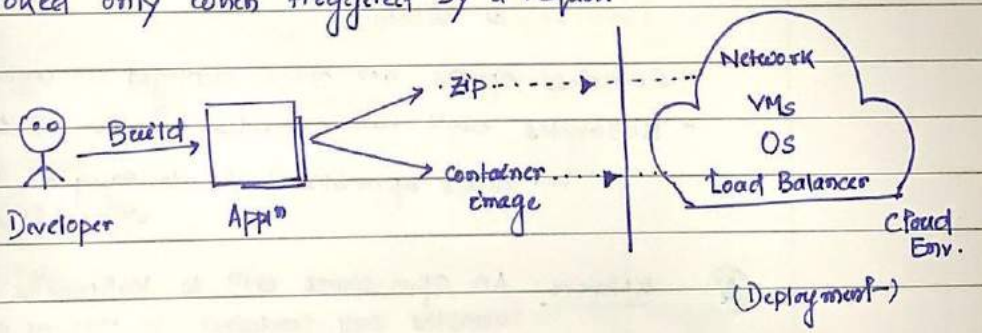
(actually a subset of serverless)

- AWS Lambda : Allows developers to run code without provisioning or managing servers. AWS charges you for the compute power along with the storage you use.
- IBM Openwhisk
- Cloud Run (Google)
- MS Azure Functions

* Faas —(also known as)→ Caas (Compute as a Service)

* Developers —(don't have to know)→ anything about the H/w or os where the code runs on. It's taken care by the CSP.

* Serverless functions are event-driven, meaning the code is invoked only when triggered by a request.



(Deployment)

* Advantages:
  1) Lowers costs 2) simplified scalability 3) simplified backend code
  4) quick turnaround

* Drawbacks: 1) Stable or predictable workloads
  2) Monitoring and debugging
  3) Vendor lock-in

__/__/____

* Serverless offerings from public cloud providers are usually metered on demand through an event-driven execution model.

event-driven arch. is a s/w arch. and model for appln design.

Event: Any significant occurrence or change in state for system H/w or S/w.

events can generate from a user like a mouse click or keystroke external source.

* Kubernetes: Open-source container orchestration platform that automates schedules the deployment management and scaling of containers.

- Serverless applns are often deployed in containers.
- Kubernetes can't run serverless applns on its own.
  - ↳ needs specialized s/w to integrate Kubernetes with CSP's serverless platform

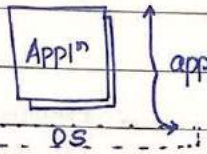
- KNative: An Open-source extn to Kubernetes that enables any container to run as a serverless workload on any cloud platform that run Kubernetes.

- CloudEvents: specifn for describing event data in a common way. to provide interoperability across services platforms and systems
  - ↳ Goal: Standardize how event publishers describes their events.
  - ↓ One-type of event driven framen

__/__/____

Events ⟶ bases for scaling serverless workloads or triggering corresponding funct^{n}s.

- Open Standards:



Common problem: Building and distribution of S/W packages.

appl^{n}s have a lot of req. and dependencies for the underlying OS and the appl^{n} run time.

solution: Appl^{n} → package as → container image

OCI: (open container initiative) → open governance structure for the express purpose of creating industry standards around container formats & runtime

02 specif^{n}s

- Run Time ← (config^{n}, execution env. and life-cycle of containers)
  - how to run a "filesystem bundle" that is unpacked on disk
- Image (Build and package container images)

OCI impl^{n} ⟶ download an OCI image
↓
unpack that image into an OCI Run-time filesys. bundle
↓
OCI Run-time bundle to be run by OCI Runtime

__/__/____

OCI Distribution spec → (API Protocol) defining distribution of content and container images.

CNI (Container Network Interface) → implement networking for containers
⇓
(CNCF proj.)

CRI (Container RunTime Interface) → implement container runtimes in container orch^n systems

CBI (Container storage Interface) → implement storage in container orch^n systems

SMI (service mesh interface) → implement service meshes in container orch^n systems

Cloud Native Roles & SRE:

Cloud Architect: responsible for adopt^n of cloud

DevOps Engineer: CI/CD

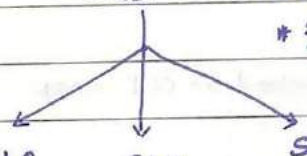
security Engineer:

DevSecOps Engineer: Build bridges between traditional devp. and security teams

Data Engineer: ETL (Developers + Admin)

Full-stack Developer: all-rounder (Front end
Back end
Infra. opr^{n}s)

SRE: Create and maintain s/w that is reliable and scalable.



* SRE is just one of the ways to implement the DevOps culture.

☆ class SRE implements DevOps {}

SLO ↓ specify a target level for the reliability of the service

SLI ↳ a carefully defined quantitative measure of some aspect of the level of the service that is provided

__/__/____

(or time)

Error Budget: defines the amount of errors your appln can have, before act^n's are taken, like stopping deployments to prod^n.

- Community and Governance:

* Open source projects (supported by CNCF)

↓

Sand Box ..... Incubation ..... Graduation

* CNCF → Technical Oversight Committee

⇓

defining and maintaining tech. vision
approving new proj.
accepting feedback from end-user committee
defining common practices

↓

practices the principle of mem^n viable governance

* An SRE Team is responsible for the availability
latency
Performance
efficiency
change mgmt.
monitoring
emergency response
Capacity planning

↓ Services

* Core Tenets of SRE: durable focus on engg.
pursuing max^m change velocity without violating a service's SLO
monitoring (alerts, tickets, logging)
Emergency Response
Change Mgmt.
Demand forecasting and capacity planning
Provisioning
Efficiency and Perf.

__/__/____

- Serverless : no upfront provisioning
  no management of servers
  pay what you use for building appl^n's

  02 Personnas

  Developer — Provedor (CSP)

- Backend-as-a-service (BaaS) :

  Cloud service model on which the developers outsource all the behind the scenes aspects of a web/mobile appl^n so that they have to write and the frontend.

  3rd party API based services that replace core subsets of functionality in an appl^n

- Best Practices for designing a Micro-Services Arch:
  1) create a separate data store for each micro service
  2) Keep code at similar level of maturity
  3) do a separate build for each micro-service
  4) deploy in containers
  5) Treat servers as stateless

- Principles for cloud-native arch. :
  1) Design for autom^n
  2) Be smart with state (design components to be stateless)
  3) Favor managed services
  4) Practice defense in depth
  5) Always be architecting

CLOUD NATIVE ARCHITECTURE

- Linux Foundation is a non-profit tech. consortium as a merger between Open Source Dev. Labs and the Free Standards Group to standardize Linux and promote its commercial adoption.

- CNCF:
  * vendor neutral home for open source proj.
  * 16 graduated proj.
    26 incubating proj.
    69 sandbox proj.
  * Cloud native tech. empower orgⁿs to build and run scalable applⁿs in public, private and hybrid clouds. Containers, service meshes, microservices, immutable infra. and declarative APIs exemplify this approach.
  * Cloud native allows IT and S/W to more faster.
  * Maintainers of CNCF hosted proj.s manage the projs.

- Faas:
  * is focussed on the event-driven computing
  * valuable tool to efficiently and cost-effectively migrate applⁿs to the cloud
  * most central and most definitional element of the serverless stack
  * Kind of cloud computing service that allows developers to build, compute run and manage applⁿ packages as functⁿs without having to maintain their own infra.

- KNative:
  * It is an open-source proj. which extends K8s with serverless capabilities and a simplified dev. exp.
  * KNative Serving defines a set of objects as K8s Custom Resource Definitions (CRDs). These obj.s are used to define and control how your serverless workload behave on the cluster.
  * open-source framework launched to bridge the gap between the containerized and serverless applⁿs among cloud providers.
  * Serverless orchⁿ platform

- Open Container Initiative:
  * The combⁿ of the image manifest, image configⁿ and one or more filesystem Serializations is called the OCI image.
  * Image manifest → provides a configⁿ, and a set of layers for a single container image for a specific arch. and OS.

* OCI develops runC (container runtime) that implements their specifn.

Container Network Interface:

* CNI (Container Network Interface), a CNCF proj. consists of

  Specifn + libraries

  ⇓

  for creating plugins to configure network interfaces in Linux containers, along with a no. of plugins.

* CNI → concerned with the network connectivity of containers and removing allocated resources when the container is deleted.

* CNI → define a common interface between the network plugins and container execution.

* K8s uses CNI as an interface between network providers and K8s pod networking.

* CNI —used by→ container runtimes such as K8s, as well as Podman, CRI-O, Mesos and others.

* The container/pod initially has no network interface. The container runtime calls the CNI plug-in with verbs such as ADD, DEL, CHECK etc. ADD creates a new network interface for the container, and details of what is to be added are passed to CNI via JSON payload.

- Container RunTime Interface:

* CRI is a plug-in which enables kubelet to use a variety of container runtimes without the need to recompile.

* CRI → Protocol Buffers + gRPC API + libraries

* Kubelet ····▸····· Shim (S/W driver) ···▸··· implements the specific details of container engine
  (communicates)

* Docker doesn't support CRI.

* Kubelet —CRI protocol buffers→ CRI Shim → Container runtime ⟵ containers

- Container Storage Interface:

* CSI enables K8s to work with a variety of storage devices.

* Before CSI, volume plugins were serving storage needs for container workloads. in case of K8s.

* CSI: benefit to the container orch^{n} sys. and storage vendors
* Once you have deployed a CSI to a K8s cluster, it's available for use with

  PV → persistent volume
  Storage classes
  PVC → persistent volume chain

* CSI adopted by K8s
  Cloud Foundry
  Mesos
  Nomad

Service Mesh Interface:

* A standard interface for service meshes on K8s.
* defines a common standard that can be implemented by variety of service mesh providers.
* SMI enables flexibility and interoperability and covers the most common service mesh capabilities

  * Top 03 service mesh features:

    a) Traffic policy
    b) Traffic Telemetry
    c) Traffic mgmt.

  * Using the CNCF Envoy project, Open service mesh implements SMI for securing and managing your microservice appl^{n}s.

SRE:

* Treat oper^{n}s as a S/W problem
* 03 metrics:

  SLA (Commitment)
  SLO (goals to meet)
  SLI (actuals)

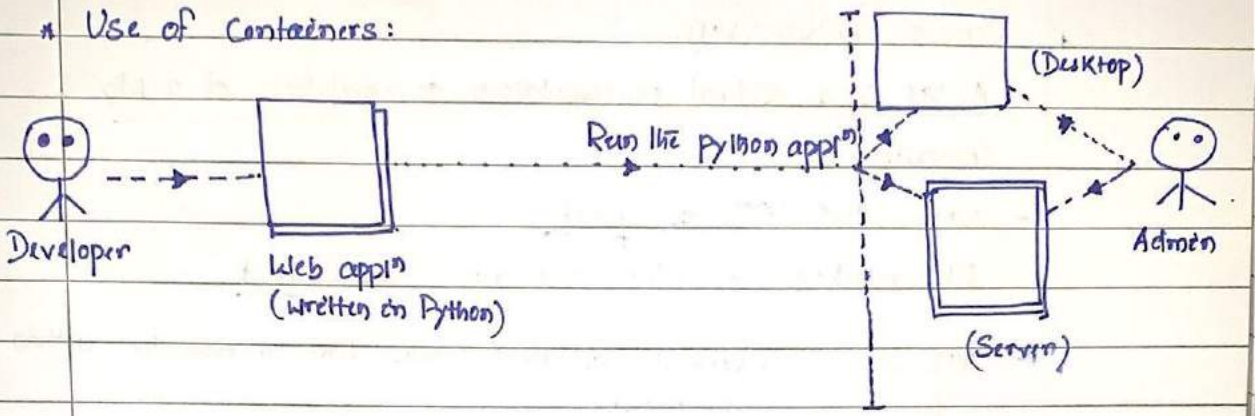
* SRE → automated solutions for operational aspects such as
  on-call monitoring,
  performance
  capacity planning and
  DR

* Error Budget: The max^{m} amount of time a sys. can fail or underperform without violating the contractual terms of the SLA.

__/__/____

CONTAINER ORCHESTRATION:

* Use of Containers:



Pre-requesites:

1) Install OS
2) Install the core Python packages to run the prog.
3) Install the Python ext^ns that the prog. uses
4) configure networking for your system
5) connect to 3rd party systems like a database, cache or storage.

Developer: codes the appln for and knows its dependencies

System Admin: Provides the infrastructure, installs all of its dependencies and configures the system on which the appln runs.

Drawback: process can be error-prone and hard to maintain

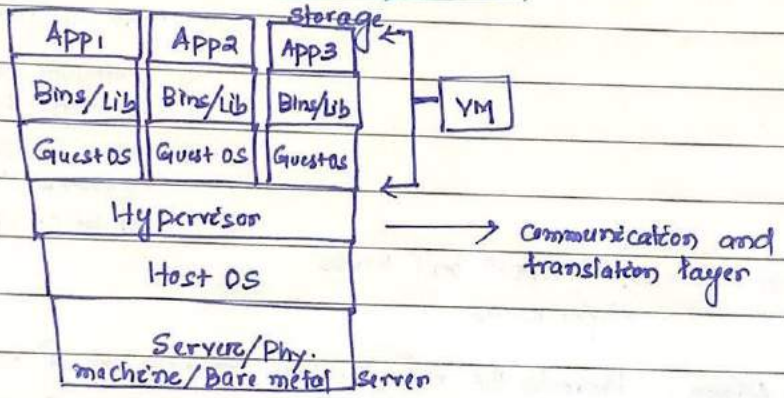
So, servers —single purpose configured→ Running a database or an appln server

* To get more effecient use of server H/W, VMs can be used to emulate a full servers with CPU
memory
storage
networking
OS and
S/W on top.
→ This allows running multiple isolated servers on the same H/W.

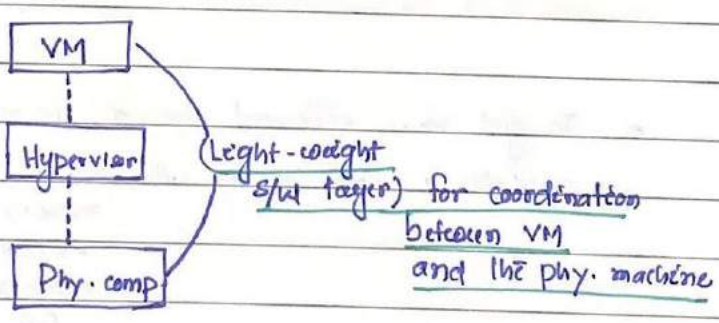
__/__/____

* Virtual Machine (VM):

- A VM is a virtual representation or emulation of a phy. computer.
- VM also known as guest.
  Phy. machines on which VMs run → Host
- VM → virtual env. that looks like a computer within a computer.
  - Has its own CPU
    memory
    network interface
    storage

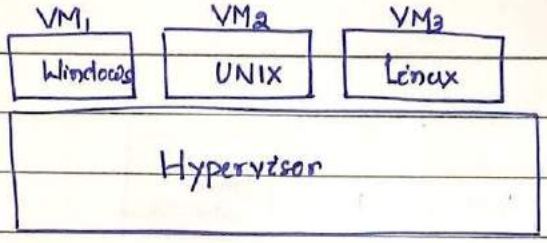


- Each VM runs its own OS and functions separately from the other VMs even when they are running on the same host.
- VMs can be moved between host servers depending on demand.
- Virtualization makes it possible to create multiple VMs.

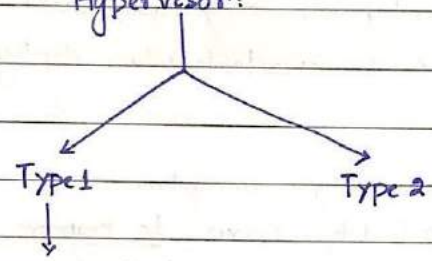


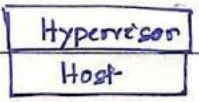
Hypervisor:
1) allocates phy. computing resources such as
   CPU
   memory and
   storage
   to each VM.
2) Keeps each VM separate from others so that they don't interfere with each other.

__/__/____

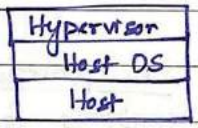


Hypervisor:



* runs directly on the phy. machine
* Bare-metal Hypervisors
* 

* e.g. Citrix Hypervisor
  Microsoft Hyper-V
  Oracle VM
* Often found in server-based env.s like enterprise data centers

* Run on the host computer's OS
* Hosted hypervisors
* 

* e.g. VMWare Work Station
  VMWare Player
* Suitable for personal users or small biz.

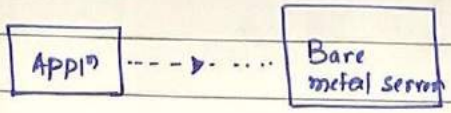
- The limit to how many VMs you can have are really limited by the amount of hard drive space.

- Advantages:
  1) VMs can host multiple OS env.s on a single phy. computer saving phy. space, time and mgmt. costs.
  2) Agility and speed → spinning up a VM is relatively easy and quick.
  3) Lowered downtime → portable and easy to move from one hypervisor to another

Disadvantages
1) VMs → less effecient than real machines because they access the H/W indirectly.
2) Unstable perf. if infra. req. are not met

__/__/____

- Why use a VM?



* Most OS and appln deployments only use a small amount of the phy. resources available when deployed to bare metal.

* By virtualizing your servers, you can place many virtual servers onto each phy. server to improve H/w utilizn. This keeps you from needing to purchase addnal phy. resources like hard drives or hard disks as well as reducing the need for power space and cooling in the data center.

* VMs provide addnal disaster recovery options by enabling failover and redundancy that could previously only be achieved through addnal H/w.

* Because VMs are isolated, they are a good option for testing new applns or setting up a prodn env.

* Server Virtualization:

- Server virtualizn is the process of dividing a phy. server into multiple unique and isolated virtual servers by means of a S/w appln called as hypervisor or VMM (virtual machine monitor).

- Server Virtualizn
  - Full virtualizn
    - uses Hypervisor
      - provides necessary services and features for smooth running of multiple OSs.
  - Para-virtualizn
    - Emerged to address early perf. issues faced by bare-metal hypervisors by modifying the host OS to recognize and interoperate with a hypervisor through commands.
  - OS-Level virtualizn
    - Virtualizn capability is a part of the phy. server OS.

