
# Cloud Computing

## Chapter 2 : Cloud Computing Technologies

By : Prof. Lamri SAYAD

2025

# Content

- Introduction
- The Foundation Technologies (Enablers)
  - Virtualization
  - SOA
  - Data Center
  - Utility Computing & pay-as-you-go model
  - Grid Computing
- The Core Technologies (Building Blocks)
  - **Virtualization**
  - **Containers**
  - **Orchestration**
  - Storage Systems
  - **Networking**
- Cloud Security Technologies
- Emerging & Advanced Cloud Tech

# Chapter Objectives

1. Define virtualization and explain its role in cloud computing.
2. Identify the key components of virtualization.
3. Describe different types of virtualization.
4. Explain how hypervisors manage virtual machines.
5. Understand how virtualization enables resource pooling and isolation.
6. Distinguish between full, para-, and hardware-assisted virtualization.
7. Discuss real-world virtualization technologies and examples.

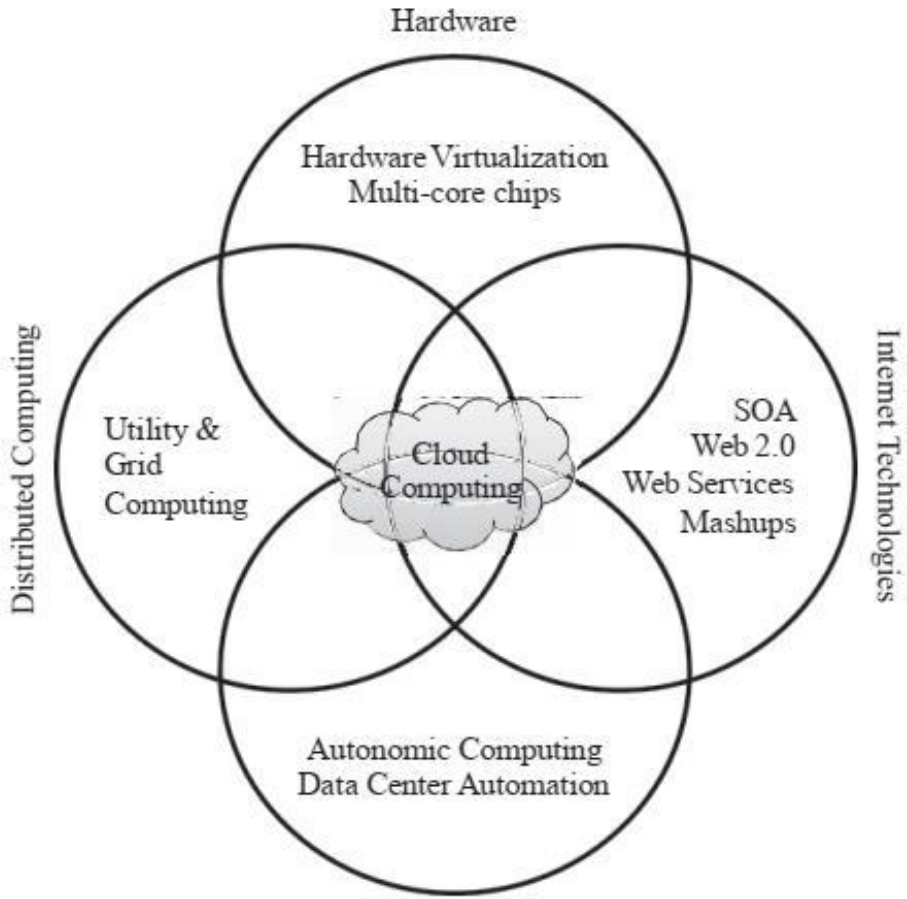
# Introduction

# Technologies

- The Foundational Technologies (Enablers)
  - Virtualization
  - SOA
  - Data Center
  - Utility Computing & pay-as-you-go model
  - Grid Computing
- The Core Technologies (Building Blocks)
  - **Virtualization**
  - **Containers**
  - **Orchestration**
  - Storage Systems
  - **Networking**
- Cloud Security Technologies
- Emerging & Advanced Cloud Tech