__/__/____

- Benefits: higher server ability
  cheaper operating costs → increased utilizⁿ of existing resources cuts down on the no. of servers needed.
  eliminate server complexity
  increased applⁿ perf.
  deploy workload quicker
  improved disaster recovery → capability to move data or applⁿs quickly from one server to another.

- Why server utilizⁿ?

  Applⁿs ... deploy must → [servers] ⇒ servers only use a small part of their processing power.

  data center may get overcrowded with racks of servers consuming power and generating heat. → Waste of server resources and power.

  servers sit idle because the workload is distributed to only a portion of the network's servers.

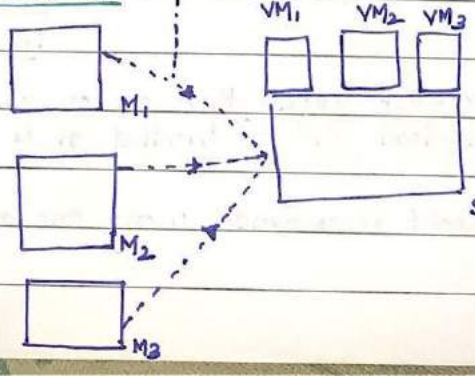
- Drawbacks:

  1) perf. is less reliable. ⇒ Biz. critical applⁿs usually experience better perf. on bare metal servers compared to virtual ones.

  2) capital investment is more expensive. ⇒ initial cost of a virtualized server is usually more expensive than its bare metal counter part.

  3) Not all applⁿs can be virtualized. ⇒ Some virtual envs must maintain some traditional server capacity and adapt into a hybrid ecosystem.

* Server Consolidation: → (migration of network services and applⁿs)



__/__/____

- Server consolidation: popular way to make more efficient use of H/W and resources.

↓

practice of reducing the number of servers or server loc^{n}s in order to use compute resources more effecently and reduce costs.

- Virtualiz^{n} → enables → Server consolidation

- Benefits:

1) Reduce H/W costs : org^{n} needs to purchase less servers.

2) Reduce operating costs : ↑ H/W support agreements
OS licenses
S/W licenses
electricity reqd. by server & data center

3) Consolidated admin tasks : ↑ Backup
patching
monitoring
Reporting

4) reduced environmental impact : Reducing the no. of phy. servers maintained is a huge step in reducing CO_2 emissions.

5) Centralized mgmt.

* Virtual server consolidation Ratio: → Number of virtual servers that can run on each phy. host machine.

VMWare ESX 3.x → 4 VMs/core

Quad-core processor: 04 core → presence of multiple cores allows your Computer to execute multiple processes at the same time.

- The more CPU cores there are in a server, the more virtual-load can be handled at a high-perf. level.

- Microsoft recommends using one core per VM.

- Laptop: 2 cores
  Intel's 8th Gen: 4 cores
  CPU cores → made up of billions of microscopic transistors within a processor that help to make it work.

- How many VMs fit on a server? → It depends on your workload and it depends on how much phy. RAM you have.

⊛ People are worried that if a host has too many VMs, on it, perf. may degrade and the impact of the phy. host failure of a on service delivery is worrisome.

✡ Hyper-threading: Process by which a CPU divides up its phy. cores into vertical cores that are treated as if they are actually phy. cores by the OS.

vertical cores — called → threads

Intel's CPU (2 cores) → 4 threads or 4 vertical cores

- more threads means more work can be done in parallel.
- Increases CPU perf. by improving the processor's efficiency, thereby allowing you to run multiple demanding apps at the same time.

e.g. Intel Pentium 4 (HT-enabled)
Intel Pentium EE 840 (HT-enabled) } processors support hyper-threading.

- RAM: Server's short-term memory
  ↓ With more RAM, server can handle more tasks at once.
  (16 GB - 6 TB)

Container Basics:

* One of the earliest ancestors of modern container tech. is the chroot command.
  - isolate a process from the root filesystem
    basically hide the files from the process
    simulate a new root directory
  - changes the root directory for currently running processes as well as its child processes.

__/__/____

* Root → refers → base directory (/)

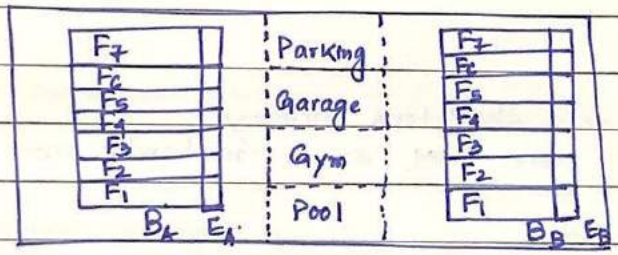
chroot command → creates a fake env. inside which the process runs.
The processes run in such an env. cannot access files and commands outside that env.
or we can say that "the process is jailed within that env."
The created env. → called "chroot jail"

recovering your file system
reinstalling bootloader

- Namespace: (Allows for isolation of global sys. resources between independent processes)

* Are a feature of the Linux Kernel that partitions Kernel resources such that one set of processes sees one set of resources and another set of processes sees a different set of resources.

*



(Apartment society)

Building → B_A B_B
Elevators → E_A E_B
Parking, Gym, Garage, Pool → Shared

Part of same apartment society
⇓
(computer)

(02 namespaces) can reside on the same phy. computer can either share access to certain resources or can have exclusive access.

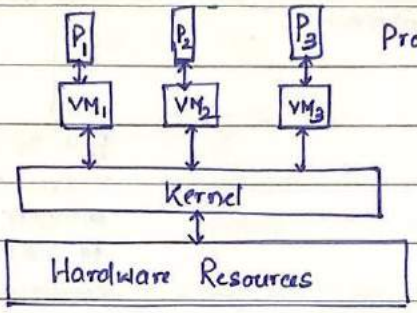
* Linux Kernel 5.6 provides 8 namespaces:

i) pid ii) net iii) mnt iv) ipc v) user vi) uts vii) cgroup viii) time

ipc → inter-process comm
uts → unix time sharing

__/__/____

- Linux Kernel:



* Core interface between the computer's H/W and its processes.
* Kernel is a computer prog. at the core of an operating sys. It is that part of OS that loads first and remains in the main memory.
* Communicates between computer's H/W and its processes, managing resources as effecently as possible.

- Cgroups: (Control Groups)
  - is a ↓ Linux Kernel feature
  - * Allows you to allocate resources (CPU time, system memory, network bandwidth or a comb" of these resources) among user-defined group of tasks (processes) running on a system.
  - * By using cgroups, sys. admins gain control over allocating, prioritizing, denying, managing and monitoring sys. resources
  - * cgroups — organized, hierarchically

- Docker:
  - * open-source platform for building, deploying and managing containerized appl"s
  - * App. code + Exe. comp. + OS lib. + dependencies → package → Container → run that code in any env.

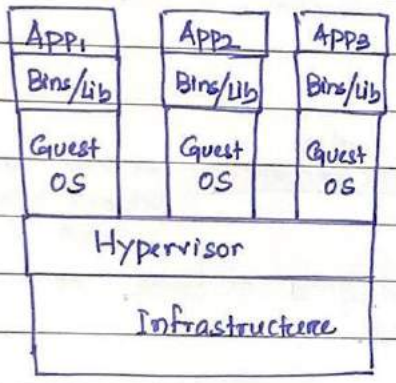
__/__/____

*  develops → create containers → without Docker

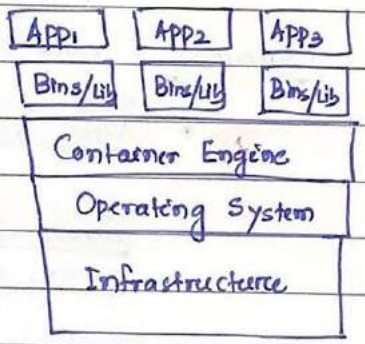
But the platform makes it easier, simpler and safer to build, deploy and manage containers.

* Docker released → public in 2013.

* Set of platform as a service (Paas) products that use OS-level virtualizn to deliver s/w in packages called containers.



Virtual Machine



Containers

* VMs measured → GB
  ↓ contain their own OS, allowing them to perform multiple resource intensive functns at once.

Containers measured → (MB)
  ↓ package app source code, appn dependencies, exe. components and OS level libraries

* VMs: H/W is virtualized to run multiple OS instances.

Containers: provide a way to virtualize an OS so that multiple workloads can run on a single OS instance.

__/__/____

* VMs: Machine-level isolation
  Containers: Process isolation

* VMs: Hypervisor — Creating the VMs. — resource alloc^n
  Containers: Kernel — namespace, cgroups

* VMs: Flexible in creating VMs by adjusting resource limits
  Containers: Run your container image in any env.
  It can be moved from one machine to another.

* VMs: VMWare, KVM
  Containers: Rancher

- Running Containers:

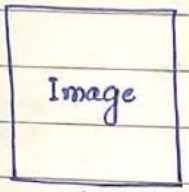
* To run industry-standard containers, you can follow the OCI run-time spec standard.

  OCI (Open Container Initiative)
  ↓ maintains
  a container runtime reference impl^n called runC.
  ↓
  - CLI tool for spawning and running containers on Linux according to the OCI spec^n without having to run a Docker daemon.
  - light-weight, portable container RunTime
  - runC is an open source project, written in Go lang.
  - low-level runtime

* Container Run-time:
  - also known as container engine
  - S/W component that run containers on a host OS.
  - e.g. runC, containerd, Docker, CRI-O

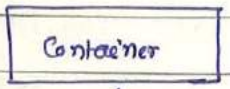
__/__/____

- Relationship between container image and a container:



↓

Immutable file (read-only) that contains source code, libraries, dependencies, tools and other files for an appln to run.



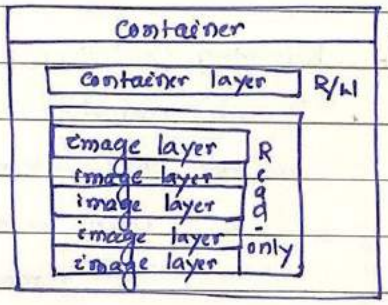
* instance of a Docker image

running image

Once you create a container, it adds a writable layer on top of the immutable image, meaning you can now modify it.

* virtualized runtime env. that provides isolation capabilities for separating the execution of appln from the underplanning sys.

* Docker image is made up of multiple layers, stacked to one another.



* Images can exist without containers, whereas a container needs an image (to run) to exist. So, containers are dependent on images and use them to construct a run-time env. and run an appln.

* Image → logical entity
Containers → real world entity

* Image → created only once
Containers → created any number of times using image.

* Image → write script in Docker file
Container → Run "docker build" command } (creation)

* Image → class
Container → instantiation of that class } → OOPs concept

__/__/____

- Create images from a Dockerfile:
  ↓
  plain text file that contains instⁿs that tell the Docker Build engine how to create an image.

Run image:
↓

⛤ docker run -it ubuntu:18.04 /bin/bash
(ubuntu:18.04 ↔ image name)

- Buildah: tool for building OCI-compatible images through a lower-level coreutils interface.
- Kaniko: tool to build container images from a Dockerfile, inside a container or Kubernetes cluster.
- Podman: (Pod Manager) → tool for managing OCI containers and pods
  ↓
  container engine that is compatible with the OCI containers specⁿ.

⛤ Running containers without root privileges

nginx → Web server

- docker run nginx:1.20 → run the container in foreground

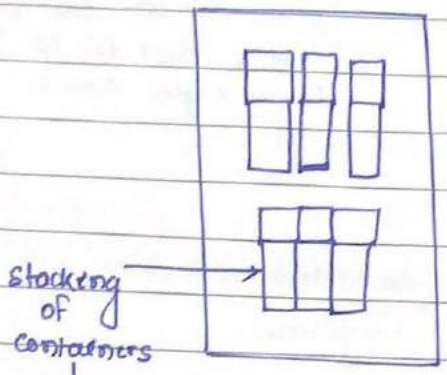
docker run --detach --publish-all nginx:1.20 → run the container in background mode
↓
container id

⛤ docker ps → overview of all the running containers
docker stop <container id> → stop the container

podman run --detach --publish-all nginx:1.20 → run the container using podman

__/__/____

- Building Container Images:



easy to unload with a crane onto a truck irrespective of what is inside

- Docker container image: light-weight standalone executable } package of s/w that includes everything to run an app'n:
  - code
  - runtime
  - sys. tools
  - sys. libraries and settings

- 2015: Docker image format —donated→ OCI-image spec

- Container images include everything a container needs to run.
  - container engine - Docker or CoreOS
  - sys. libraries
  - utilities
  - config. settings and
  - specific workloads

__/__/____

- Container image is composed of layers, added to a parent image.
  - (Base image)
  - intermediate images
  - 1st layer of the Docker image
    Found" upon which all other layers are built
    available in the Docker Hub (public container repository)

Docker File → (Build) → Docker Image → (Run) → Docker Container

- Each cmd" in the Docker file adds a layer to the container image.

Building image: docker build -t my-python-image -f Dockerfile

- Container Registry: repository or coll" of repositories used to store container images for Kubernetes, DevOps and container based appl" devp.
  - Server-side appl" that stores and lets you distribute Docker images.
  - JFrog Container Registry → comprehensive registry supporting Docker containers and Helm chart repositories
  - (upload and download images)

__/__/____

* docker push my-registry.com/my-python-image
  docker pull my-registry.com/my-python-image

* e.g. Node.js appln → source code

  i) Create a Docker file : → instns

```
FROM <base-image>
RUN <dependencies>
WORKDIR <setting the dir.>
COPY <copy the source code
      to the container>
RUN <install dependencies>
CMD <docker run>
```

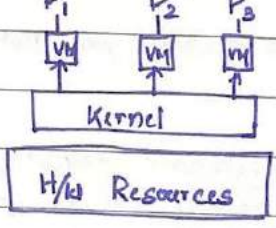
  ii) docker build -t <image name>
  ↳ sending build context to Docker daemon

  iii) docker images → list all the images currently installed on the system

  iv) docker run --detach --publish 3000:3000 <image name>
  ↳ run the container

Security:

* When containers are started on ~~the~~ a ~~same~~ machine, they always share the same Kernel.



* If containers are allowed to call Kernel functions to
  - kill other processes or
  - modifying the host network by creating routing rules

(Risk to the whole system)

WHY CONTAINERS?

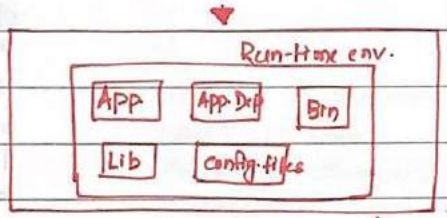
Appl^{n}: DEV ⟶ PROD ⋯▸⋯ Containers is the solution. __/__/____

deployment- and effectent run post deployment-

Problems arise when DEV ⋯▸⋯ PROD

supporting S/W versions are not identical

differences in
- network topology
- Security policies
- storage



(Container)

⟹ differences in OS and underlying infrastructure are abstracted away.

---

* Starting processes as root or administrator → Security risk
* Use of public images
  - Public image registries : Docker Hub, Quay

Risk ↓

These images were not modified to include malicious S/W.

* DockerFile Best Practices: (image builds)

1) Avoid unnecessary privileges :
   a) avoid running containers as root (UID 0)
   ← b) don't bind to a specific UID (0644)
   c) make executables owned by root and not writable

run the container as non-root user

root owned by, user executed by non-root user shouldn't be world writable

__/__/____

2) Reduce attack surface
   a) leverage multi-stage builds
   b) use distroless images or build your own from scratch
   c) update your images frequently
   d) watch out for exposed ports

Multi-stage build includes only the minimal reqd. binaries and dependencies in the final image, and not build tools or intermediate files.
→ reduces the attack surface decreasing vulnerabilities

distroless images are designed to contain only the minimal set of libraries reqd. to run Go, Python or other frameworks.

3) Prevent confidential data leaks
   a) never put secrets or credentials in Dockerfile instructions (use Docker secrets / Kubernetes secrets)
   b) prefer COPY over ADD
   c) be aware of the Docker context, and use .dockerignore
      ⇓
      "." parameter → build context
      ↳ dangerous as you can copy confidential or unnecessary files into the container.

4) Others
   a) reduce the no. of layers and order them intelligently (grouping multiple commands together will reduce the no. of layers)
   b) add metadata and labels (image management like including the appln version)
   c) leverage linters to automate checks
   d) scan your images locally during devp.
      ↓
      shift left security

Tools like Haskell Dockerfile linter (Hadolint) can detect bad practices in your Docker file and even expose issues inside the shell commands executed by the RUN instn.

5. Beyond image building

a) protect the docker socket and TCP connections

b) sign your images and verify on runtime

c) avoid long mutability

d) don't run your env. as root

e) include a health check

f) restrict your appln capabilities

docker content trust, docker notary, harbor notary

⇓

digitally sign images

include a HEALTHCHECK instn in your Dockerfile wherever possible

(Kubernetes → use Liveness Probe config)

big privileged door into your host sys.

/var/run/docker.sock

⇓

Correct permissions

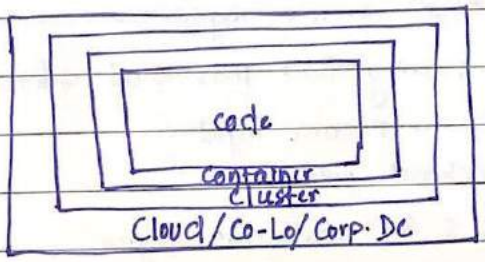
In execution, restrict the appln capabilities to the minimal reqd. set

- --cap-drop flag (Docker)
- Security Context. capabilities. drop (Kubernetes)

Tags are a volatile ref. to a concrete image version in a specific point in time.