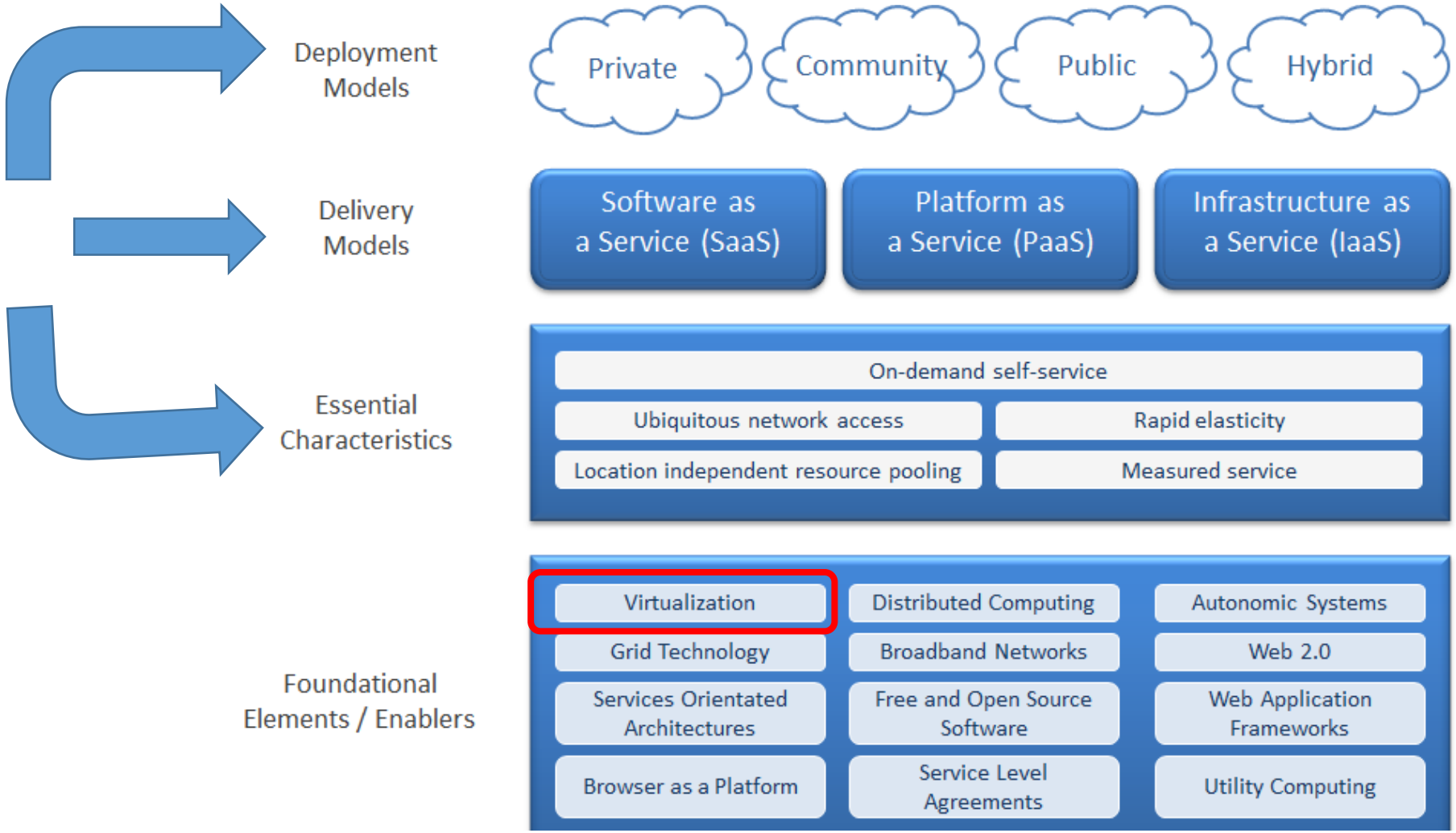
# The Foundational Technologies

# Enabling technologies



# Where we are

**Already covered**



# Virtualization

# Motivations

- Since 1960
- Three fundamental abstractions are necessary to describe the operation of a computing systems:
  (1) Interpreters/Processors,
  (2) Memory,
  (3) Communications links
- As the scale of a system and the size of its users grows, it becomes very challenging to manage its recourses (see three points above)

- Resource management issues:
  - provision for peak demands → **overprovisioning**
  - **heterogeneity** of hardware and software
  - machine failures
- **Virtualization is a basic enabler of Cloud Computing, it simplifies the management of physical resources for the three abstractions**

# Motivation

## Why virtualization ?

In traditional Environments :

- Only 10%-15% of total resources are consumed by the applications (according to VMware).
- Scaling up the environment is expensive
- The environment management isn’t an easy task

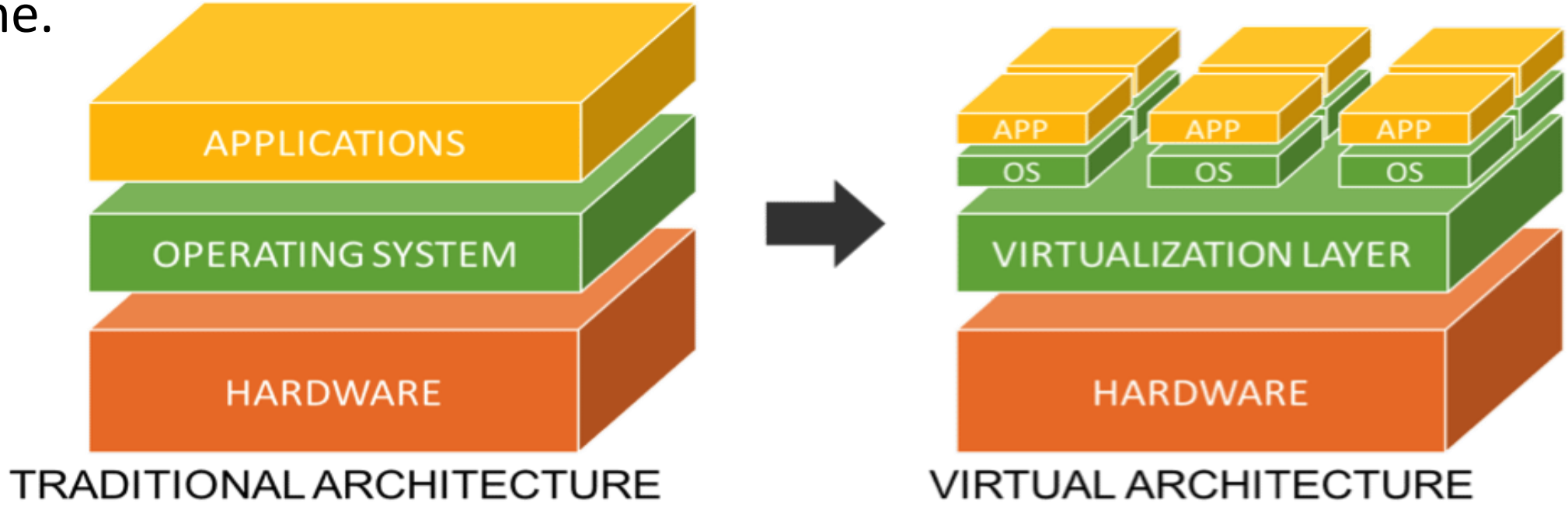
# What is virtualization ?

- *"**Virtualization**, in computing, refers to the act of creating a **virtual (rather than actual) version of something**, including but not limited to a virtual computer hardware platform, operating system (OS), storage device, or computer network resources."* from Wikipedia

- Virtualization abstracts the underlying resources; simplifies their use; isolates users from one another; and supports replication which increases the elasticity of a system

# What is virtualization ?

Virtualization refers to the creation of a virtual resource such as a server, desktop, operating system, file, storage or network to run multiple applications and various operating systems on the same SERVER at the same time.