Cloud Native Security (4C's)

Cloud → Cluster → Container → Code



__/__/____

cloud: trusted computing base of a Kubernetes cluster.
CSPs ⟶ security best practices

cluster: infrastructure security

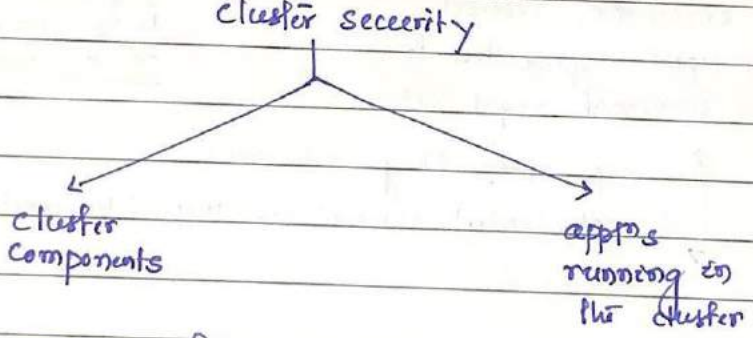
i) all access to Kubernetes control plane
→ not allowed publicly over the internet
→ controlled by network ACLs restricted to the set of IP addr. needed to administer the cluster.

ii) nodes configured to only accept conn^ns from the control plane on the specified ports

iii) Kubernetes access to cloud API ⟶ principle of least privilege

iv) access to etcd → should be limited to control plane only

v) etcd encryption → etcd disks should be encrypted at rest.



area of concern for workload security:

1) RBAC authorization → access to Kubernetes API
2) Authentication
3) app^n secrets mgmt.
4) ensuring that pods meet defined Pod Security Standards
5) cluster resource mgmt.
6) network policies
7) TLS for Kubernetes ingress

__/__/____

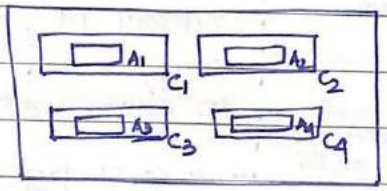
Container:

Container vulnerability scanning
image signing and enforcement
disallow privileged users
use container runtime with stronger isolation

Code:

access over TLS only
limit port ranges of comm^n
3rd party dependency security
static code analyses
dynamic probing attack → few automated tools that you can run against your service to try some of the well-known service attacks.

Container Orchestration Fundamentals:

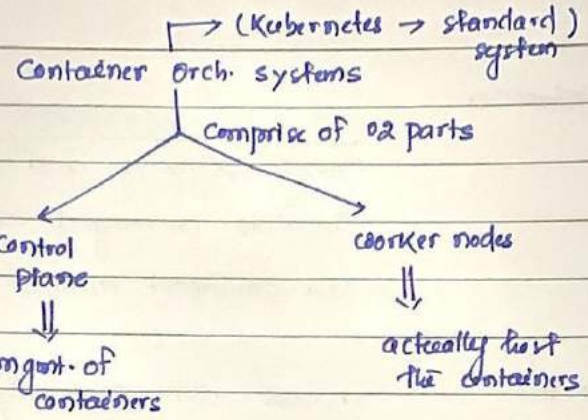


loosely coupled
isolated
independent
| services ⇒ micro-service arch.

container orch^n systems: → (cluster of multiple servers and host containers on top)

1) provide compute resources like VMs where containers can run on
2) schedule containers to servers in an efficient way
3) allocate resources like CPU and memory to containers
4) manage the availability of containers and replace them if they fail
5) scale containers if load increases
6) provide networking to connect containers together
7) provision storage if containers need to persist data

__/__/____



- Networking:

* Microservice arch. depends on network comm^{n}.
* Network namespaces allow each container to have its own unique IP address.
* Modern impl^{n}s of container networking are based on the Container Network Interface (CNI).

* The type of network a container uses is transparent from within the container.
  - Bridge → (when you need multiple containers to communicate on the same Docker host)
  - Overlay
  - Macvlan
  - custom network plugin

  when you need containers running on diff. Docker hosts to communicate

  Take care of IP add. mgmt.

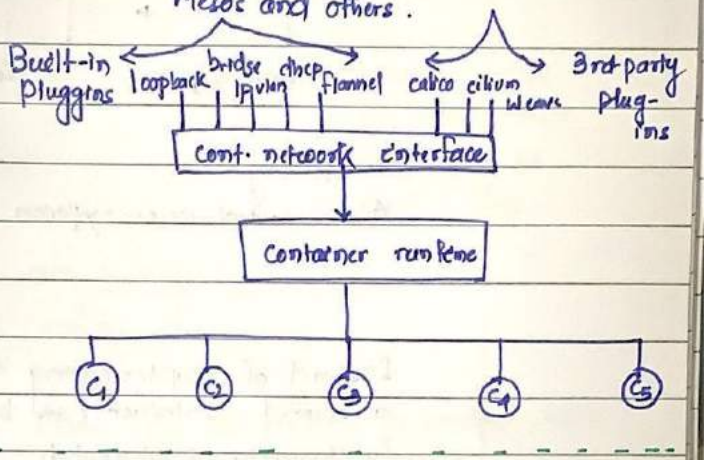
* From a container's point of view, it has
  - Network interface with IP add.
  - Gateway
  - Routing table
  - DNS services
  - other net. details

* Host network - when the network stack should not be isolated from the Docker host

- CNI: standard that can be used to write or configure network plugins and makes it easy to swap out diff. plugins in various orch^{n} platforms.
- a CNCF project consists of a spec^{n} and libraries for writing plugins to configure network interfaces in Linux containers, along with a no. of plugins.
- CNI: deals with network connectivity of containers and removing allocated resources when the container is deleted.
- Kubernetes use CNI as an interface between network providers and Kubernetes pod networking.

__/__/____

- CNI is used by container runtimes such as Kubernetes, as well as Podman, CRI-O, Mesos and others.

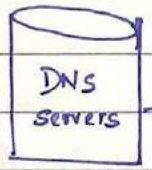


- Service Discovery & DNS:

* In container orch^{n} systems,
  - 100's/1000's of containers with individual IP addresses
  - containers deployed on variety of diff. hosts, diff. data centers or even geo-loc^{n}s
  - containers or services need DNS to communicate. Using IP add. is nearly impossible.

solution ⟶ automation ⟶ all info^{n} put in Service Registry

* Finding other services in the network and requesting info^{n} about them ⟶ Service discovery

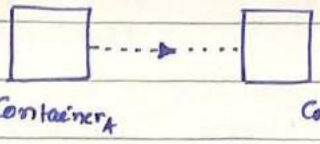


* have a Service API ⟶ register new services as they are created

* Key-value store: store info^{n} about services
  e.g. etcd, Consul, Apache Zookeeper

__/__/____

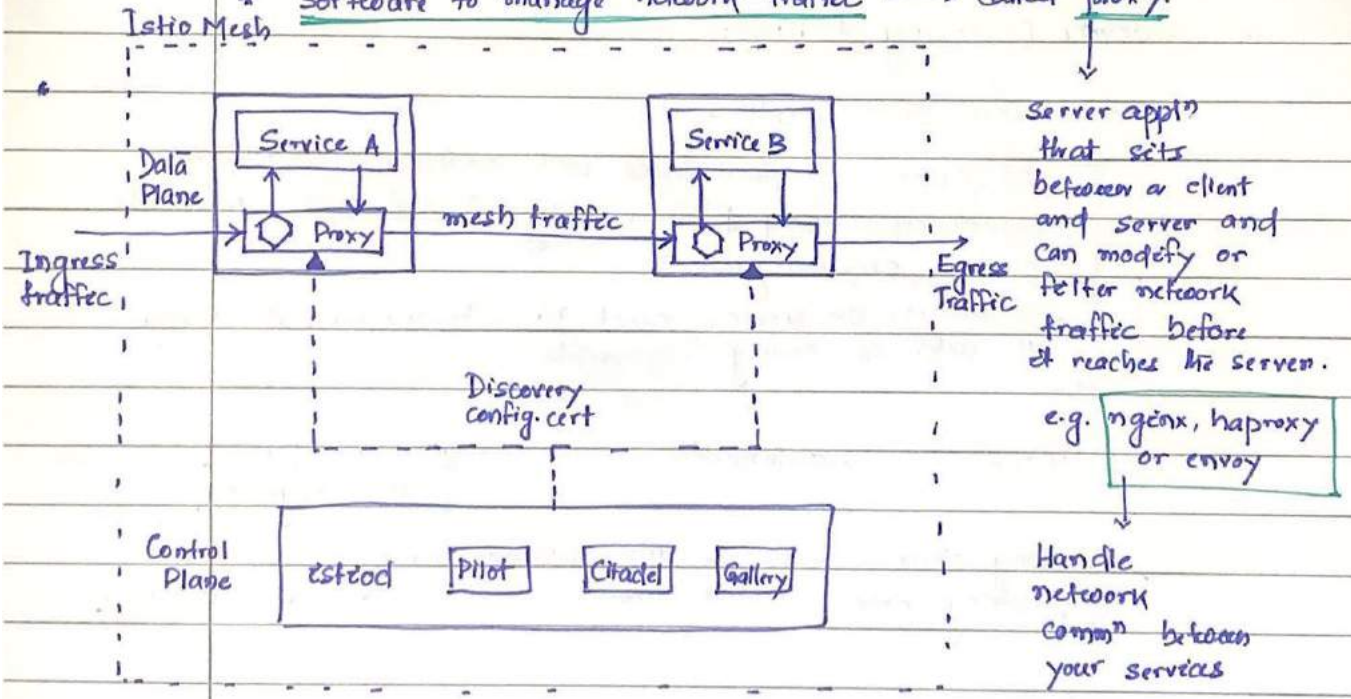
- Service Mesh:



Monitoring
Access control or encryption
→ of network traffic → desired when containers communicate with each other

Instead of implementing this functionality into your appln, a second container can be started that has this functionality implemented.

* Software to manage network traffic → called proxy.



Server appln that sits between a client and server and can modify or filter network traffic before it reaches the server.

e.g. nginx, haproxy or envoy

Handle network comm^n between your services

* Istio Architecture:

- When a service mesh is used, appln's don't talk to each other directly, but the traffic is routed through the proxies instead.

__/__/____

- Popular service meshes → Istio and Linkerd
- Proxies in a service mesh form the data plane.
  ↓
  networking rules are implemented and shape the traffic flow.
- Networking rules are managed centrally in the control plane of the service mesh.
  ↓
  define how traffic flows from one service to another and what config should be applied to the proxies
- Config file : tells → service mesh that services should always communicate encrypted.
  ↓
  uploaded to the control plane and distributed to the data plane to enforce the new rule.
- Service Mesh Interface (SMI): define a spec for how service mesh from various providers can be implemented.
  ↓
  Standard for Service mesh providers config with Kubernetes

* Service Mesh:

- is a way to control how different parts of an app share data with one another.
- dedicated infrastructure layer built right into an app
- Each part of an app — called → service

- Service mesh is a way to comprise a large no. of discrete services into a functional appln.

- Designed to handle a high volume of network-based inter-process comm^n among appln infra. services using APIs.

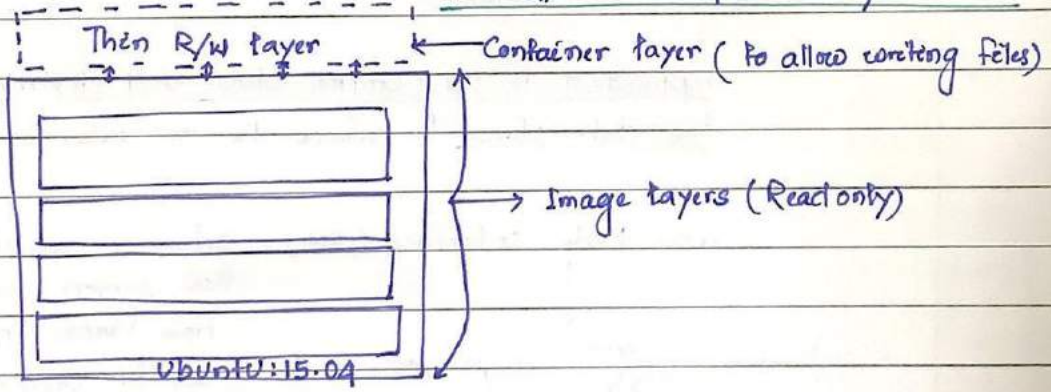
- common features provided by service mesh:

  Service discovery
  load balancing
  encryption
  failure recovery

* Storage:

- From storage perspective: containers have a major disadvantage.

  when the container is stopped or deleted, the read-write layer is lost.

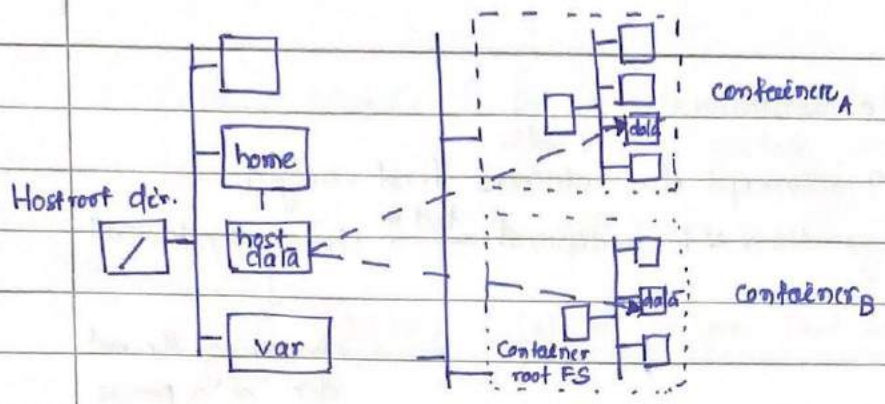


Container
(based on ubuntu:15.04 image)

- R/W layer is lost when the container is stopped or deleted.
- To persist data, you need to write data to disk.
- A container need to write data to a volume on a host.

  Directories that reside on a host — passed → through into the container file system.

- Data is shared between 02 containers on the same host.



- Challenges:

a) Persisting the data on the host where the container was started → when you orchestrate a lot of containers

b) Data needs to be accessed by multiple containers that are started on diff. host systems

c) When a container gets started on a diff. host, it still should have access to its volume.

(problem mitigation)

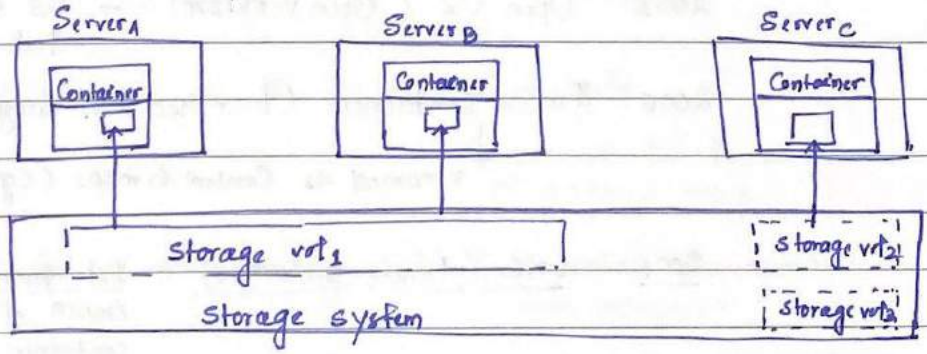
Kubernetes → require → robust storage system that is attached to the host servers.

$container_A$ and $container_B$ can share a volume to read and write data.

- Container Storage Interface (CSI):

⇓

standard for exposing arbitrary block and file storage systems to containerized workloads on container orch^n sys. like Kubernetes



1. Host based persistence
2. Implicit per container storage
3. Explicit shared storage

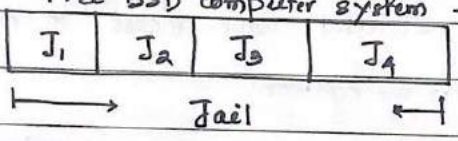
__/__/____

- History of Containers:

1979: concept of containers first emerged.
Unix V7 development led to chroot system call
↓
changing the root dir. of a process and its children to a new loc^n in the FS.

Beginning process isolation → segregating file access for each process

2000: Free Berkeley software distribution (BSD) Jails
Free BSD computer system → partition into several independent smaller sys. called "jails" having separate IP add.



2001: Linux Vserver (Linux Virtual Server)
↓
jail mechanism that can partition resources on a comp. sys.

2004: Solaris Container (combines sys. resource controls and boundary separation provided by zones)

2005: Open VZ (Open Virtuzzo) → OS level virtualiz^n tech. for Linux

2006: Process Containers (launched by Google)
↓
renamed as Control Groups (cgroups)

2008: LXC (Linux Containers) → 1st most complete impl^n of Linux container manager

__/__/____

2011: Warden (CloudFoundry started Warden which can isolate envs on any OS running as a daemon and providing an API for container mgmt.)

2013: LMCTFY (Let me Contain That For you)

↓

Open source version of Google's container stack, providing Linux appln containers.

2013: Docker → used LXC in initial stages and later replaced container mgr. with its own library, libcontainer.

2016: Container Security realized → DevSecOps

2017: Maturity in Container Tools (Kubernetes — CNCF - 2016)

2018: CSPs —offer→ managed Kubernetes service

2019: New Runtime engines started replacing Docker runtime engine → containerd / CRI-O

- Best Practices for building Containers:
  1) Package single app per container
  2) Properly handle PID 1, signal handling and zombie process (init system)
  3) optimize for the Docker build cache (accelerate the building of container images)
  4) Remove unnecessary tools (to protect your apptns from attackers)
  5) Build the smallest image possible
  6) Reduce the amount of clutter in your image

__/__/____

7) Try to create images with common layers
8) Scan images for vulnerabilities
9) Properly tag your images
10) Carefully consider whether to use a public image

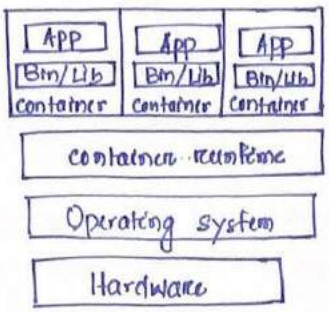
google/cloud-sdk: ← 193.0.0
- google/cloud-sdk: → image name
- 193.0.0 → image tag

- Play with Docker (PWD) → Project sponsored by Docker
  - allows users to run Docker commands in a matter of secs.
  - → Free Linux VM in browser → Build and run containers and create clusters in Docker Swarm Mode

* The usage of containers and VMs are not mutually exclusive. (Two or more events that cannot happen simultaneously)
* Container isolation achieved by namespaces and groups
* Service Discovery → Automatic process of discovering services on a network

CONTAINER ORCHESTRATION

- Containers: * are an appln centric method to deliver high-performing scalable applns on any infra. of your choice.



  * encapsulate micro-services and their dependencies but do not run them directly.
  * containers run container images.

- Container Orch^{n} Tools:
  * Marathon → framework to run containers on Apache Mesos
  * Nomad → HashiCorp

- Container Runtime Interface (CRI):
  * Container Run Time → at the lowest layers of a K8s cluster, CRT is the s/w that starts and stops containers.
  * CRI: New plug-in API for container runtime in K8s
  * CRI: consists of protocol buffers and gRPC API and libraries.
  * Kubelet (gRPC client) - - - communicates - - - CRI Shim (container runtime) (gRPC server) over UNIX sockets using the gRPC framework.

    Protocol buffer API → 02 gRPC services
    Image Service
    Runtime service

- Container Network Interface (CNI):
  * CNI (a CNCF proj.) → specifn + libraries
    - for writing plugins to configure network interfaces in Linux containers

- Container Storage Interface (CSI):

* Using CSI, 3rd party storage providers can write and deploy plugins exposing new storage sys. in K8s. without ever having to touch the core Kubernetes code.

* CSI Adoption by container orchestrators:

  Cloud Foundry
  K8s
  Mesos
  Nomad

* Once you have deployed a CSI to a K8s cluster, it is available for use with persistent volumes (PV), storage classes and persistent volume claims (PVCs).

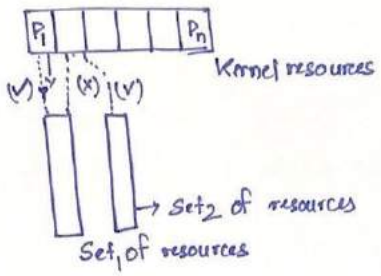
Service Mesh Interface (SMI):

* SMI ⟶ specfn that covers the service mesh capabilities.
  a) traffic policy b) traffic telemetry c) traffic mgmt.

* defines a standard set of K8s custom resource definitions (CRDs) and APIs for service meshes.

* open source specfn for interoperable service meshes on K8s

- Namespace:
  - * Linux network namespaces are a kernel feature allowing us to isolate network envs through virtualizn.
    e.g. Using network namespaces, you can create separate network interfaces and routing tables that are isolated from the rest of the system and operate independently.



It is a feature of the Linux kernel that partitions kernel resources such that one set of processes sees one set of resources while another set of processes sees a different set of resources.

- Cgroups:
  - * provides resource limiting
    prioritization
    accounting
    control

- runC:
  - * It's a low level tool and not designed with an end user in mind.
  - * Has the ability to run containers without root privileges.
  - * command line client for running applns packaged according to the OCI format.
  - * It's a container run time originally developed by Docker and later extracted out as a separate open source tool.

- Docker file:
  - * Text document that contains all the commands a user could call on the command line to assemble an image.
  - * docker build command ... builds an image from a Docker file and a context.
    ⇓
    The build's context is a set of files at a specified locn PATH or URL.
  - * FROM command → sets the base image for the rest of the instns.
    MAINTAINER command → author of the generated images

RUN Command → used to execute any commands

* Alpine images tightly controlled and small in size → good for Go, bad for Python

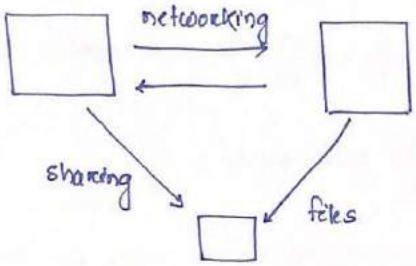
* Privileged container in Docker is a bad idea.
  → Containers that have all of the root capabilities of a host machine
  → One use case of a privileged container: Running a docker daemon inside a docker container.

* Do not expose unused ports
* Do not run SSH within containers
* do not share the host's network namespace

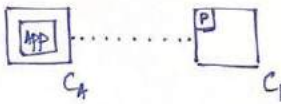
} Best practices - Docker runtime security

- Networking:

*



* Two containers can talk to each other in one of the 02 ways.
  a) Communicating through networking → containers can send and receive requests to other appl^n's using networking.
  b) Sharing files on disk



An appl^n running in one container will create a network ~~container~~ connection to a port on another container.

* Docker comes with network drivers geared towards different use cases. The common network types are:
  i) Bridge - limited to containers within a single host running the Docker eg. port mapping needs to be configured. default Docker network type.
  ii) Overlay - multi-host network comm^n, uses network tunnels to communicate between hosts
  iii) macvlan - connect Docker containers directly to the host network interfaces through layer 2 segmentation.

* Network plugins in K8s come in a few flavors:
  a) CNI plugins
  b) Kubenet plugin - default network plugin
* Flannel - overlay network provider that can be used with K8s
* Calico - most popular open-source networking and networking security solution for K8s.

DNS Server:
* DNS is a service that translates domain names into IP addresses.
* DNS has been designed to use port 53.
* When you deploy a container on your network, if it cannot find a DNS server defined in /etc/resolv.conf, by default it will take on the DNS configured for the host machine.
* DNS Server takes a

FQDN (www.example.com) —returns IP add.→ 93.184.216.34

Service Mesh:
* A service mesh is a way to control how diff. parts of an app'n share data with one another.
* A service mesh is built into an app as an array of network proxies.
* Individual proxies that make up a service mesh are called "sidecars"; since they run alongside each service, rather than within them.
* SM: manages all service-to-service comm'n within a distributed s/w sys. It accomplishes via the use of "sidecar" proxies that are deployed alongside each service through which all traffic is transparently routed. Sidecars enable service req. to flow through the app'n.
* Popular service meshes: Istio, Linkerd, Consul, Kuma, Maesh

Supporting proxies: Envoy, HAProxy, NGINX, MOSN

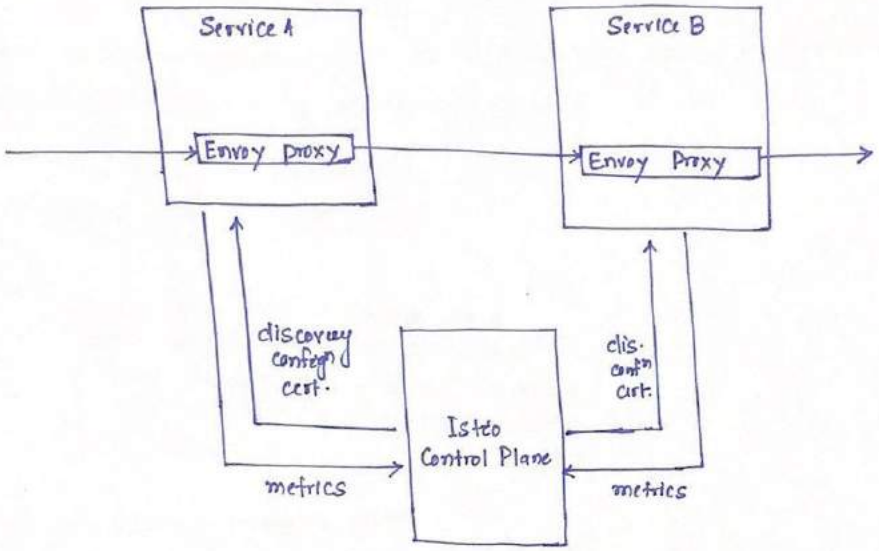
* SM: manages "east-west" traffic within a microservices based s/w system.
  ↓
  network traffic within a data center, network or K8s cluster

* SM works with a service discovery protocol to detect services as they come up.

* Istio backed by Google, IBM and Lyft is the best-known service mesh arch. K8s is the only container orch^{n} framework supported by Istio.

* Proxies → data plane
  management processes → control plane

* Istio Mesh:



→ SM: dedicated infra. layer that you can add to your appl^{n}s.
→ SM: addresses more complex operational requirements like
  A/B testing
  canary deployments
  access control
  encryption
  end-to-end authentic^{n}

→ Control Plane: [Secure service-to-service comm^{n} in a cluster with TLS encrypt^{n}
  [automatic load balancing
  (features)
  takes your desired config^{n} and its view of the services

→ Data Plane: communication between services
  SM uses a proxy to intercept all your network traffic, allowing a broad set of appl^{n} aware features based on config^{n} you set.

→ Istio extends K8s to establish a programmable appln-aware network using the Envoy proxy.

→ Istio is platform independent and is designed to run on

Cloud
On-premises
K8s
Mesos

→ Istiod : control plane

→ Envoy : data plane agent

Service Discovery:

* Process of automatically detecting devices and services on a network
* Service discovery protocol (SDP) is a networking standard that accomplishes detection of networks by identifying resources.
* 02 types of service discovery:

i) Server-side — allows client apps to find services through a router or a load balancer

ii) client-side — allows client apps to find services by looking through or querying a service registry

* Service discovery implementations:

i) DNS (NGINX proxy which can periodically poll for service discovery)
ii) Key-value store and side-car (Consul or Zookeeper)
iii) Specialized service discovery and library/sidecar (Eureka)

__/__/____

Working with Kubernetes
~~Kubernetes Fundamentals~~

- Pod: smallest compute unit on Kubernetes
- Kubernetes: ↑ config^n mgmt.
  - cross-node networking
  - routing of external traffic
  - load balancing
  - scaling of the pods ↓

- Kubernetes Objects:
  - * Kubernetes objects are persistent entities in the Kubernetes system. (represent the state of the cluster)
    - describe
      - → what containerized appl^ns are running (and on which nodes)
      - → the resources available to those appl^ns
      - → the policies around how these appl^ns behave, such as
        - restart policies
        - upgrades
        - fault tolerance
  - * Kubernetes obj. → record of intent ⇒ once you create that obj., Kubernetes system will constantly work to ensure that the obj. exists.
  - * To work with Kubernetes objects -- whether to create, modify or delete them
    - need ↓
    - Kubernetes API (core of Kubernetes' control plane)
  - → Kubectl CLI: ..... makes the API call for you
  - * In Kubernetes, deployment obj. → represent an appl^n running on your cluster

__/__/____

* Every Kubernetes obj. includes 02 nested obj. fields that govern the obj.'s config^{n}.

i) obj. spec — you have to set this when you create the obj, providing a desc^{n} of the char. you want the resource to have; its desired state.

ii) obj. status — describes the current state of the obj.

* When you create an object in Kubernetes, you must provide

a) obj. spec^{n} → desired state

b) Basic info^{n} about the obj. e.g. name

→ When you use the Kubernetes API to create the obj., (either directly or via kubectl), API request must include that info^{n} as JSON in the req. body

(JavaScript Obj. not^{n}, standard text based format for representing structured data based on JavaScript obj. syntax.

→ Sending some data from server to client so that it can be displayed on a web page)

→ info^{n} in a .yaml file → kubectl → makes the API req. → info^{n} gets converted to JSON

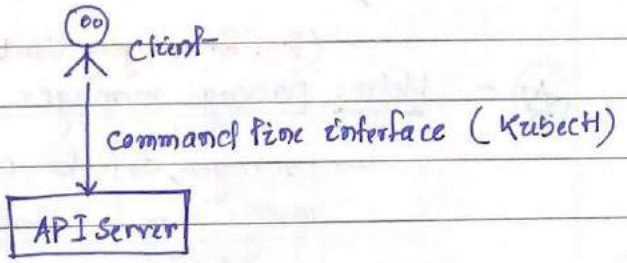
__/__/____

```
apiVersion: apps/v1   → which version of the Kubernetes API you are using to create this obj.
kind: Deployment      ← what kind of obj. you want to create
metadata:             → data that helps uniquely identify the obj.
  name: nginx-deployment
spec:                 → what state you desire for the obj.
  selector:
    matchLabels:
      app: nginx
  replicas: 2 # tells deployment to run 2 pods matching the template
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.19
        ports:
        - containerPort: 80
```

→ Create a deployment: kubectl apply -f https://k8s.io/examples/application/deployment.yaml --record

o/p: deployment.apps/nginx-deployment created

- Interacting with Kubernetes:



__/__/____

- kubectl api-resources

name	shortnames	api-version	namespace	kind
configmaps	cm	v1	true	configmap
namespaces	ns	v1	false	namespace
pods	po	v1	true	pod

- kubectl explain pod → get doc^n of various resources

Kind : Pod
Version: v1

[Pod → coll^n of containers that can run on a host. This resource is created by clients and scheduled onto hosts.]

- kubectl explain pod.spec

- kubectl --help (kubectl → controls → Kubernetes cluster manager)

↓ overview of all the commands

help from the terminal window

- To create an object in Kubernetes, from a YAML file,

↓

kubectl create -f <yourfile>.yaml

- Tools for interaction with K8s:

1) Kubernetes dashboard
2) detailed/k9s
3) Lens
4) VMWare Tanzu Octant

- Helm: package manager for Kubernetes (Tool for managing charts, tool that streamlines installing and managing K8s apps)

↳ packages K8s objects in so called Charts which can be shared with others via a registry. (Helm Charts is a coll^n of files that describe a related set of K8 resources)

Helm is the K8s equivalent of yum or apt.

→ * Open-source proj.

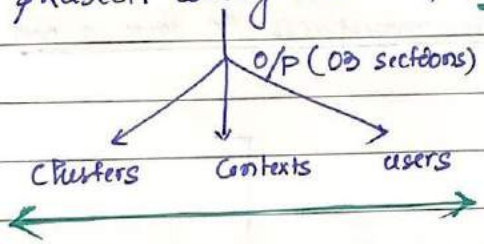
__/__/____

- Artifact Hub: web-based appln that enables finding, installing, and publishing k8s packages. and configns for CNCF proj.

- Kubectl: command line tool lets you control k8s clusters.

For confign, kubectl looks for a file named config in the $HOME/.kube directory.

$kubectl config view → current kubectl config



1) define a k8s object in a YAML file (pod running a nginx container on port 80)

$ vim pod.yaml. [Object specn]

2) $kubectl create -f pod.yaml ⇒ pod/nginx created

3) $kubectl get pods

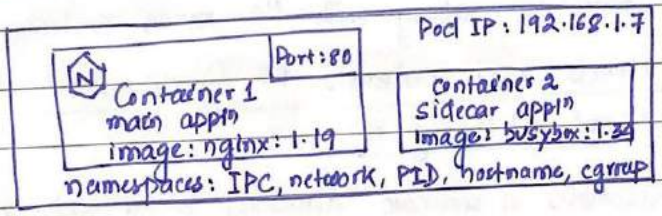
name	ready	status	restarts	age
nginx	1/1	running	0	10s

4) $kubectl delete pod nginx ⇒ pod "nginx" deleted

5) $ kubectl get pod → no resources found in default namespace.

6) $ kubectl explain --help

- Pod Concept:



* A pod is a group of one or more containers with shared storage and network resources, and a specifn of how to run the containers.
* A pod's contents are always co-located and co-scheduled and run in a shared context.

* Pod describes a unit of one or more containers that share an isolation layer of namespaces and cgroups.

* smallest deployable unit in Kubernetes ⇒ Kubernetes is not interacting with containers directly.

* all containers inside a pod share an IP address and can share via the filesystem.
* multiple containers share namespaces to form a pod.

* apiVersion: v1
  Kind: pod
  metadata:
    name: nginx-with-sidecar
  spec:
    Containers:
    - name: nginx
      image: nginx:1.19
      ports:
        - ContainerPort: 80
    - name: count
      image: busybox:1.34

simple pod object with two containers

* Sidecar Container:
  → a utility container in a pod that's loosely coupled to the main application container
  → should run along with the main container in the pod.
  → extends and enhances the functionality of current containers without changing it
  → Resembles a sidecar attached to a motorcycle

__/__/____

- Init Container:
  * is the one that starts and executes before other containers in the same pod.
  * meant to perform initialization logic for the main appln hosted on the pod.
  * always run to completion

- Create a pod: $ Kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml

- One CPU in kubernetes ≅ 1 vCPU/core for cloud providers and 1 hyperthread on bare-metal Intel processors

- Liveness probe: Configure a health check that periodically checks if your appln is still alive

- Security Context: set user & group settings, as well as Kernel capabilities.

- $ docker run --detach nginx:1.19 → run a container
  $ docker ps → check the running status of the container on that host

container id	image	command	created	status	ports	names
	nginx:1.19					

$ kubectl run nginx --image=nginx:1.19 → run a pod
⇒ pod/nginx created

$ kubectl get pods

name	ready	status	restarts	age
nginx	1/1	running	0	5s

$ kubectl describe ~~nginx~~ pod nginx → more details about nginx pod.

__/__/____

$ curl http://192.168.36.137 ⇒ talking to the nginx container

(pod ip address)

$ Kubectl create -f pod-two-containers.yaml

⇓

pod/nginx-with-sidecar created

$ Kubectl get pods

⇓

name	ready	status	restarts	age
nginx	1/1	running		
nginx-with-sidecar	2/2			

2 out of 2 containers are running

- Workload Objects:

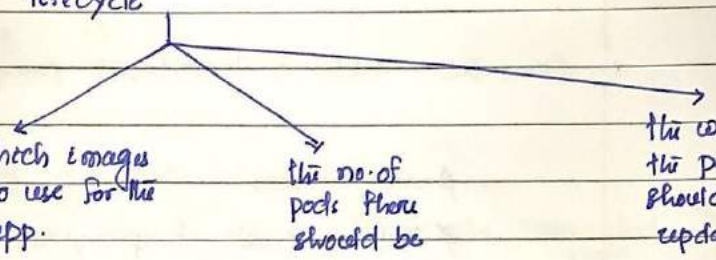
* ReplicaSet: a controller obj. that ensures a desired number of pods is running at any given time.
  - ReplicaSets can be used to scale out apps and improve their availability. They do this by starting multiple copies of a pod def.
  - ensures how many replica of pod should be running.
  - To manage the scaling of pods, Kubernetes uses an API object called a ReplicaSet.

* Deployment: most feature-rich object in Kubernetes.
  - describe the complete app life-cycle
    - they do this by managing multiple ReplicaSets that get updated when the app is changed by a new container image.

__/__/____

→ Resource obj. in Kubernetes that provides declarative updates to appln's. (for pods and replica sets)

→ Deployment - allows you to describe an appln's lifecycle



→ deployments are perfect to run stateless appln's in K8s.

(appln prog. that doesn't save client data generated in one session for use in next session with that client)

* Stateful Set: - workload API object used to manage stateful appln's.

- Controller that helps you deploy and scale groups of Kubernetes pods.
- try to retain IP addresses of pods and give them a stable name, persistent storage and more graceful handling of scaling and updates.

(Programs that saves client data from the activities of one session for use in the next session)

e.g. databases (mongodb elastic search) etc.