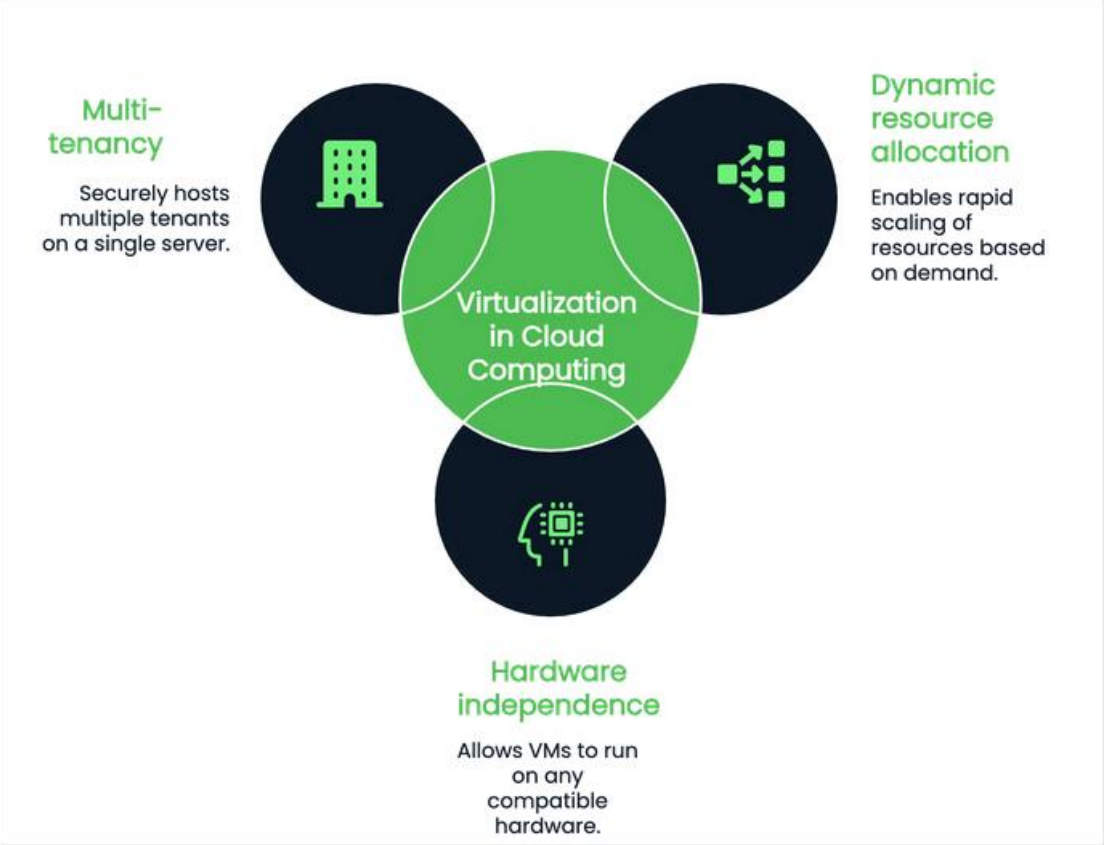


# What is virtualization ?

- Virtualization simulates the interface to a physical object by:
  - Multiplexing: creates multiple virtual objects from one instance of a physical object. Many virtual objects to one physical. Example - a processor is multiplexed among a number of processes or threads.
  - Aggregation: creates one virtual object from multiple physical objects. One virtual object to many physical objects. Example - a number of physical disks are aggregated into a RAID disk.
  - Emulation: constructs a virtual object of a certain type from a different type of a physical object. Example - a physical disk emulates a Random Access Memory (RAM).
  - Multiplexing and emulation. Examples - virtual memory with paging multiplexes real memory and disk; a virtual address emulates a real address.

# The role of virtualization in cloud computing

- **Dynamic resource allocation**: Virtualized environments can be scaled up or down quickly based on workload demands.
- **Hardware independence**: VMs and containers can run on any compatible physical host, making cloud workloads highly portable.
- **Multi-tenancy**: A single physical server can securely host multiple tenants with isolated environments.



# Key Components of Virtualization

**Key Components of Virtualization**

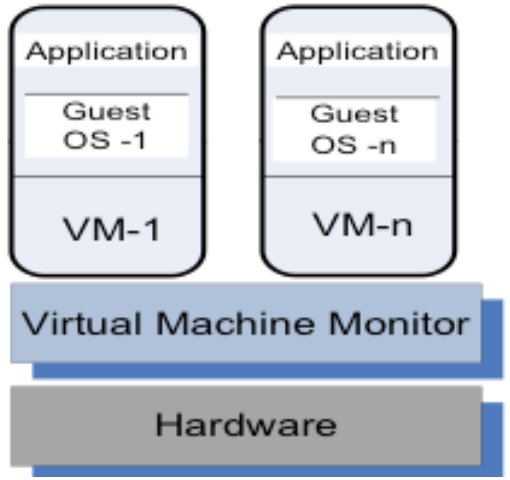
- Hypervisor or virtual machine monitor (VMM)
- Virtual machines (VMs)
- Virtual management tools

# Hypervisor

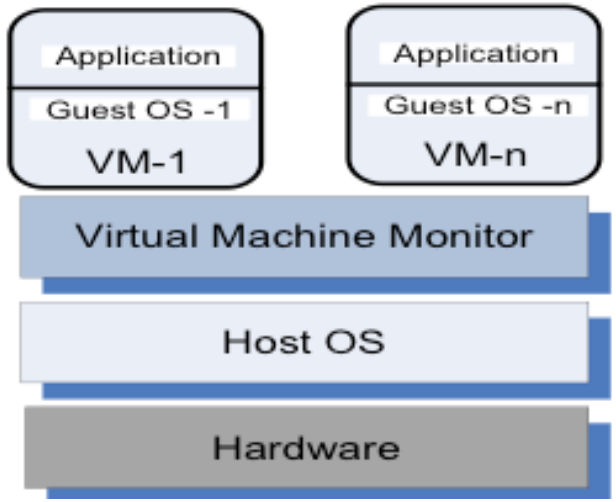
- Also called **Virtual Machine Monitor (VMM)**
- **Definition:** The core software layer that enables virtualization by separating physical hardware from virtual machines (VMs). It is a software layer responsible for creating and running virtual machines (VM) by partitioning the resources of a computer system.
- **Function:** It allocates physical resources (CPU, memory, storage, network) to multiple VMs while keeping them isolated and manages their execution.
- Allows several operating systems to run concurrently on a single hardware platform
- A VMM allows:
  - Multiple services to share the same platform
  - Live migration - the movement of a server from one platform to another
  - System modification while maintaining backward compatibility with the original system
  - Enforces isolation among the systems, thus security

# Hypervisor

- Two types of hypervisors (VMMs):
  - **Type 1 (bare-metal, native)**:
    - Runs directly on the physical hardware without a host OS.
    - These are used in data centers and cloud environments for high performance and security.
    - Examples include VMware ESX, Microsoft Hyper-V and Denali.




  - **Type 2 (hosted):**
    - Runs on top of a host operating system.
    - These are more common in development and testing environments.
    - Examples include VMware Workstation and VirtualBox.

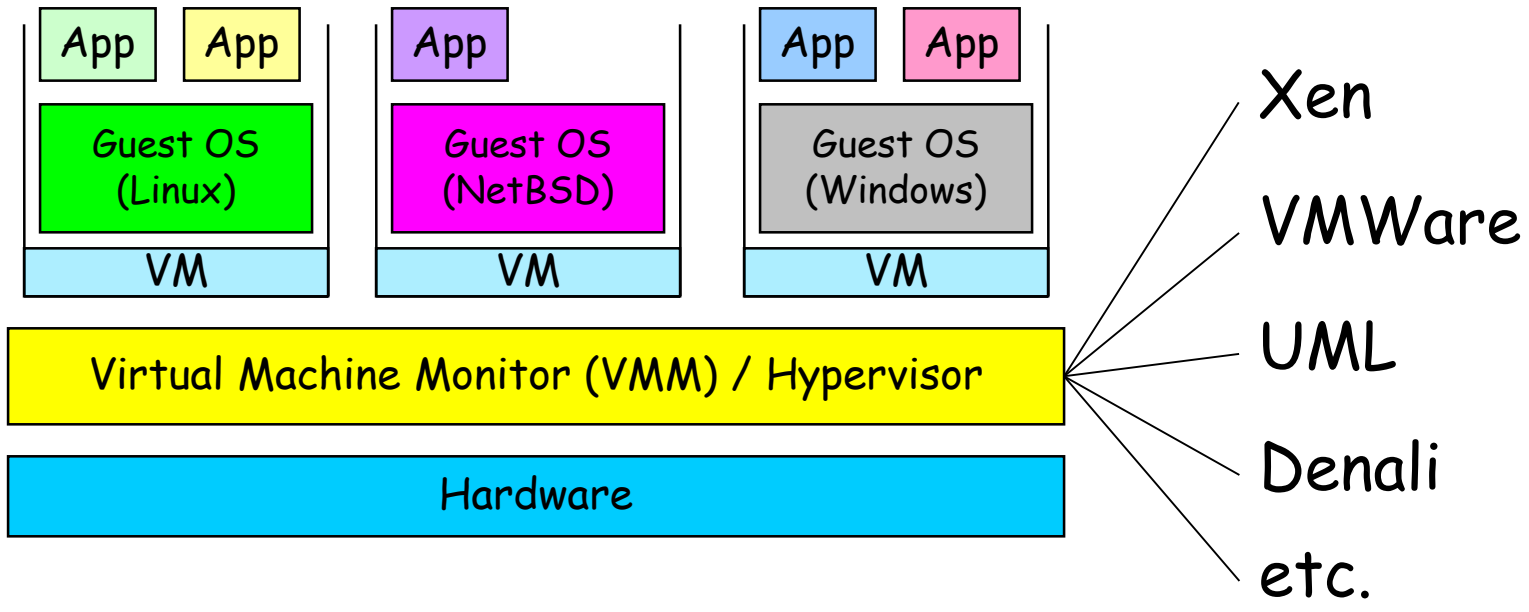


# Virtual machines

- **Definition**: A virtual machine is a virtualized environment (a software-based computer) operates like a physical machine.
- **Role**: Each VM behaves like an independent computer but shares hardware resources from the host.
- Each VM contains a virtualized hardware stack (CPU, memory, disk, and I/O devices), an operating system, and applications with processes.
- A **guest operating system** is an OS that runs in a VM under the control of the VMM.
- VMs can be paused, cloned, migrated across hosts, or restored from snapshots. This flexibility supports continuous delivery, disaster recovery, and infrastructure automation.

# Virtual Machines

- VM technology allows multiple virtual machines to run on a single physical machine.




*Performance*: Para-virtualization (e.g. Xen) is very close to raw physical performance!

# Virtualization management tools

- **Definition**: The set of tools or platforms used to manage, monitor, and orchestrate virtual resources.

- **Examples**: VMware vCenter, Microsoft System Center, OpenStack Nova, oVirt, Proxmox.

- **Functions**: Resource allocation, load balancing, live migration, fault tolerance, and monitoring & reporting (Real-time insights into VM health, performance, and resource usage).

# Types of Virtualization

- Server (or Hardware) virtualization
- Storage virtualization
- Network virtualization
- Application virtualization

# Server virtualization

- It divides a physical server into multiple virtual servers (or VMs).
- The hypervisor abstracts the physical hardware and allocates virtualized CPU, memory, storage, and networking resources to each VM.
- This type of virtualization is foundational for Infrastructure as a Service (IaaS) offerings in cloud computing, where physical servers are abstracted and offered as flexible, on-demand resources.
- Examples : VMware vSphere, KVM

# Server Virtualization techniques

- It describes *how* virtualization is implemented at the technical level — i.e., how the hypervisor manages to run multiple operating systems on the same physical hardware.

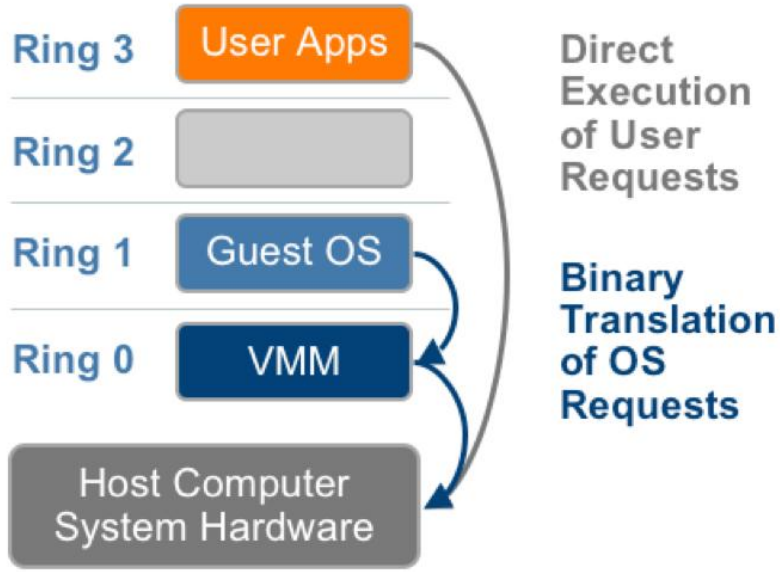
- **3 Techniques (**depending on how the **guest OS** interacts with the **hypervisor** and the **hardware) :**
  1. **Full** virtualization with binary translation
  2. OS-assisted Virtualization or **Paravirtualization**
  3. **Hardware** assisted virtualization

# Server Virtualization techniques :
## Full virtualization

- **Concept**

In **full virtualization**, a guest OS can run unchanged under the VMM as if it was running directly on the hardware platform. Each VM runs an exact copy of the actual hardware.

- **Mechanism**
  - **Binary translation** rewrites parts of the code on the fly to replace sensitive but not privileged instructions with safe code to emulate the original instruction
  - "The hypervisor translates all operating system instructions on the fly and caches the results for future use, while user level instructions run unmodified at native speed."
  - The **guest OS** doesn't know it's running in a virtualized environment.
- **Example**
  - VMware Workstation / ESXi
  - Microsoft Virtual PC

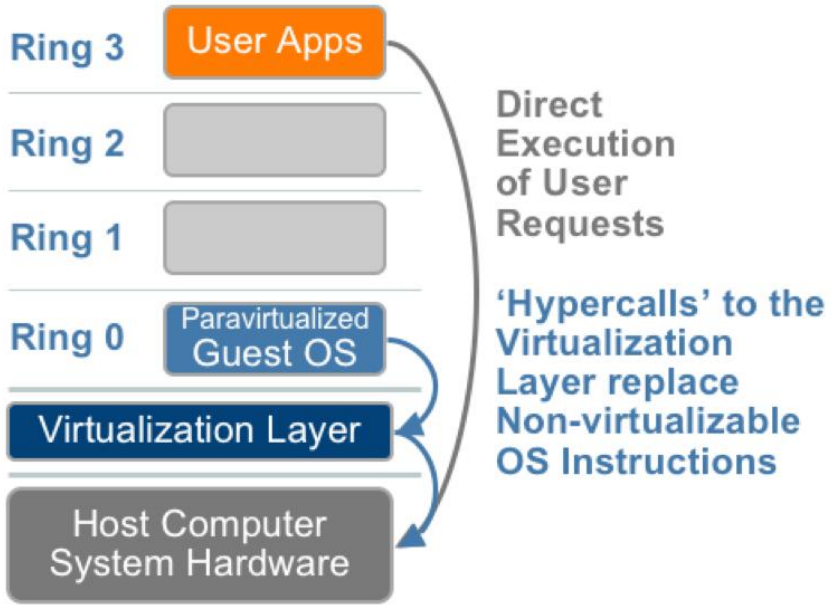


# Server Virtualization techniques :
## Paravirtualization

- **Concept**

In **para-virtualization**, the guest operating system is **modified** to work cooperatively with the hypervisor.
Instead of emulating hardware, the hypervisor provides special **APIs or hypercalls** to the guest OS.