* daemon Set: → ensures that a copy of the pod runs on all (or some) nodes of your cluster

→ perfect to run infra.-related workload e.g. monitoring or logging tools

kind = daemon-set

__/__/____

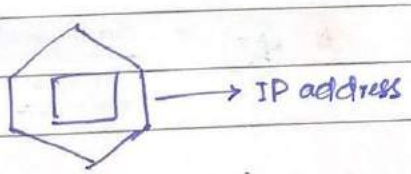
* Job: Creates one or more pods that execute a task and terminate afterwards.

Job objects ⟶ one-shot scripts → database migrations or administrative tasks

* Cron Job: adds a time-based config to jobs.

$ vim pod.yaml

$ kubectl create -f pod.yaml → pod/simple-nginx-pod created



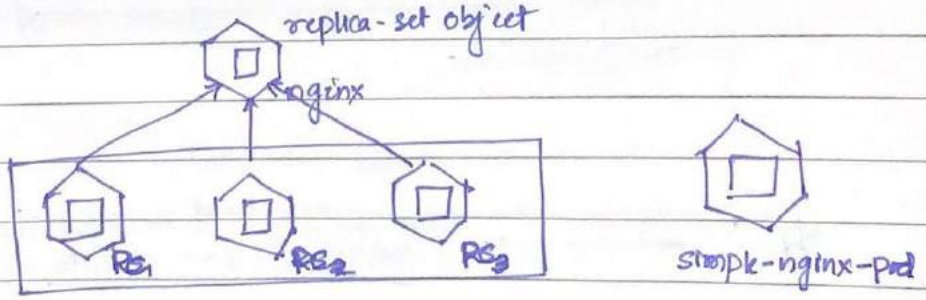
Create multiple ~~instances~~ copies of the pod object

$ vim replicaset.yaml

replicas =

$ kubectl create -f replicaset.yaml

→ replicaset.apps/nginx created



__/__/____

$ kubectl get pods

name	ready	status	restarts	age
nginx-PC1	1/1			
nginx-PC2	1/1			
nginx-PC3	1/1			
simple-nginx-pod	1/1			

$ kubectl get pods -o wide

name	ready	status	restarts	IP	node	nominated node	readiness gates

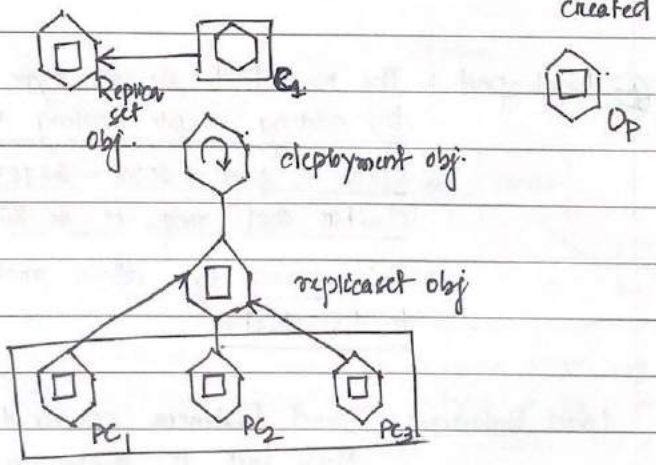
$ kubectl scale --replicas=10 rs/nginx → (10 copies of the pod created)

⇓

(replicaset.apps/nginx scaled)

$ vim deployment.yaml (deployment obj manage multiple replica sets)

$ kubectl create -f deployment.yaml ⇒ deployment.apps/nginx created



$ kubectl scale --replicas=10 deployment/nginx

⇓

(deployment.apps/nginx scaled)

__/__/____

↳ (update the deployment)

$ kubectl set image deployment/nginx nginx=nginx:1.20

↓

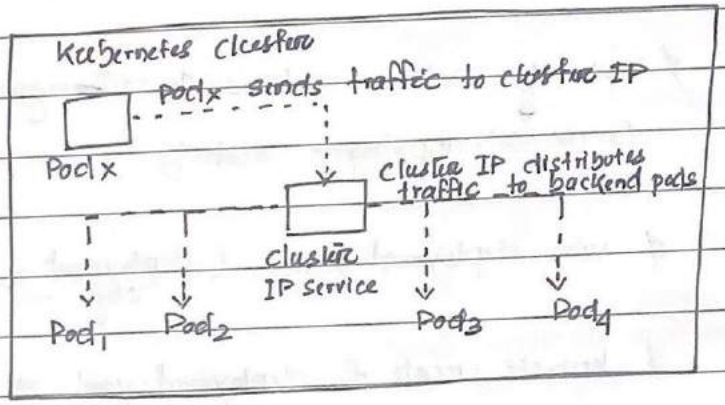
deployment.apps/nginx image updated

- Networking Objects:

Cluster IP: virtual IP inside Kubernetes that can be used as a single endpoint for a set of pods.

↓ default service type.

↳ this service-type can be used as a round-robin load balancer.



Node-port: The NodePort Service type extends the clusterIP by adding simple routing rules.

→ opens a port (30000 - 32767) on every node in the cluster and maps it to the Cluster IP.

→ this service type allows routing external traffic to the cluster.

Load Balancer: Load Balancer service type extends the Node Port by deploying an external Load Balancer instance.

→ will only work if you are in an env. that has an API to configure a Load Balancer instance like GCP
AWS
Azure

__/__/____

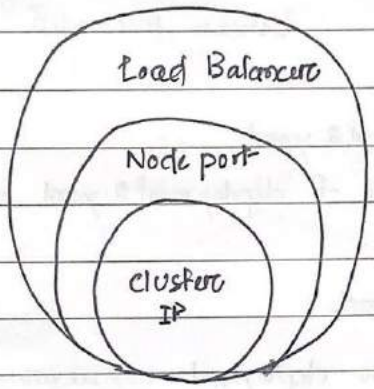
External Name: a special service type that has no routing

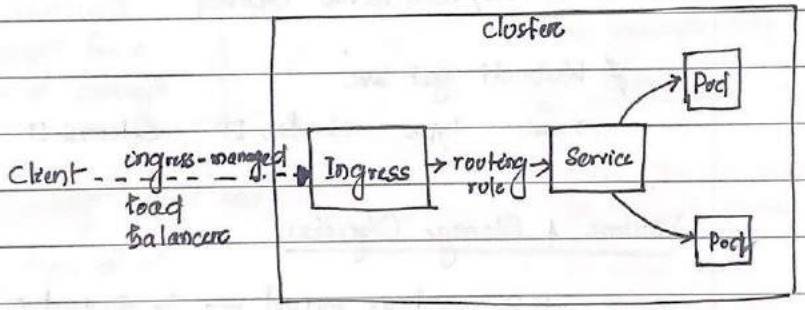
→ Ext. Name —uses (k8s internal DNS server)→ create a DNS alias

used to resolve rather a complicated host name

e.g.

my-cool-database-az1-uecj123.cloud-provider-example.com





* Ingress obj: provides → a means to expose HTTP and HTTPS routes from outside of the cluster for a service within the cluster.

↓

configuring routing rules that a user can set and implement with an Ingress controller.

↓

Features: load balancing, TLS offloading / termin
name-based virtual hosting, path based routing

__/__/____

- K8s provide a cluster external firewall with the Network Policy concept.

  ↓

  simple IP firewall (OSI layer 3 or 4) that can control traffic based on rules.

  e.g A typical use-case for Network Policies could be restricting the traffic between two diff. namespaces.

$ vim deployment2.yaml

$ kubectl create -f deployment2.yaml → deployment.apps /echoserver created

↓

03 pods

$ kubectl expose deployment echoserver --port=8080

⇓

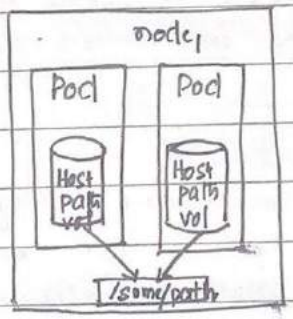
service/echo-server exposed

$ kubectl get svc

name type cluster-IP external-IP port(s) age

- Volume & Storage Objects:

K8s → volume mount → as a part of a pod



↳ sharing data between multiple containers within the same pod.

Prevents data loss when a pod crashes and is restarted on the same node.

__/__/____

Persistent storage:

cloud block storage - Amazon EBS
Google Persistent Disks
Azure disk storage

storage systems - Ceph
GlusterFS
NFS

K8s → using → container storage interface (CSI) → allows → Storage vendors (write a plugin)

Storage drivers

↓ used in K8s.

02 storage objects

* PVCs consume PV resources.
* PVCs are requests for PVs.

Persistent Volumes (PV)

* Provide APIs for users & admins to manage and consume persistent storage
* Provisioned by admin. or dynamically provisioned by storage classes
* volume plugins like volumes
* resources in the cluster

an abstract description for a slice of storage

obj. config holds info.
- type of vol.
- vol. size
- access mode and unique identifiers

Persistent Volume Claim (PVC)

If the cluster has multiple PVs,

create → PVC

* PVC can be used to request storage.
* 03 access modes
  - ReadWriteOnce
  - ReadOnlyMany
  - ReadWriteMany

Rook: Open-source cloud-native storage orchestrator for K8s,
Providing the platform, framework, and support for a diverse set of storage solns

↓

automates the task of a storage admin.

__/__/____

- Configuration Objects:

App. ---→--- config files, conn^n to other services, DB sys., storage

↓ config^n

(conn^n strings)

Bad practice: Incorporate the config directly into the container build

↓

any config change ---→--- entire image need to be rebuilt and the entire container or pod to be redeployed.

↓

solved ——K8s——→ decoupling the config from the pods with a config-map.

↓ API object

store config-files or variables as key-value pairs.

use in 2 ways

- mount a config-map as a volume in pod
- map variables from a config-map to env. variables of a pod

* Pods can consume config-maps as env. variables, command line args or as config^n files in a volume.
* Config-maps allow you to decouple env. specific info^n from your container images.
* data stored in a CM cannot exceed 1 MiB.
* Usage: configure settings for containers running in a pod in the same namespace.

* Opaque Secret: arbitrary user-defined data
default Secret type

* Kustomize provides resource Generators to create secrets and/configmaps.
↳ you don't need to include confidential data in your app's code.

- Secrets: K8s objects to store sensitive info like pods, token or a key
↳ base64 encoded

Cloud-native env. Secret mgmt. tools have emerged.

⇓

(HashiCorp Vault)

* Stored unencrypted in etcd. * Bootstrap token → Automate
Secrets node regist°

- Autoscaling Objects:

To scale the workload in a K8s cluster.

03 autoscaling mech°s

Horizontal Pod autoscaler (HPA)

* HPA means that the response to increased load is to deploy more pods.

- most used autoscaler
- HPA can watch deployments or replicasets and can increase the no. of replicas if a certain threshold is reached.

⇓

install° of an add-on server called metrics server.

↓

replace metrics server with Prometheus adapter for Kubernetes Metrics APIs.

↳ use custom metrics

cluster autoscaler

⇓

can add new worker nodes to the cluster if the demand increases.

* When your cluster runs low on resources, the CA provisions a new worker node and adds it to the cluster.

* Unregularities in the cluster load

Vertical Pod Autoscaler

↓

allocas pods to increase the resource requests and limits dynamically.

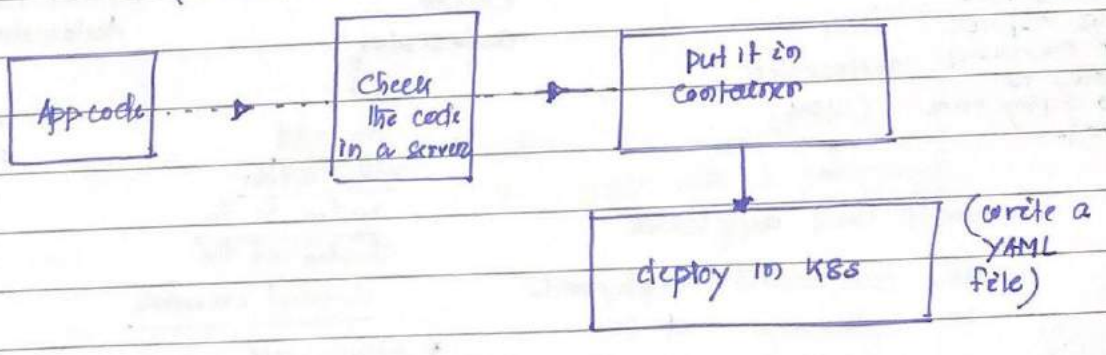
* VPA is useful when you can't create more copies of your pods, but you still need to handle more traffic.

__/__/____

- KEDA: (Kubernetes-based Event Driven Autoscaler)
  - Proj. KEDA can be used to scale the K8s workload based on events triggered by external systems.
  - 2019 (Microsoft - RedHat partnership)

* K8s objects can be described in a data serialization lang. called YAML.
* Main difference between ConfigMaps and Secrets
  - Base64 encoding

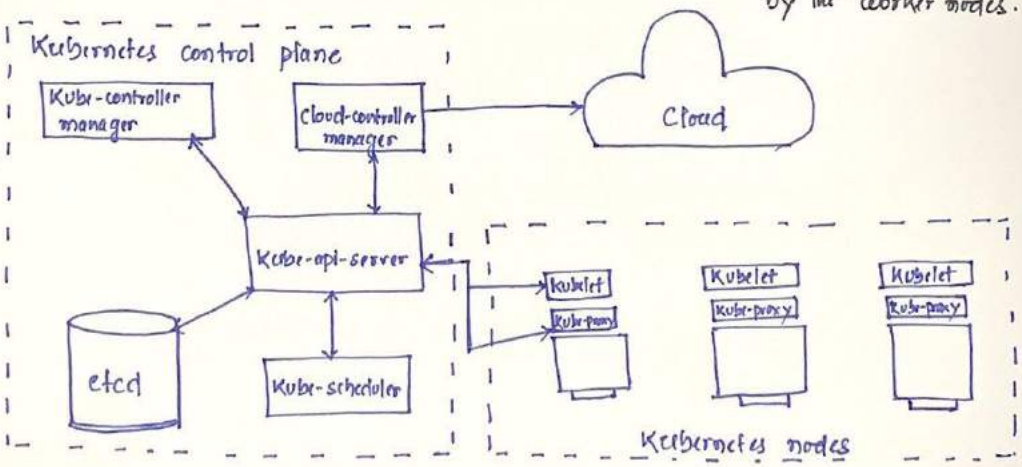
(Cloud-Native Application Delivery)



- Application Delivery Fundamentals:
  * Git - standard version control system
    - cheap local branching
      staging areas
      multiple workflows
  * IaC (Infra as code) → process of managing and provisioning data centers through machine readable def^n files. rather than phy. H/W config^n.
    - Ansible Autom^n Platform
      - Playbooks are used to describe the desired state of your Infra.
    - config^n/network policies/security (can be described as code)

KUBERNETES FUNDAMENTALS

- Kubernetes : container orch^n platform
  - ↳ automate deployments, scaling and mgmt. of containerized workloads
- Kubernetes : developed by Google in 2013
  - ↳ written in Go language
  - highly inspired by Google Borg system
  - ↓ donated to CNCF in 2014 (1st proj.)
  - current version of Kubernetes - 1.23
- Architecture: Kubernetes clusters consist of 02 different server nodes.
  - 1) Control Plane (nodes) (manage the cluster and the worker nodes)
    - * brain of oper^n
    - * contain various components which manage the cluster and control various tasks like deployment scheduling and self-healing of containerized workloads.
    - * one or more master nodes as part of the control plane
  - 2) Worker nodes
    - * appl^{n}s run in your cluster.
    - * only job of worker nodes (running appl^{n}s)
    - * Network traffic between client users and the containerized appl^{n}s deployed in pods is handled directly by the worker nodes.



* Kube-controller manager runs controllers responsible to act when nodes become unavailable, to ensure pod counts are as expected.

* Control-plane nodes typically host the following services:
  1) Kube-api-server: → centerpiece of Kubernetes
     → all other components contract with the api server and it is where users should access the cluster.

2) etcd : database which holds the state of the cluster.
↳ stand alone proj. and not an official part of Kubernetes.

3) Kube-scheduler : When a new workload should be scheduled, the kube-scheduler chooses a worker node that could fit based on CPU and memory.

4) Kube-controller-manager : contains many non-terminating control loops that manage the state of the cluster.

5) Cloud-controller-manager : can be used to interact with the API of the CSPs to create external resources like
- load balancers
- storage or
- security groups

* Components of worker nodes :

1) Container runtime : responsible for running containers on the worker node.
e.g. Docker → popular choice
↓
replaced by containerd

2) Kubelet : a small agent that runs on every worker node in the cluster.
→ receives pod defns from the API server, and interacts with the container run time on the node to run containers associated with the pod.
→ monitors the health and resources of pods running containers.
→ Kubelet talks to the kube-api-server and the container run-time to handle the final stage of starting containers.

3) Kube-proxy : a network proxy that handles inside and outside comm" of your cluster.
* abstracts the details of pod networking and forwards comm" reqs to pods.
→ relies on the networking capabilities of the underlying OS for managing traffic flow.
→ responsible for dynamic updates and maintenance of all networking rules on the node.

* Control Plane :
1) If not available → appl"s that are already started on a worker node will continue to run.
→ scaling, scheduling new appl"s etc : (not possible)

* NS provides a mech. for isolating groups of resources within a single cluster.
* NS → way to divide cluster resources between multiple users

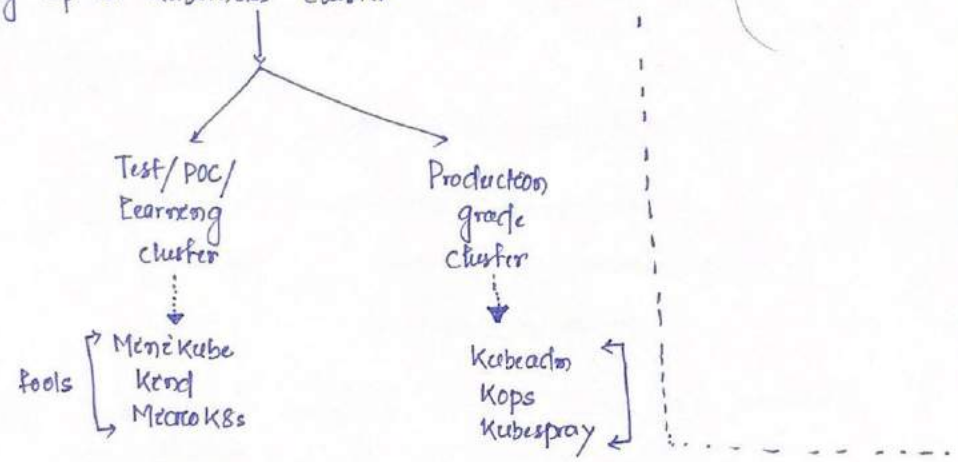
* Kubernetes namespace: can be used to divide a cluster into multiple virtual clusters, which can be used for multi-tenancy when multiple teams share a cluster.