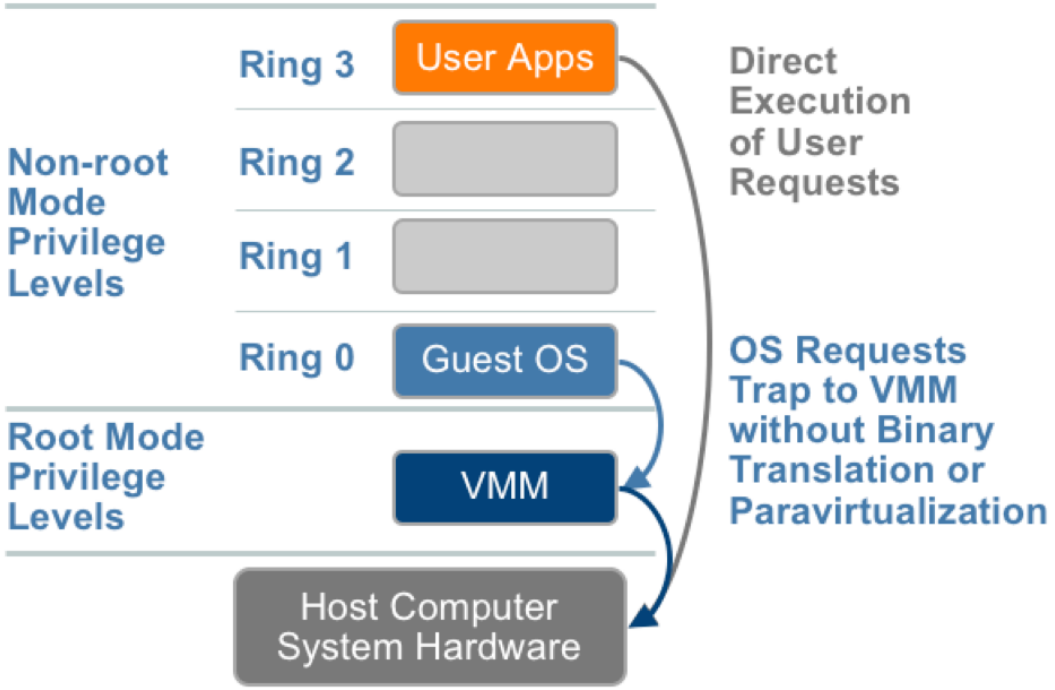
- **Mechanism**
  - The **guest OS kernel** is modified to replace sensitive hardware instructions with **hypercalls** (calls directly to the hypervisor).
  - This eliminates the need for instruction emulation, which boosts performance.
  - The hypervisor provides a **virtual API** rather than emulated hardware.
- **Example**
  - Xen (in para-virtualized mode)
  - VMware's early ESX versions



# Server Virtualization techniques :
## Hardware assisted virtualization

- **Concept**

**Hardware-assisted virtualization** *"a new CPU execution mode feature that allows the VMM to run in a new root mode below ring 0. As depicted in this Figure, privileged and sensitive calls are set to automatically trap to the hypervisor, removing the need for either binary translation or paravirtualization"*



# Storage virtualization

- **Storage virtualization** is the process of pooling multiple physical storage devices (hard drives, SSDs, SAN, NAS, etc.) into a **single, unified virtual storage resource** that appears to users and applications as one logical storage system.
- In simple terms:
  - ✓ It **hides the complexity** of where and how data is physically stored.
  - ✓ Users just see a **big virtual storage space**, not the individual disks or locations.

- **Analogy : bank account system**:
  - You don’t care in which **vault or locker** your money is physically kept.
  - The bank shows you **one balance (logical view)**, even though your money might be spread across multiple locations (physical storage).

- **Examples** : SAN, NAS, Ceph

# Storage virtualization
## Types of Storage Virtualization

1. **Block-level virtualization**
   - Used in **Storage Area Networks (SANs)**.
   - Aggregates block devices into logical volumes.
   - Example: RAID, VMware vSAN.

2. **File-level virtualization**
   - Used in **Network Attached Storage (NAS)**.
   - Pools multiple file servers into one namespace.
   - Example: NFS, DFS (Distributed File System).

3. **Object storage virtualization**
   - Data stored as objects (Data + metadata + unique ID).
   - Example: Amazon S3, OpenStack Swift.

# Network virtualization

- Network virtualization abstracts the physical network into logical networks.
- It creates virtual networks that operate independently from the underlying physical network hardware.
- It uses technologies like **VLANs**, virtual switches, and **Software-Defined Networking (SDN)** to abstract network resources, enhancing the scalability of cloud environments.
- This virtualization allows administrators to programmatically define network policies, segment traffic, and deploy isolated networks across shared infrastructure without physically rewiring or adding new hardware.

# Application virtualization

- With application virtualization, users can run applications in isolated environments without being installed directly on the user's operating system.
- The application is packaged with dependencies and executed in a containerized or virtual runtime environment.
- This prevents conflicts between apps, simplifies deployment, and supports centralized management.

# Where are we ?

- Introduction
- The Foundation Technologies (Enablers)
  - Virtualization
  - **SOA** 
  - Data Center
  - Utility Computing & pay-as-you-go model
  - Grid Computing
- The Core Technologies (Building Blocks)
  - **Virtualization**
  - **Containers**
  - **Orchestration**
  - Storage Systems
  - **Networking**
- Cloud Security Technologies
- Emerging & Advanced Cloud Tech

# Service-Oriented Architecture(SOA)

## What is a service?

- In the context of **Service-Oriented Architecture (SOA)**, a **service** is the **basic building block.**
- **Definition of a Service in SOA**
  - It is a self-contained, reusable, and network-accessible software component that performs a specific function and communicates with other services using standard protocols (like HTTP, SOAP, or REST).
  - the smallest functional unit that performs a specific business task and can be accessed remotely through a standardized interface.
- **Main Characteristics of a Service**
  - Self-contained
  - Loosely coupled
  - Interoperable
  - Stateless
  - Discoverable
  - Reusable

# Service-Oriented Architecture(SOA)

## What is a service?

- Each service:
  - Has its own code and possibly its own database.
  - Exposes its functions via an API or web service.
  - Communicates over the network using standard formats (like XML or JSON).
- **Example :** an **online store** that follows SOA principles. It could have the following services:
  - User Service : Manages user registration and authentication.
  - Product Service : Handles product catalog and availability.
  - Order Service : Processes customer orders.
  - Payment Service : Handles transactions with credit cards or PayPal.
  - Notification Service : Sends emails or SMS to customers.