↓

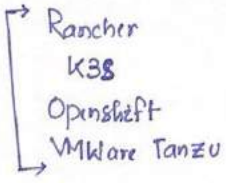
Viewed like a directory on a comp. where you can organize objects and manage which user has access to which folder.

- Kubernetes Setup: - - - - - - - - - - - - - - - -

* Setting up a Kubernetes cluster



* Vendors started packaging Kubernetes into a distn and offer commercial support.



* CSPs:
  - Amazon: (EKS) → Elastic Kubernetes service
  - Google: (GKE) → Google Kubernetes Engine
  - Microsoft: (AKS) → Azure Kubernetes service
  - Digital Ocean: (DOKS) → Digital Ocean's managed Kubernetes service

- Demo: Create 02 node Kubernetes cluster with → Kubeadm

  Pre-requisites:

  1) Linux Host 2) 2 GB or more of RAM per machine

3) 2 CPUs or more

4) Full network connectivity between all machines in the cluster

* Installation of Container RunTime:

→ To run containers in Pods, Kubernetes uses a container runtime.

→ By default, Kubernetes uses a container runtime interface (CRI) to interface with your chosen runtime.

→ If you don't specify a runtime, kubeadm automatically tries to detect an installed container runtime by scanning through a list of well known Unix domain sockets.

RunTime	Path to Unix domain Socket
Docker	/var/run/dockershim.sock
Containerd	/run/containerd/containerd.sock
CRI-O	/var/run/crio/crio.sock.

→ Docker takes precedence if both Docker and containerd are detected.

* Installation of packages in all the machines.

→ kubeadm : command to bootstrap the cluster (getting the initial cluster up and running)

→ kubelet : primary node agent that runs on each node (starts pods and containers)

→ kubectl : Kubernetes command line tool → allows you to run commands against Kubernetes clusters.

* Initialize control plane node : sudo kubeadm init (root privileges)

* installation of a pod-network : install Calico networking and network policy for on-premises deployments

↓

install a overlay network.

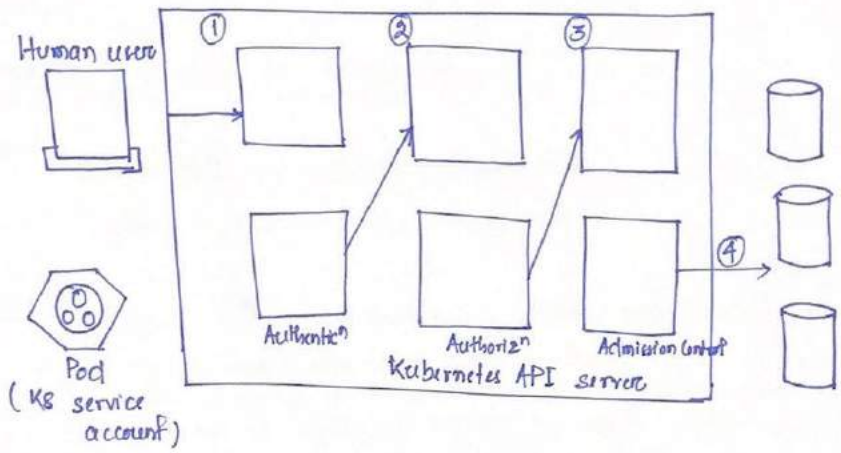
* sudo kubeadm join command → join a node to the cluster

* kubectl get nodes

name	status	roles	age	version
cp	...	...	...	...
worker node	...	...	...	...

* Kubectl cluster-info : info about cluster

- Kubernetes API:



* Comm^n with the cluster → Kubernetes API server

* Before a req^s is processed by the Kubernetes, it has to go through 03 stages:

i) Auth^n ii) Authoriz^n iii) Admission Control

(Normal users)

→ requestor (means of identity) → API

↓

Digitally signed cert. (X.509) or ext. identity mgmt. sys

→ Kubernetes → externally mged. users

→ Service accts → authenticate → Tech. users allocd in-cluster processes to communicate with API server to perform ops.

RBAC

1) Subjects: set of users & processes that want to access K8s API

2) Resources: pods, deployments, services, nodes, persistent volumes

3) Verbs: create, read, update, delete | CRUD operns on resources

→ Admission controllers can be used to modify or validate the request.

→ can block a request if a user tries to use a container image from an untrustworthy registry.

→ Tools like the Open Policy agent can be used to manage admission control externally.

→ used to specify granular access control policies, e.g allowing privileged containers, resource quotas, default storage class etc.

* Kubernetes API is implemented as a RESTful interface that is exposed over HTTPS.

(Representational State Transfer)

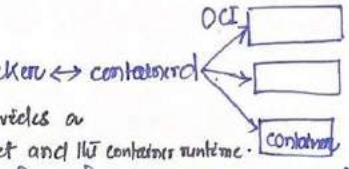
→ Architectural style for an API that uses HTTP requests to access and use data.

→ also known as REST API.

→ most common method for connecting components in micro-services architectures.

- Running Containers on Kubernetes:

* define Pods ⟶ smallest compute unit
  translate Pods ⟶ into running containers.



* Kubelet uses a shim app^n which provides a clear abstraction layer between Kubelet and the container runtime.

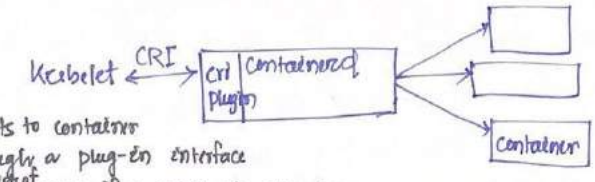
containerd → lightweight and performant impl^n to run containers.

↓

most popular container runtime

CRI-O → created by RedHat with a similar codebase as podman and buildah

Docker → usage of Docker as the runtime for Kubernetes has deprecated. and will be removed in Kubernetes 1.23.



* CRI implements 02 services.
  Image Service → image related ops
  Runtime Service → pod & container related ops.

* Kubelet connects to container runtimes through a plug-in interface (CRI). CRI consists of protocol buffers, gRPC, libraries etc.

* containerd/CRI-O : provides a runtime that only contains the absolutely essentials to run containers.

* Frakti CRI shim ⟶ aimed at enabling Kubelet to interact with kata containers.

* containerd / CRI-O —integrate→ with container runtime sandboxing tools

→ Solve the security problem that comes with sharing the kernel between multiple containers.

Tools: 1) gVisor 2) Kata containers

- gVisor → Provides an appln kernel that sits between the containerized process and the host kernel.
- Kata containers → a secure runtime that provides a lightweight VM, but behaves like a container.

- Networking: * A container runtime creates an isolated network space for each container it starts.

* ← 1) container-to-container comm^{n} → pod concept [On Linux, isolated network space → network namespace]

→ When a pod is started, a special Pause container is instantiated by the container runtime to create a network namespace for the pod.

2) pod-to-pod comm^{n} → overlay network

3) pod-to-service comm^{n} → Kube-proxy and packet filter on the node

4) external-to-service comm^{n} → (- do - .)

→ K8s network model treats pods as VMs on a network, where each VM is equipped with a network interface - thus each pod receiving a unique IP addr. This model is called "IP-per-pod" and ensures pod-to-pod comm^{n}.

(network proxy that runs on each node in your cluster)

* Impln of networking in Kubernetes

⇓

03 imp. requirements

1) all pods can communicate with each other across nodes.
2) all nodes can communicate with all pods.
3) no network address translation (NAT)

* K8s enables ext. accessibility through Services, complex ext. rule defns stored in iptables on cluster nodes and implemented by kube-proxy agents.

* To implement networking, you can choose from a variety of network vendors:

1) Project Calico 2) Weave 3) Cilium

* Every pod ⟶ own IP address ⟶ no manual config. involved.

Kubernetes set-ups ⟶ include ⟶ DNS server add-on called **core-dns**

↓

provide service discovery and name resolution inside the cluster.

* serve as the K8s cluster DNS.

- **Scheduling:**

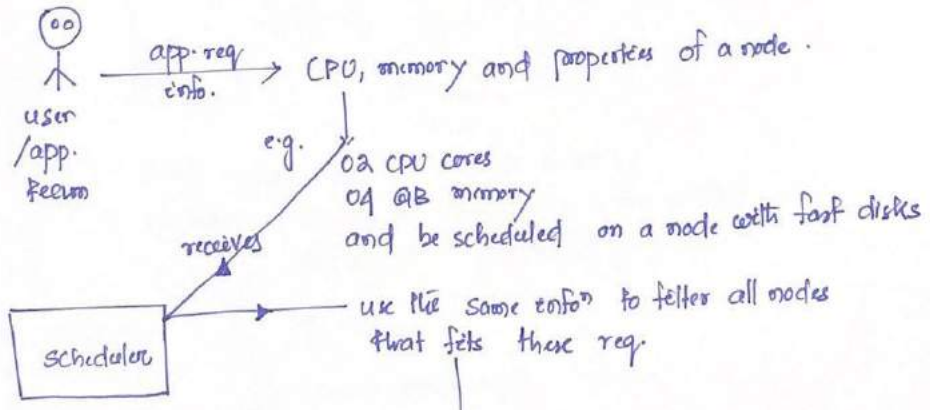
* container orch^{n} sub-category (Scheduling) → process of automatically choosing the right (worker) node to run a containerized workload.

* Kubernetes: kube-scheduler ⟶ scheduling daemon

* scheduling process starts ⟶ when a new pod is created.

* Kubernetes uses ⟶ declarative approach
  - pod is only described first.
  - scheduler selects a node where pod will actually get started by the kubelet and the container run time.

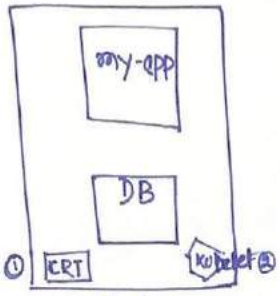
*



↓

Kubernetes will schedule the pod on the node with the least amount of pods.

Kubernetes Architecture



(02 apps running inside it)
e.g. my-app & DB

→ Each node has multiple pods on it.
→ 03 processes must be installed on each node
→ Worker nodes — computation.

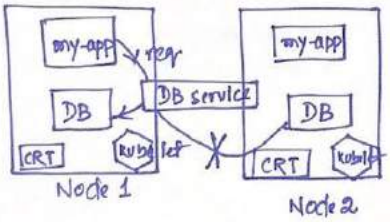
* It's a container engine, basically a S/W component that can run containers on host OS.

1) container run time
2) Kubelet (node agent)
3) Kube-proxy. (proxy)

* Setting up namespaces and cgroups for containers
* Docker, CRI-O, containerd, frakti (hypervisor based container runtime for k8s)

* Appln pods have containers running inside it.
* Container run time needs to be installed on every node.
* Kubelet interacts with the container run time and the node.
  ↓
  takes the config and starts the pod with a container inside. assigning resources from the node to the container.
  ↓
  (CPU, memory, storage resources)

*

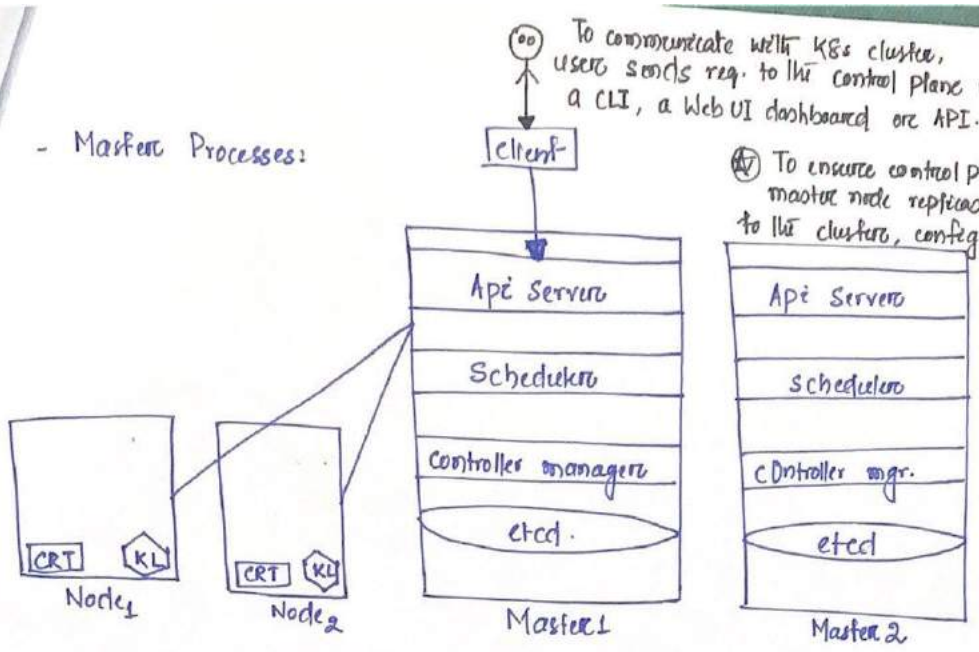


→ Comm" between pods is through services.
→ Kubeproxy: intelligent routing logic inside that ensures comm" works in a performant way with low overhead.

- Schedule pod
  Monitoring pods
  Re-schedule/re-start pods
  ↓
  → Master node (managing processes)

- Master Processes:

To communicate with K8s cluster, user sends req. to the control plane via a CLI, a Web UI dashboard or API.



(IV) To ensure control plane's fault-tolerance, master node replicas can be added to the cluster, configuring HA.

04 processes run on every master node (run the cluster state and the worker nodes)

1) Api Server:

* user → wants to deploy a new appln in a K8s cluster, you interact with Api Server using some client like UI/K8s dashboard or command line tool like Kubelet.
* Api server → cluster gateway which gets the control req., may be updates to the cluster, or the queries from the cluster.
* gatekeeper for authn
* To schedule new pods or deploy new applns, api server receives the req, validates it and feeds it to other processes.
* only 01 entrypoint to the cluster
* Api Server — load balanced
* Api Server → only master plane component to talk to the etcd store.

2) Scheduler

* intelligent way of deciding which worker node can handle the running of the pod.
* scheduler ⇒ decn making.
* process that does the scheduling is the Kubelet which starts the pod.

3) controller mgr.

* detects the state changes of the cluster
* if pods die, reschedules the pod and restarts them.
* regulate the state of the K8s cluster.
* controllers are watch-loops continuously running and comparing the cluster's desired state with its current state.

4) etcd

* Key-value store of a cluster state
* cluster brain
* cluster state → saved
* resources availability cluster health → stored → etcd
* appln data → not stored in etcd.
* etcd ⇒ distributed storage across all cluster nodes.
* Both stacked and external etcd configns support HA configns.
* etcd is based on the Raft consensus algorithm.

- Add new Master/Node Servers:

1) Get the bare metal server
2) install the master/worker nodes processes
3) add it to the cluster

- Kubernetes Security Tools:

↑ Popeye
Kube audit
Kube-bench

---

* Kubernetes Features:

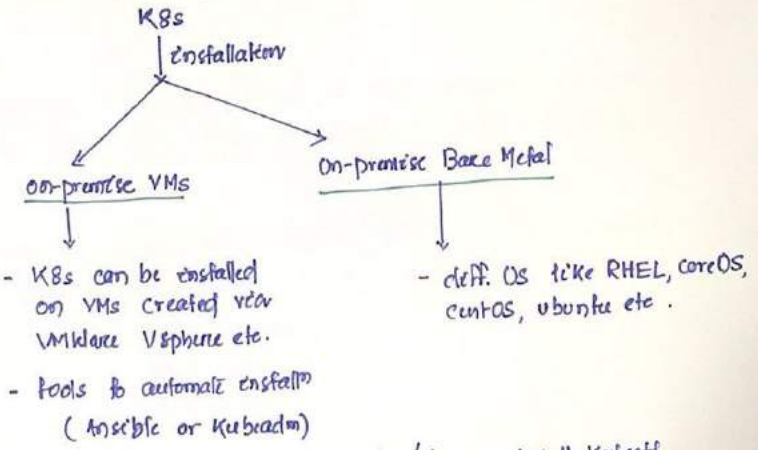
1) Automatic bin packing → automatically schedules containers
2) Self-healing → automatically replaces and reschedules containers from failed nodes.
3) Horizontal scaling → Based on CPU and custom metrics
4) Service discovery and load balancing
   ⇓
   Containers receive their IP address from Kubernetes.
   ⇓
   assigns a single DNS name to a set of containers to aid in load balancing requests across the containers of the set.
5) automated rollouts and rollbacks
6) Secrets and config mgmt (K8s manages sensitive data and config details for an app separately from the container image, in order to avoid a re-build of the respective image.)
7) Storage orchestration

Kubernetes Configuration:

* Master installation types are
  - → all in one single node (learning, testing)
  - → single master and multiple worker (master running a stacked etcd instance)
  - → single master with single node etcd, and multiple worker (external etcd instance)
  - → multi-master and multi-worker (HA)
  - → multi-master with multi-node etcd and multi-workers (external etcd instance) (prodn env.)

* Installation tools to deploy single node or multi-node K8s cluster
  - → MiniKube - easiest and preferred to set up K8s locally.
  - → Kind
  - → Docker Desktop
  - → MicroK8s
  - → K3S - lightweight K8s cluster from Rancher

*



* apt - package manager (native pkg. mgmt) / brew - Install Kubectl on macOS
* Cloud Environment:
  1) Hosted Solutions
  2) Turnkey Cloud Solutions (install K8s on Iaas platform)
     - AWS EC2, GCE, Alibaba Cloud
  3) Turnkey On-premise solutions
     - Open Shift Container Platform

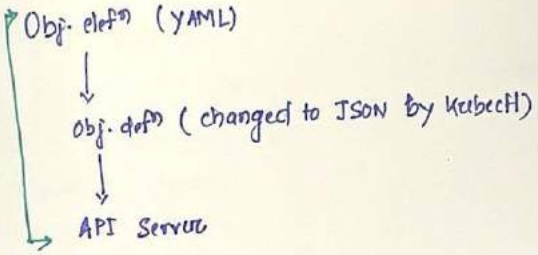
* Kubeadm: Secure and recommended method to bootstrap a multi-node prodn ready HA K8s cluster

* Kubespray: (formerly known as Kargo) - can install HA prodn ready clusters on CSPs or Bare Metal.