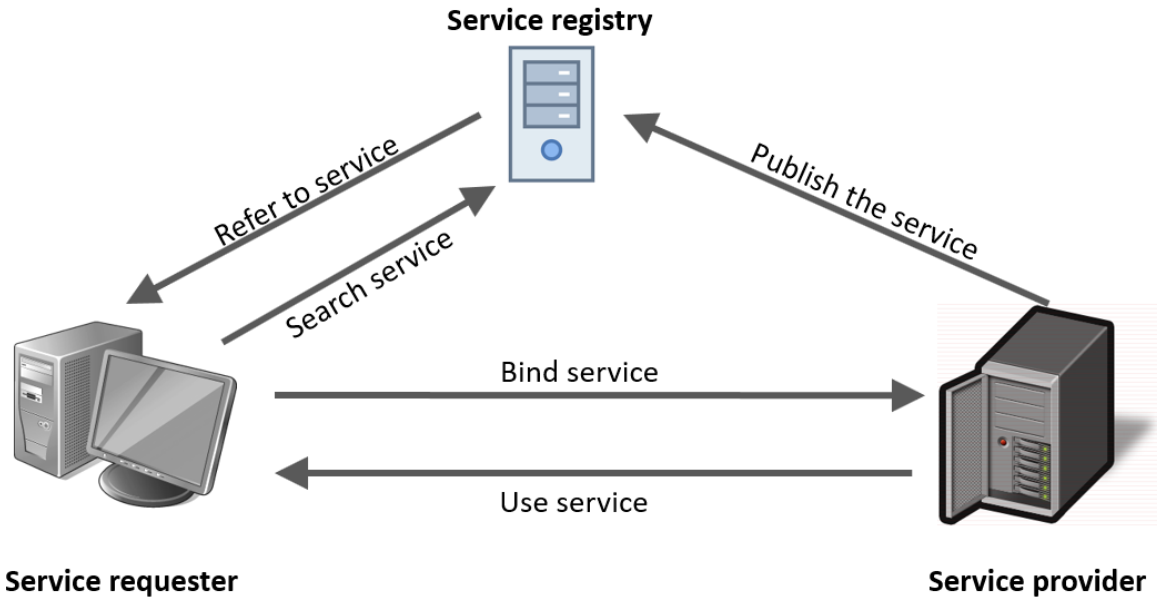
# Service-Oriented Architecture(SOA)
## SOA Definitions

- **Service-Oriented Architecture (SOA)** is
  - a common concept in cloud computing and distributed systems.
  - a design approach in which software components (called services) provide functionality to other components over a network, using standardized communication protocols.
  - It helps to use applications as a service for other applications regardless the type of vendor, product or technology.

# Service-Oriented Architecture
## SOA Components

- **SOA Components:**

1. **Service Provider** – creates and publishes services.
2. **Service Consumer** – requests and uses services.
3. **Service Registry** – a directory where available services are described and can be discovered.



# Service-Oriented Architecture(SOA)

## Real-World Example: Online Banking System

- Instead of building **one large monolithic application**, banks use **SOA** : dividing functionality into **independent services** that communicate through well-defined interfaces.
- Key services in the Banking SOA :
  - Customer Service : Manages customer profiles, authentication, and permissions.
  - Account Service : Handles checking, savings, and credit accounts (balance, transactions).
  - Payment Service : Processes payments, transfers, and bill settlements.
  - Loan Service : Manages loan applications, approvals, and repayments.
  - Notification Service : Sends SMS/email alerts for transactions, OTP codes, or low balances.
  - Reporting Service : Generates financial statements and dashboards for clients and managers.
  - Security Service : Provides encryption, authentication, and fraud detection mechanisms.

# Service-Oriented Architecture(SOA)

## Real-World Example: Online Banking System

- **How It Works (Example Scenario) : A user wants to transfer money via a mobile app:**

1. **Customer Service** authenticates the user (login).
2. The app calls the **Account Service** to display available accounts and balances.
3. The user enters the transfer details → the **Payment Service** handles the transaction.
4. The **Security Service** verifies the transaction with two-factor authentication.
5. After success, the **Notification Service** sends an SMS confirmation.
6. The **Reporting Service** updates the user's monthly statement.

# Service-Oriented Architecture(SOA)

## Real-World Example: Online Banking System

- **Example of Communication:** Each service exposes an interface, for example:

```
POST /api/payment/transfer
Content-Type: application/json

{
  "fromAccount": "12345",
  "toAccount": "98765",
  "amount": 500,
  "currency": "USD"
}
```

- The **Payment Service** processes the transfer, logs it, and responds with (JSON):

```
{
  "transactionId": "TXN-2025-00123",
  "status": "SUCCESS",
  "message": "Transfer completed successfully."
}
```

# Where are we ?

- Introduction
- The Foundation Technologies (Enablers)
  - Virtualization
  - SOA
  - Data Center 
  - Grid Computing
  - Utility Computing & pay-as-you-go model
- The Core Technologies (Building Blocks)
  - **Virtualization**
  - **Containers**
  - **Orchestration**
  - Storage Systems
  - **Networking**
- Cloud Security Technologies
- Emerging & Advanced Cloud Tech

# Data Center
## Definition 1 (in general)

- A data center is a physical room, building or facility that houses IT infrastructure for building, running and delivering applications and services.
- It also stores and manages the data associated with those applications and services.

# Data Center
## Definition 2 (Cloud Computing context)

- A large IT is that houses :
  - **computer servers**,
  - **storage systems**,
  - **networking equipment**,
  - and **cooling and power infrastructure**
- Used to run and deliver **cloud services** to users over the Internet.

- All “virtual” cloud resources are actually hosted on **physical machines** inside these data centers.

# History

- **1945** : The US military's ENIAC, completed at the University of Pennsylvania, is an early example of a data center that required dedicated space to house its massive machines.

- **In the 1990s** :
  - computers became more size-efficient, requiring less physical space.
  - These microcomputers that began filling old mainframe computer rooms became known as “servers,” and the rooms became known as “data centers.”

- **In the 2000s** :
  - The advent of Cloud computing drives the development of data centers

# Types of data centers

- **Enterprise (on-premises) data centers**
  - The company is responsible for all deployment, monitoring and management tasks.
  - The company have more control over information security and can more easily comply with regulations
- **Cloud data centers (hyperscale)**
  - Hosts IT infrastructure resources for shared use by multiple customers.
  - Hyperscale data centers are larger than traditional data.
  - They typically contain at least 5,000 servers and miles of connection equipment, and they can sometimes be as large as 60,000 square feet.
- **Edge data centers**
  - Cloud service providers typically maintain smaller, edge data centers (EDCs) located closer to cloud customers.
  - It is a distributed computing framework that brings applications closer to end users.
- **Managed data centers and colocation facilities**
  - options for organizations that lack the space, staff or expertise to manage their IT infrastructure on-premises.
  - ideal for those who prefer not to host their infrastructure by using the shared resources of a public cloud data center.
  - In a managed data center, the client company leases dedicated servers, storage and networking hardware from the provider
  - In a colocation facility, the client company owns all the infrastructure and leases a dedicated space to host it within the facility.