* Kops: Create, upgrade and maintain prodn grade K8s clusters from the command line.

Kubernetes Object Model:

* spec section: declare the desired state of the object.
* status section: K8s system manages the status section for objects. Records the actual state of the obj.
* Objects: Pods, ReplicaSets, Deployments, Namespaces
*



* Pod:
  - → The containers in a pod share the same network namespace, meaning that they share a single IP add. originally assigned to the pod.
  - → ephemeral in nature, and they do not have the capability to self-heal themselves. So, they are used with controllers which handle Pod's replication, fault tolerance, self-healing
  - → e.g (Controllers): Deployments, ReplicaSets, Replication Controllers

* Labels:
  - → Key-value pairs attached to K8s objects
  - → used to organize and select a subset of obj.
  - → do not provide uniqueness to obj.
  - → Controllers use labels to logically group together decoupled obj., rather than using objects' names or IDs.

* Default controller → Deployment which configures a ReplicaSet controller to manage Pod's lifecycle.

* With the help of a ReplicaSet, we can scale the number of pods running a specific application container image.

* Deployment:
  - → provides declarative updates to pods and replicasets.
  - → ensures that the current state always matches the desired state.
  - → directly manages its ReplicaSets for application scaling.

* Namespaces:
  - → Partition the K8s cluster into virtual sub-clusters called namespaces
  - → $ kubectl get namespaces : list all namespaces

04 namespaces created by K8s

NAME	STATUS	AGE
default	Active	11h
kube-public	Active	11h
kube-system	Active	11h
kube-node-lease	Active	11h

  - → Resource Quotas: Help users limit the overall resources consumed within Namespaces
  - → LimitRanges: Help limit the resources consumed by pods or containers in a Namespace

Authentication, Authorization, Admission Control:

* Authentication:
  - → K8s uses a series of authentication modules.
    1. X509 client certificate
    2. Static token file
    3. bootstrap tokens
    4. Service account tokens
    5. OpenID connect tokens
    6. Webhook Token Auth
    7. Authenticating Proxy

* Authorization:

→ After a successful authentication, users can send the API req. to perform diff. oper^{n}s.

→ Authorization modes:

Node → authorizes API req. made by kubelets

Attribute-based access control (ABAC)

Webhook – K8s can request authoriz^{n} decisions to be made by 3rd-party services

RBAC – a) Role → access to resources within a specific Namespace
b) ClusterRole → cluster-wide access

Services:

* offers a single DNS entry for a containerized appl^{n} managed by the K8s cluster, regardless of the no. of replicas, by providing a common load balancing access point to a set of pods logically grouped and managed by a controller such as a Deployment, Replicaset or DaemonSet.

* Service: logically group pods and defines a policy of any user or appl^{n} wants to access them. This grouping is achieved by Labels and Selectors. This service name is registered with the cluster's external DNS service.

* User/client —connects to the→ service via its ClusterIP
↓
forward traffic to one of the pods attached to it

* Kube-proxy is responsible for implementing the service config^{n} in order to enable traffic routing to an exposed appl^{n} running in pods.

* As services are the primary mode of comm^{n} between containerized appl^{n}s managed by K8s, K8s supports 02 methods for discovering services.

i) Environment Variables: As soon as the pod starts on any worker node, the kubelet daemon running on that node adds a set of env. variables in the pod for all active services.

ii) DNS: DNS record for each service.

* Access scope of the service is decided by the ServiceType property.

* ClusterIP - default ServiceType
  ↳ virtual IP addr. received by a service.
  ↓
  used for communicating with the service and is accessible only from within the cluster.

* NodePort ServiceType → High-port picked up from the range (30000 – 32767) is mapped to the respective service from all the worker nodes.
  ↓
  useful when we want to make our services accessible from the external world.

* LoadBalancer ServiceType: Nodeport, ClusterIP (→ automatically created)
  ↓
  will only work if the underlying infra. supports the automatic creation of loadbalancers.

* A Service mapped → ExternalIP address if it can route to one or more of the worker nodes.

  ExternalIP address ——→ not managed by K8s.

* ExternalName : special ServiceType that has no selectors and does not define any end points.
  ↓
  make externally configured services available to apps inside the cluster.

Kubernetes Volume Management:

* containers ——→ ephemeral in nature. (all data inside a container is deleted if the container crashes).
  ⇓
  Volumes (storage abstractions)

* Volume - mount point on the container's file system.

* VolumeType - decides the properties of the directory, like size, content, access modes etc.

  e.g. emptyDir
  hostpath
  nfs
  ✓ Secret
  ✓ configmap
  ✓ persistentVolumeClaim

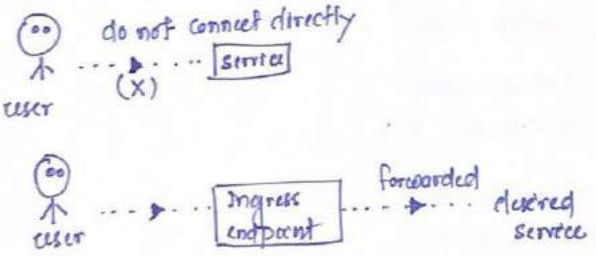
* Container Storage Interface (CSI) – designed to work on diff. container orchestrators

ConfigMaps and Secrets:

* ConfigMaps allow us to decouple the config details from the container image.
* Using ConfigMaps → pass config. data as key-value pair → Pods/sys. components → in the form of env. variables
* Secrets → share sensitive info (passwords, tokens, keys)

Ingress:

* Coll^n of rules that allow inbound comm^s to reach the cluster services.
  ↓
  configures a Layer 7 HTTP/HTTPS load Balancer for services
* With ingress,



* Ingress controller → appl^n watching API server for changes in the Ingress resources and updates the Layer 7 load balancer.
  ↓
  (K8s supports AWS, GCE, and NGINX Ingress controllers)

Quota and Limits Management:

* Following typs of quota per namespace.
  1) Compute Resource Quota
  2) Storage Resource Quota
  3) Object Count Quota

Kubernetes Cluster Federation:

* manage multiple K8s clusters from a single control plane.

Security Contexts:

* Set discretionary access control for obj. access permissions
* defines privilege and access control settings for a pod or container

Pod security policies: apply security settings to multiple pods and containers cluster wide.

Network Policies:

* Set of rules which define how pods are allowed to talk to other pods and resources inside and outside the cluster.
* namespaced API resources

Deployments

* Stateless appl^ns
* Pod has random hash at the end.
* pods deployed by Deployment are identical.

Stateful Sets

* Stateful appl^ns (databases)
* Pods deployed by stateful set are not identical.
* maintains
* Sticky identity for each pod
  - (fixed ordered names)
  - persistent identifier across any rescheduling

When a pod dies and replaced by a new one, it maintains its identity.

* all pods are not identical.
  - one — master
  - rest — slaves
* data persistence → recommended stateful appl^ns.
  - If pods die, all data is gone.
  - data will survive, if pods die.
    (Persistent Volume) → remote storage

- mysql-0
- mysql-1
- mysql-2

* fixed ordered DNS name

Objects:

* pods
  namespaces
  replicasets
  deployments
  statefulsets
  daemonsets
  Services
  configmaps
  volumes

* To display all k8s obj.s → `kubectl api-resources`

NAME	SHORTNAMES	KIND	APIGROUP	NAMESPACED
bindings		Binding		true
configmaps	cm	ConfigMap		true

* To check k8s cluster's version → `kubectl version`

```
client version:
server version:
```
(k8s cluster version)

Interacting with k8s:

* To check the list of available API resources → kubectl api-resources
* To check the list of available API groups and their versions → kubectl api-versions

```
admissionregistration.k8s.io/v1
admissionregistration.k8s.io/v1beta1
:
```

* To list all pods in the namespace → kubectl get pods
* To list all pods in all namespaces → kubectl get pods --all-namespaces
* To list all pods in the current namespace with more details → kubectl get pods -o wide

* Tools for testing k8s clusters and appln's:
  1) K8s dashboard
  2) Kube-hunter
  3) Project Quay
  4) Kube-burner
  5) Kube-bench
  6) Kube-monkey

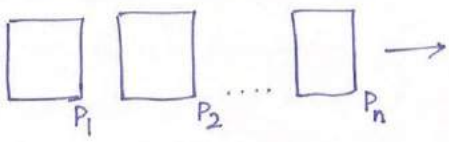
Kubectl:

* Run interactive containers in your cluster → kubectl run
* kubectl logs -f
  ↓
  live stream new logs to your terminal
* cockpit to control K8s
* client for the K8s API

* supports JSONPath template.
  ⇓
  {JSONPath expression}

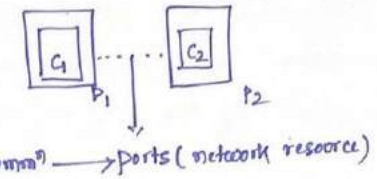
Pod Concept:

* A pod contains one or more appln containers which are relatively tightly coupled
* A pod can contain init containers that run during pod start up.
* Shared context of a pod is a set of Linux namespaces and cgroups.
* Container image - nginx:1.14.2
  obj. spec^{n} & status - simple-pod.yaml
  Create the pod: kubectl apply -f https://k8s.io/examples/pods/simple-pod.yaml
* Pods in a K8s cluster are used in 02 main ways:
  1) pods that run a single container (Pod - wrapper around a single container)
  2) pods that run multiple containers that need to work together
* Pod → meant to run a single instance of a given appln



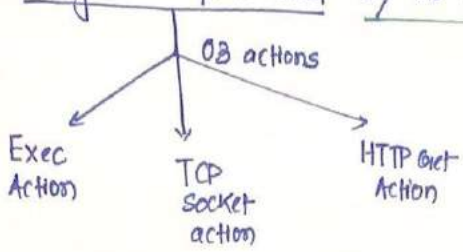
Scaling your appln horizontally (to provide more overall resources by running more instances), you should use multiple pods, one for each instance.
⇓
Replication (in K8s)

* Pods natively provide 02 kinds of shared resources for their constituent containers:
  i) networking
  ii) storage

* pod remains on the node until the pod finishes execution.
* A pod is not a process, but an env. for running containers.
* You can use workload resources to create and manage multiple pods for you.
  i) deployment
  ii) statefulset
  iii) daemonset
  * Pod templates → specifn for creating pods and are included in workload resources such as deployments, jobs, daemonsets
  * Pod update operations → patch → partially update the specified pod
    replace → replace the specified pod
  * Pod networking:
    → Pod assigned unique IP add.
    → Each container in a pod shares the same namespace including IP add. and network ports.
    → Inside a pod, the containers that belong to the pod can communicate with one another using localhost.
    → 

  * Static pods:
    → managed directly by the kubelet daemon on a specific node without the API server observing them
    → Kubelet directly supervises each static pod and restarts if it fails.
    → always bound to one Kubelet on a specific node.
    → main use: Run a self-hosted control plane ⇒ using kubelet to supervise the individual control plane components
    → specifn of a static pod cannot refer to other API objs

* Probe: diagnostic performed by the kubelet on the container



* Pod evicted from the node due to lack of resources or, if the node fails.
* Kubectl get pod → inspect a pod running on your cluster

A pod can be in one of the following phases.

1) Pending → Pod has been created and accepted by the cluster, but one of its containers are not yet running.
2) Running
3) Succeeded → all containers in the pod have terminated successfully.
4) Failed → all containers in the pod have failed.
5) Unknown → state of the pod cannot be determined.

Side-car Container:

* Dedicated sidecar container for logging in an appln pod
* Examples: Monitoring agents, data loaders
* Sidecar container must be placed on the same pod to use the same resources being used by the main container.
* sidecar containers do not provide a user-focussed service.
* run side-by-side with your main container.

Init Container:

* Examples: Create user accounts
  Perform DB migrations
  Create DB schemas
* Init containers are started and executed in sequence.
* If a pod's init container fails, the kubelet repeatedly restarts that init container until it succeeds.
* To specify an init container for a pod, add the initContainers field into the pod specfn.
* Init containers can contain utilities or custom code for setup that are not present in an app image.
* During pod startup, the kubelet delays running init containers until the networking and storage are ready.

* If the pod restarts or is restarted, all init containers must execute again.
* A Pod cannot be in Ready status until all init containers have succeeded.
* Init-containers can be given access to Secrets that app containers cannot access.
* provide an easy way to block or delay the startup of app containers until some set of preconditions are met.

Liveness Probe:

* Kubelet uses liveness probes to know when to restart a container.
* Apps running for long periods of time eventually transition to broken states, and cannot recover except by being restarted. K8s use liveness probe to detect and remedy such situations.

Kubectl describe pod liveness-exec
Kubectl get pod liveness-exec → RESTARTS has been incremented.

Workload Objects:

ReplicaSet:

* Purpose is to maintain a stable set of replica pods at any given time
* Identifies new pods to acquire by using its selector.
* A deployment is a higher-level concept that manages ReplicaSets and provides declarative updates to pods.
* Name of a ReplicaSet obj. must be a valid DNS subdomain name.
* A ReplicaSet can be auto-scaled by an HPA (Horizontal Pod Scalers).
* Deployments own and manage their ReplicaSets.
* When load becomes too much for the no. of existing instances, K8s enables you to easily scale up your appln, adding additional instances as needed.

Deployment:

* define Deployments to create new ReplicaSets
* Create the deployment

Kubectl apply -f https://k8s.io/examples/controllers/nginx-deployment.yaml

check whether the deployment was created

Kubectl get deployments

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	0/3	0	0	1s

↓
how many replicas of the appln are available to the users (ready/desired)

kubectl rollout status deployment/nginx-deployment

↳ waiting for rollout to finish : 2 out of 3 replicas have been updated.

kubectl get deployments

↳

NAME	READY	UP-TO-DATE	AVAILABLE	AGE
nginx-deployment	3/3	3	3	18s

kubectl get rs

NAME	DESIRED	CURRENT	READY	AGE
nginx-deploy ...-756....	3	3	3	18s

* Pod-template-hash label — added by → Deployment controller to every replica set that a deployment creates.

* scale a deployment:
  kubectl scale deployment/nginx-deployment --replicas=10

* Rolling update deployments support running multiple versions of an appln at the same time.

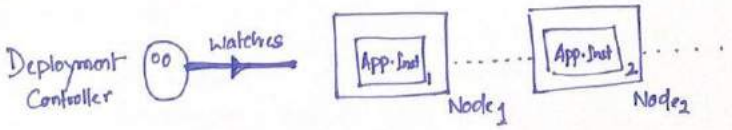
* Deployment status : → progressing
  complete
  fail to progress

* Canary deployment – roll out releases to a subset of users or servers using the deployment.

* Strategy → specifies the strategy to replace old pods by new ones.
  - Recreate
  - Rolling update (default)

* Deployment – creating and updating instances of your appln

* Deployment : ——→ API server ——→ Kube-scheduler



Stateful Sets:

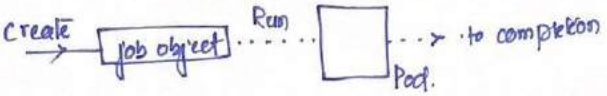
* Provides guarantees about the ordering and uniqueness of pods.
* Kind - Service

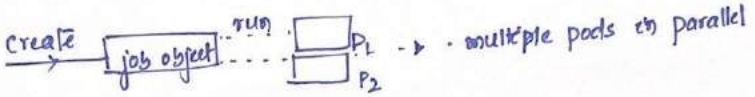
Daemonset:

* daemonset pods are created and scheduled by the Daemonset Controller.

Job:

* Creates one or more pods and will continue to retry execution of the pods until a specified no. of them successfully terminate. When a specified no. of successful run of pods is completed, then the job is considered complete.





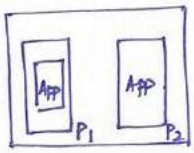
* Jobs are complementary to Replication Controllers.
* needs no pod selector
* To make your job fail, you need to exit with a code other than 0.

Cron Job:

* Timezone $\xrightarrow{\text{set}}$ Kube-controller mgr.
* Running backups or sending emails

Service:

* An abstract way to expose an appln running on a set of pods as a network service.



*In a Deployment, set of pods running in one moment could be diff. from the set of pods running that appln a moment later.

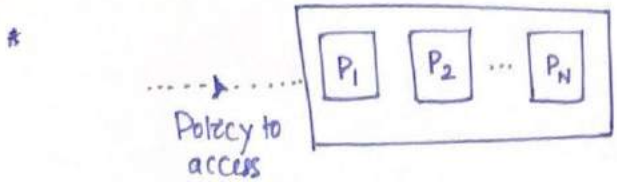
T_1 : P_1, P_x
T_2 : P_2, P_y

⇓

Problem

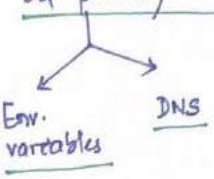
⇓

How do the front ends find out and keep track of which IP address to connect to so that the frontend can use the backend part of the workload?



Service → abstraction which defines a logical set of pods and a policy by which to access them.

* Set of pods targeted by a service is determined by a selector.
* K8s supports 02 primary modes of finding a service.



* Create a service:

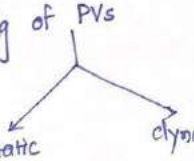
Kubectl expose deployment/my-nginx → service/my-nginx exposed

* Exposing a service onto an external IP add.

K8s supports it : 1) nodeports
2) loadbalancers

Volume & Storage objects:

* Provisioning of PVs



!! When none of the static PVs, the admin created match a user's PVC, then the cluster may try to dynamically provision a volume specially for the PVC. This provisioning is based on the StorageClasses.

* ClaimRef: bi-directional binding between PV and PVC
* PVs can have various reclaim policies:

1) Retain
2) Recycle
3) Delete (Default)

Autoscaling:

* HPA → automatically updates a workload resource (deployment/statefulset). with the aim of automatically scaling the workload to match demand.