# Data center Infrastructure Components

- **Servers**
- **Storage systems**
- **Networking**
- **Power supply and cable management**
- **Redundancy and disaster recovery**
- **Environmental controls**

# Servers

- Servers are powerful computers that deliver applications, services and data to end-user devices.

| SERVER TYPE | CPU'S | STORAGE | RAM |
|---|---|---|---|
| **Powerful-128GB** | 24 core CPU / 48 vCPU's | 6TB SSD with Raid 5 | 128GB |
| **Powerful-256GB** | 36 core CPU / 72 vCPU's | 10TB SSD with Raid 5 | 256GB |
| **Powerful-512GB** | 40 core CPU / 80 vCPU's | 12TB SSD with Raid 5 | 512GB |

- Data center servers come in several form factors:
  - Tower servers
  - Rack-mount servers
  - Blade servers
  - Mainframes

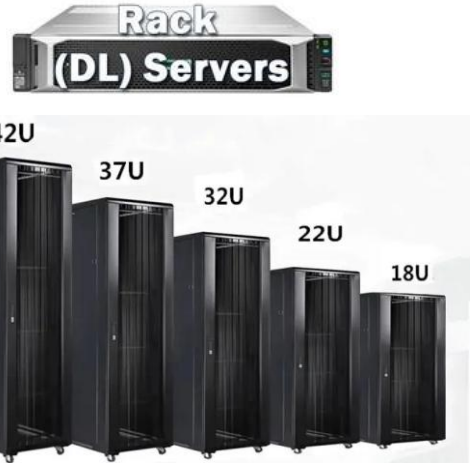
# Server types

- **Tower servers :**
  - Its shape and structure are almost the same with the ordinary PC host, but a slightly larger head,
  - The size does not have a uniform standard.


- **Mainframe :**
  - **Is a high-performance computer designed for large-scale, mission-critical processing of massive amounts of data and transactions.**
  - can do the work of an entire room of rack-mount or blade servers.

# Server type

- **Rack-mount servers :** They are stacked on top of each other in a rack to save space.
  - A 42U rack might hold 20-40 individual servers.
  - Each server has its own power supplies, its own network ports and cables at the back.
  - Good for small deployments or one-off purchases.




- **Blade servers :** designed to save even more space. They're made to fit into a chassis that holds multiple blades.
  - A single 10U chassis could hold 16 blades, allowing for over 100 servers in a 42U rack.
  - The chassis has a few, large, shared power supplies and cooling fans for all blades.
  - You scale by adding blades to a chassis.
  - Good for large scale virtualization, HPC clusters and large web and Apps farms

# Storage systems

- A **storage system** is a combination of **hardware and software** designed to **store, manage, protect, and retrieve digital data** efficiently and reliably.

- Components of a Storage System
  - Storage Devices (e.g., HDDs, SSDs, NVMe drives, or tape drives.)
  - Storage Controllers : Manage how data is read/written from devices
  - Storage Network (e.g., via iSCSI, Fibre Channel, or Ethernet).
  - Management Software (e.g., RAID, replication, snapshots).

# Types of Storage systems

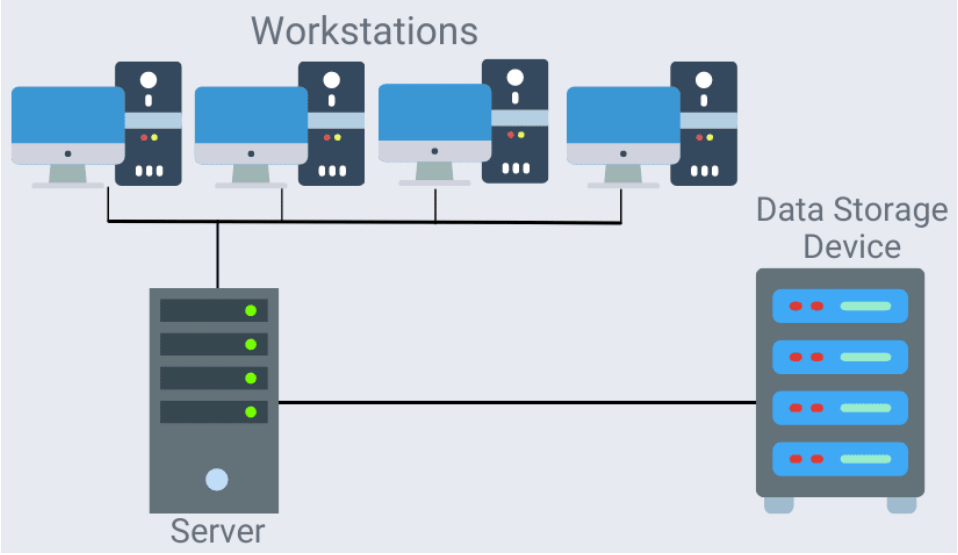
- Direct-Attached Storage (DAS) :
  - Most servers include some local storage capability:
  - Enable the most frequently used data (hot data) to remain close to the CPU.

- Network Attached Storage (NAS)
  - File-level storage accessible over a network (like a shared folder).

- Storage Area Network (SAN)
  - Block-level storage accessible via a dedicated network (used in data centers).

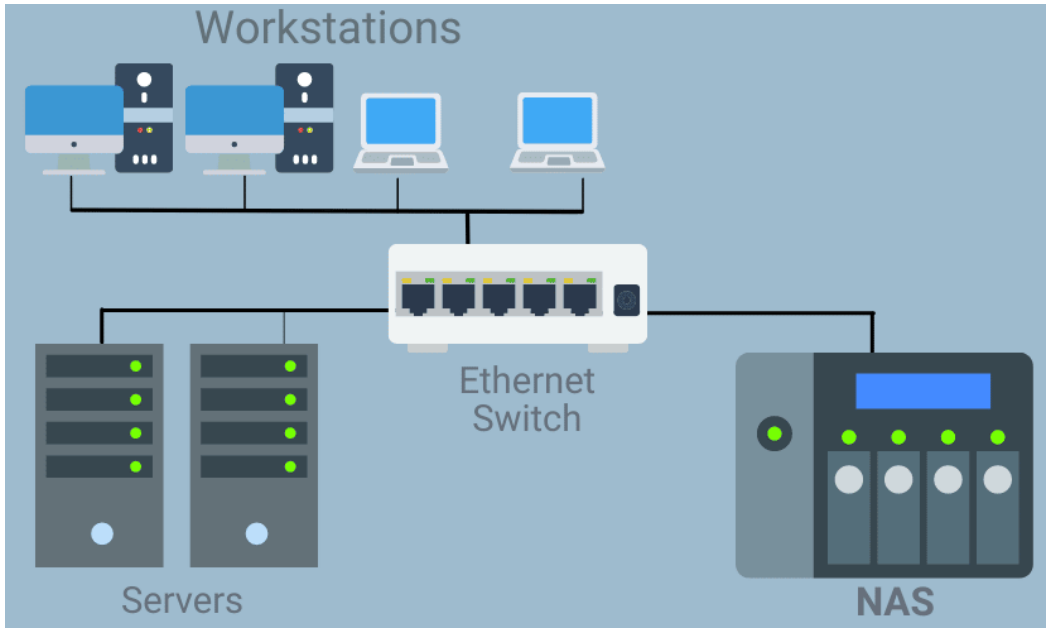
# Direct Attached Storage (DAS)