→ HPA controls the scale of a Deployment and its Replicaset.

→ K8s implements HPA as a control loop that runs intermittently.

→ common use: Fetch metrics from aggregated APIs

→ While managing the scale of a group of replicas using the HPA, it is possible that the no. of replicas keep fluctuating frequently due to the dynamic nature of the metrics evaluated. (known as thrashing or flapping) → similar to hysteresis in cybernetics

Taints and Tolerations:

* Node Affinity is a property of pods that attracts them to a set of ~~pods~~ nodes
* Taints: allow a node to repel a set of pods.
* Tolerations: are applied to pods and allow the pods to schedule onto nodes with matching taints.

Kube-proxy:

* network proxy that runs on each node in your cluster, implementing part of the K8s service concept.
* maintains network rules on nodes. These network rules allow network comm^n to your pods from network sessions inside or outside of your cluster.
* is only used to reach services

Namespace:

* Kubectl api-resources --namespaced=true (to see K8s resources are there in a NS)
  Kubectl api-resources --namespaced=false
* K8s uses namespaces to organize objs in the cluster.

Kubeadm:

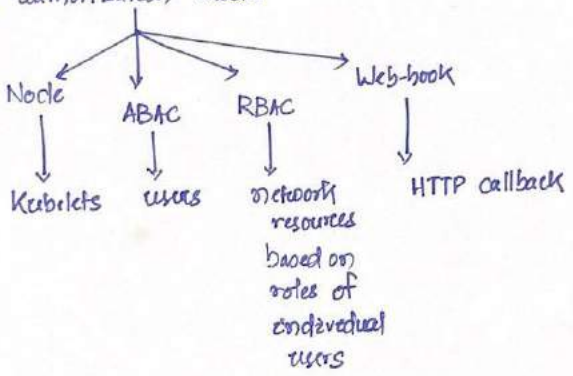
* tool built to provide Kubeadm init and Kubeadm join } for K8s clusters.
* By design, it cares about bootstrapping, not about provisioning machines.

Kops: easiest way to get prodn grade K8s cluster up and running, but also will provision necessary cloud infra.

Kubespray : deploy K8s clusters on multiple machines by combining Ansible and K8s.

Controlling Access to the K8s API:

* API serves on port 443, protected by TLS.
* API server serves HTTP on 02 ports.
    i) localhost port - 8080
    ii) Secure port - 6443
* Authentecation: → Service accounts are tied to a set of credentials stored as Secrets.
    → Bootstrap tokens are defined with a specific type of secrets that lives in the Kube-system NS.
* Authorization:
    → 04 authorization modes



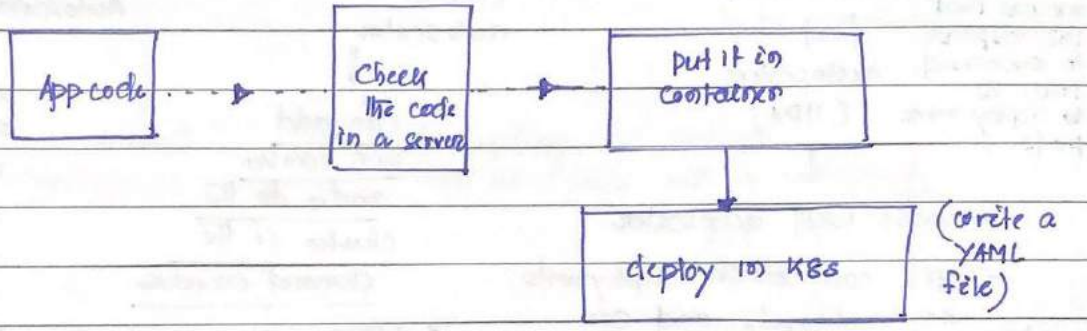
* Admission controller: piece of code that intercepts requests to the K8s API server prior to persistence of the obj.
* Container-to-container communication: Shared volume in a pod
  IPC namespace
* pod-to-pod comm^{n} : pods usually connect to other pods through a service.
  Network plug-in called Calico
* pod-to-service comm^{n} : Kube-proxy
* external-to-service comm^{n} : Ingress controllers

__/__/____

- KEDA: (Kubernetes-based Event Driven Autoscaler)
  ↓
  Proj. KEDA can be used to scale the k8s workload based on events triggered by external systems.
  ↳ 2019 (Microsoft - RedHat partnership)

* K8s objects can be described in a data serializ^{n} lang. called YAML.
* Main difference between ConfigMaps and Secrets
  ↓
  Base64 encoding

(Cloud-Native Application Delivery)



- Application Delivery Fundamentals:
  * Git - standard version control system
    ↳ cheap local branching
    staging areas
    multiple workflows

  * IaC (Infra. as code) → process of managing and provisioning data centers through machine readable def^{n} files. rather than phy. H/W config^{n}.
    ↓
    Ansible Autom^{n} Platform
    ↓
    Playbooks are used to describe the desired state of your infra. → config^{n}/network policies/security
    ↓ (can be described as code)

__/__/____

- CI/CD:

  CI: High automation and usage of version control

  CD: autom° of deployment process

  Popular CI/CD tools:

  - Spinnaker
  - GitLab
  - Jenkins
  - Jenkins X
  - Tekton CD
  - Argo

- GitOps:

  ↳ integrates the provisioning and change process of infra. with version control ops.

  02 approaches

  Push-based ⇓

  pepeline is started and run tools that make the changes in the platform.

  * K8s suited → GitOps (provides an API and is designed for declarative provisioning and change of resources right from the beginning)

  Pull-based ⇓

  - an agent watches the git repository for changes and compares the def° in the repo. with the actual running state. If changes are detected, the agent applies the changes to the infra.

  - Flux, ArgoCD ↳ 02 popular GitOps frameworks

  - ArgoCD (Kubernetes controller)

  - Flux (GitOps Toolkit)

CLOUD NATIVE APPLICATION DELIVERY

GitOps:

* Declarative config is the key to dealing with infra. at scale.
* way to K8s cluster management and appln delivery
* works by using Git as a single source of truth for declarative infra. and applns
* an operating model for K8s and other cloud native tech, providing a set of best practices that unify Git deployment, mgmt. and monitoring for containerized clusters and applns.

* Benefits:
  a) increased productivity → continuous deployment autom (deploy faster)
  b) enhanced developer exp. → push code and not containers
  c) improved stability → convenient audit log of all cluster changes outside of K8s
  d) higher reliability → Reduce your mean time to recovery (MTTR) from hours to mins (quick and easy recovery)
  e) developer centric → deployment by pull request
* GitOps uses Git pull requests to automatically manage infra. provisioning and deployment.

* GitOps → an evolution to IaC that uses Git as the version control sys. for infra. configs

* In GitOps workflow, changes are made using pull requests which modify state in Git repository.

* GitOps gives you tools and a framework to take DevOps practices and apply them to infra. autom and appln deployment.

* GitOps: Running operations out of Git | by pull requests with the goal of continuous delivery of cloud native appln and typically Kubernetes.

  4 Tech.:
  a) IaC (Ansible, Terraform etc.) — Ops
  b) Git (workflows) — Dev
  c) CI/CD
  d) convergent platforms (Kubernetes) — APIs, extensible

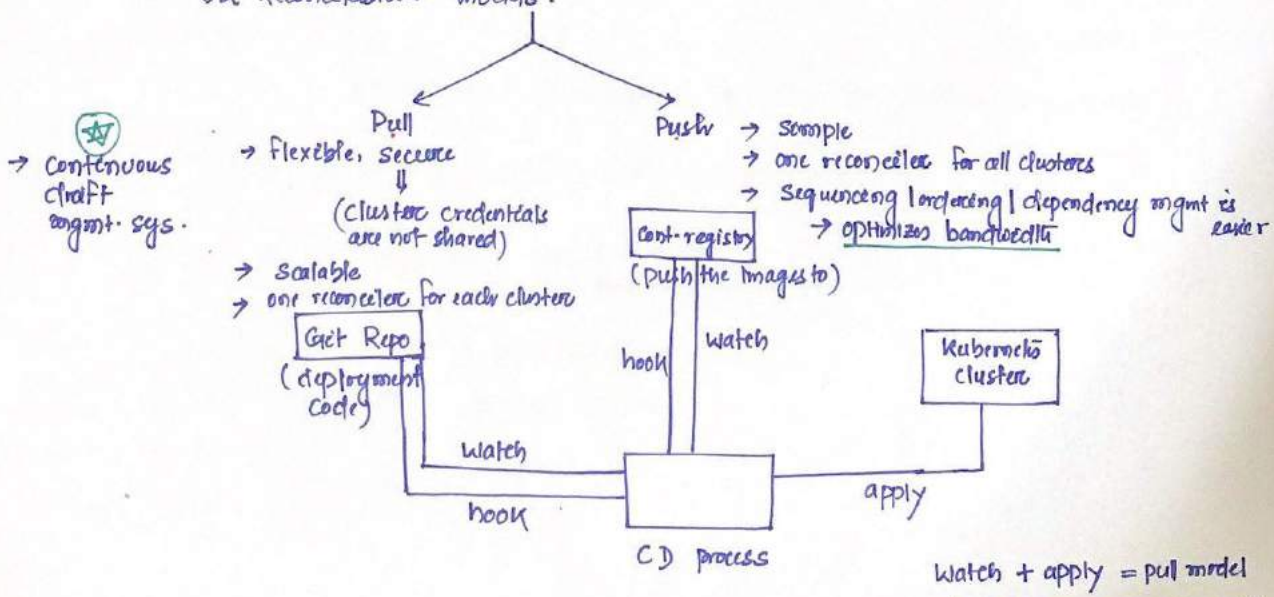
* Common usecases:

a) continuous delivery of appln configns (YAML manifest)
b) apply release strategies (Blue Green
Rolling update
Canary)
c) Infra. rollouts to K8s (ingress controllers
RBAC
network policies)
d) Disaster Recovery
e) Sync Secrets
f) Drift detection (identify and detect any changes in cluster config. settings)
g) deploy to multiple K8 clusters
h) securely Brandoff deployments to Engineering Teams.
i) auto update K8s YAMLs on new image in Registry

* principles and practices:

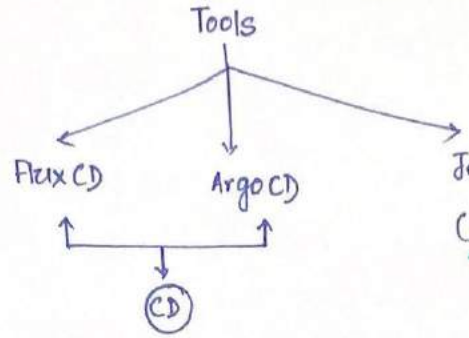
a) 100% configns → declarative code (YAML)
b) store desired state in Git
c) apply approved changes automatically
d) check and correct configns with a S/W agent (reconciler)

* D2 Reconciliation models:



watch + apply = pull model
hook + apply = push model

*



→ sync Git with K8s
→ support progressive releases
→ CNCF projs

Jenkins X
↓
(CI + CD) + Image building

→ K8s native
→ not a new version of Jenkins
→ uses Tekton (CI)
→ uses Scaffold and Kaniko to build container images

	Flux CD	Argo CD
Reconciliation model	Pull	Push
RBAC	Kubernetes	Its own
SSO	X	✓
Automatic image updates	✓	X
multi-cluster for all clusters	yes, one per cluster	yes, one
drift mgmt.	Continuous reconciliation	Event

→ Complex and Heavier (than Argo/Flux)
→ doesn't not support multitenancy

* Weaveworks → CNI plugin (weave)
↳
beyond CNI (CD custom) ··· ▸ · Flux ··▸ GitOps (coined in 2016)

2018/2019 CSPs ▸· managed deployment services.

2019: Argo Flux (decided to merge and ~~Flux~~ later changed it)

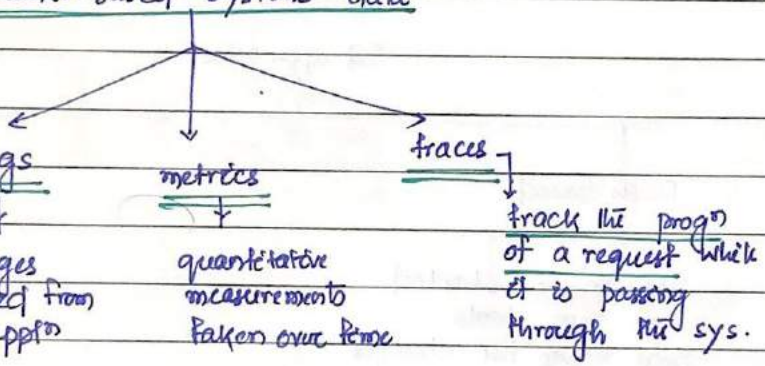
✓ Flux built → GitOps Toolkit
✓ Argo built → GitOps Engine

__/__/____

(Cloud Native Observability)

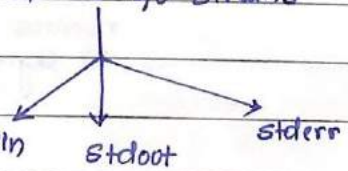
- Observability:
  * monitoring sub-category of observability
  * observability → closely related to the control theory which deals with behavior of dynamic systems.
  * You set the desired state of the sys. and trigger scaling events based on the load of the sys.
  * Goal: Allow analysis of the collected data.
    ↓
    better understanding of the sys. and react to error states

- Telemetry:
  * container-based systems data



- Logging: levels → debug, info, warning, error

  * Unix progs → provide 03 I/O streams



  * CLI tools: Docker, kubectl, podman → command → to show the logs

__/__/____

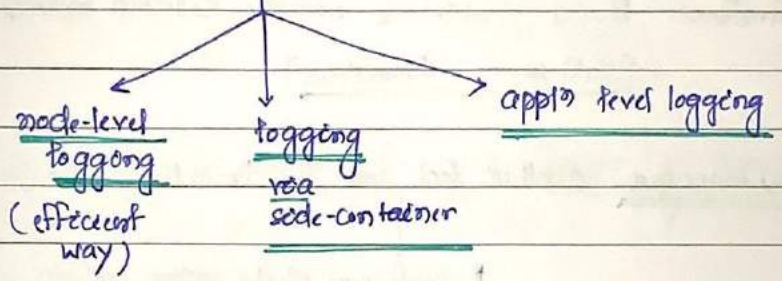
$ docker logs nginx → container "nginx" logs

$ kubectl logs nginx

$ kubectl logs -p -c ruby web-1 ⇒ return snapshot of previously terminated ruby container logs from pod web-1

$ kubectl logs -f -c ruby web-1 ⇒ stream logs on real-time (-f)

* Logs shipped System (log storage)

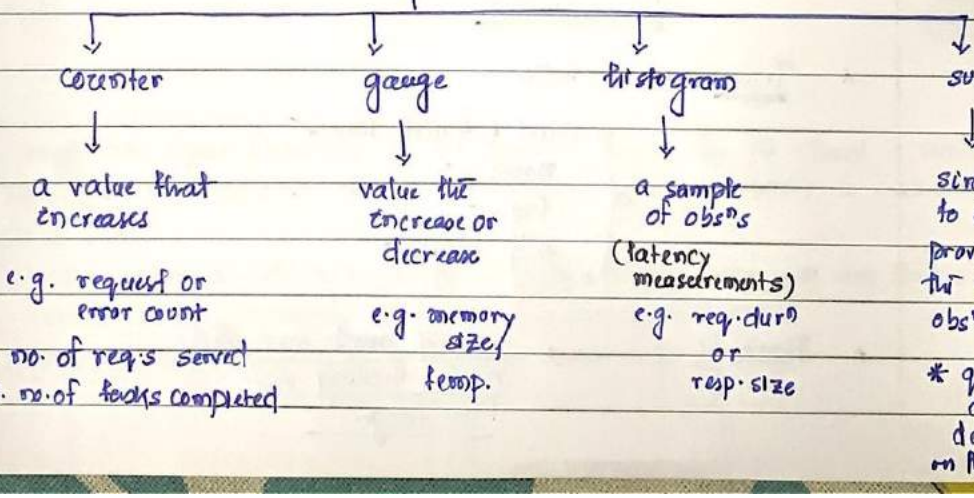


* Tools for shipping the logs: Fluentd / filebeat

* Log storage: OpenSearch / Grafana Loki

- Prometheus:

* Open-source monitoring sys. → 2nd CNCF hosted proj. in 2016.
* Collects metrics emitted by appln's and servers as time series data
* Prometheus data model provides 04 core metrics.



__/__/____

* Metrics exposure : [Apps] ..... expose HTTP endpoint under /metrics
  ↓
  Go/Java or Scala/Python/Ruby

* Prometheus ..... [Time-series DB]
  Query lang. called PromQL

* Grafana : Build dashboards ~~from the collected metrics~~.
  [data source - Prometheus]

* Alert manager : another tool from the Prometheus ecosys.
  ↓
  configure alerts when certain metrics reach or pass a threshold

* Thanos / Cortex → storage for Prometheus

- Tracing :

*

→ Microservice arch ←
(How a req. is processed) → tracing can be of good use.
⇓
describes the tracking of a req. while it passes through the services.

* Trace can include
  - start & finish time ✓
  - name ✓
  - tag ✓
  - log msg ✓

* Traces → stored and analyzed in a tracing sys. like Jaeger

__/__/____

* 2019: OpenTracing + OpenCensus → OpenTelemetry proj. (CNCF proj.)
  (projects)

  → APIs, SDKs, Tools
  used to integrate telemetry such as metrics protocols

* OpenTelemetry clients —export telemetry data→ Jaeger (platform)

- Cost Management:
  Automatic and Manual optimizn

  - identify wasted and unused resources
    → Autoscaling helps to shut down instances that are not needed.
  - Right-sizing
  - reserved instances
    → on-demand pricing model
    (reserve resources and even pay for them upfront)
  - Spot instances
    e.g. Batch jobs or heavy load for short time
    → get unused resources that were over-provisioned

FINOPS:
* Cloud Fin. Mgmt (FinOps) is the operating model for the cloud - combining sys., best practices and culture to increase an org.'s ability to understand cloud costs.
* FinOps empowers collaborn between IT, Engg., Finance, procurement and the biz.
* Framework for managing operating expenditures in the cloud.
* Financial accountability to the variable spend model of cloud.