- low-cost digital storage system, having from one to several hard disk drives.
- is directly connected to a server or workstation through a host bus adapter, without a network in between



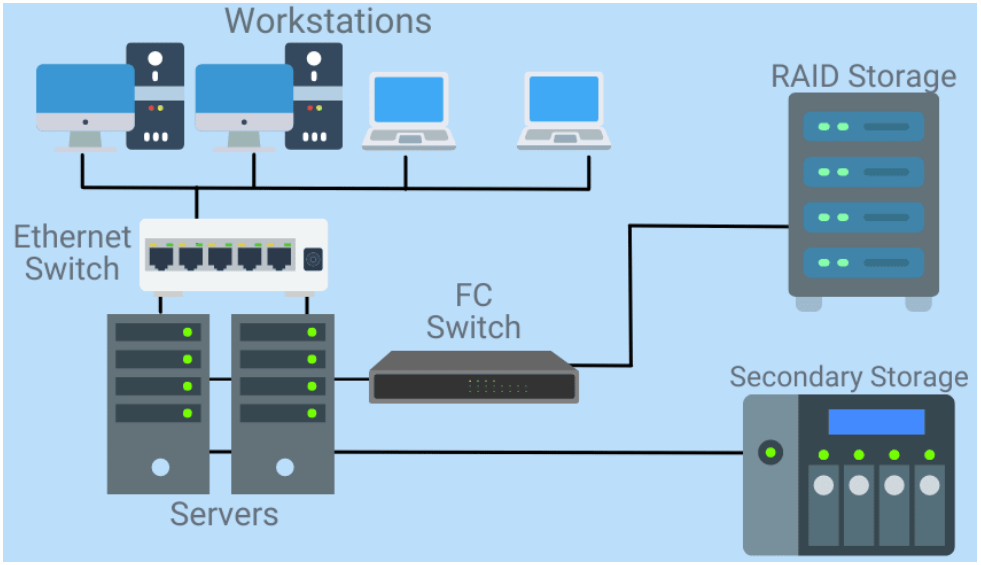
# Network Attached Storage (NAS)

- NAS is a dedicated server that attaches to a network with various storage media such as hard disk drives (HDDs) or solid-state drives (SSDs)
- NAS uses an Ethernet connection and can be accessed through an assigned network address.
- appears to the server operating system as a file share (e.g., \\NAS\SharedFolder or nfs://nas/server/).

# Storage Area Network (SAN)

- The key idea is that the storage is *networked* but appears to the server operating system as a **local disk**.
- but it uses a separate network for the data
- It involves a more complex mix of multiple storage servers, application servers and storage management software.



# Where are we ?

- Introduction
- The Foundation Technologies (Enablers)
  - Virtualization
  - SOA
  - Data Center
  - Grid Computing 
  - Utility Computing & pay-as-you-go model
- The Core Technologies (Building Blocks)
  - **Virtualization**
  - **Containers**
  - **Orchestration**
  - Storage Systems
  - **Networking**
- Cloud Security Technologies
- Emerging & Advanced Cloud Tech

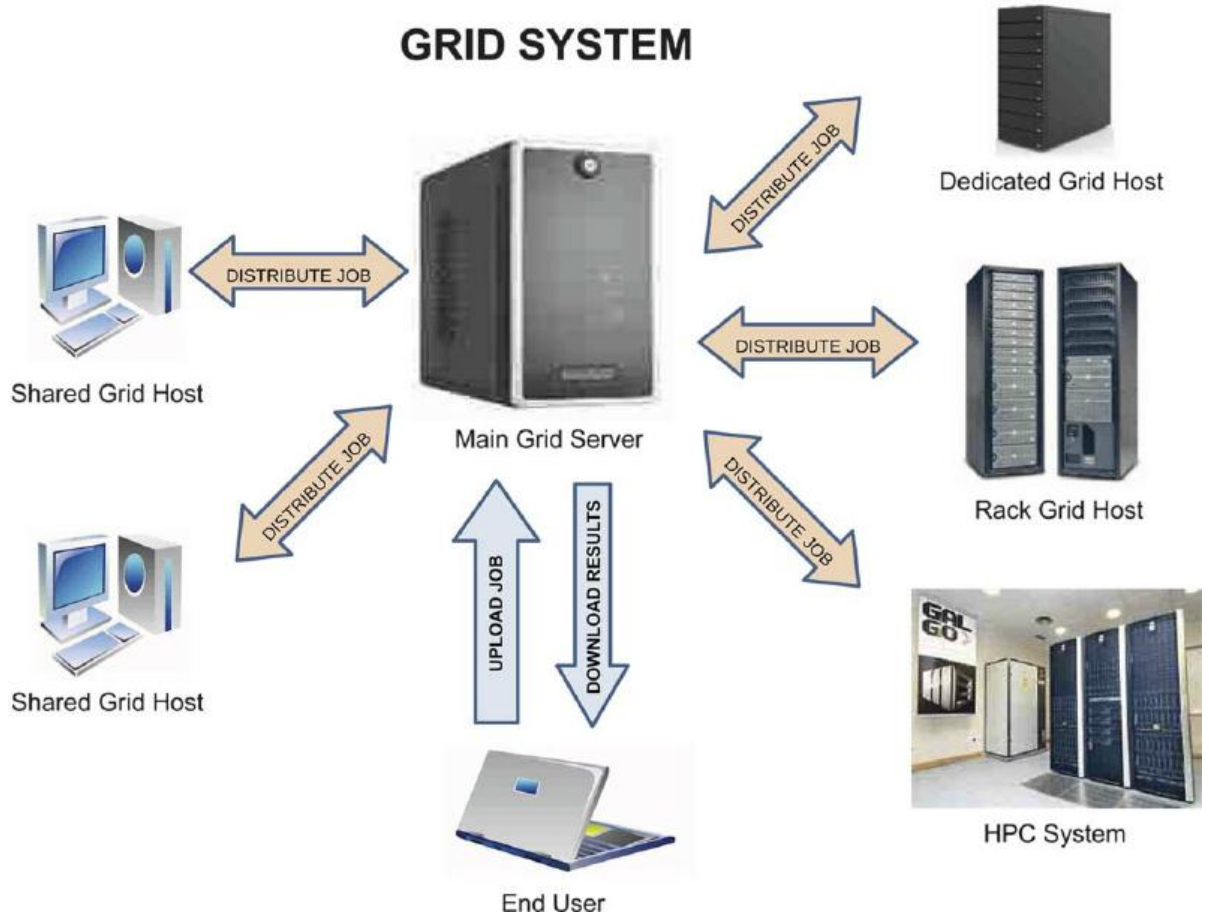
# Why do we need Grids?

- Many large-scale problems cannot be solved by a single computer
- Globally distributed data and resources

# Grid Computing

- **Grid computing** is a distributed architecture that combines computer resources from different locations to achieve a common goal. It breaks down tasks into smaller subtasks, allowing concurrent processing.
- These computer resources are heterogeneous and geographically dispersed.
- Grid Computing breaks complex task into smaller pieces. These smaller pieces are distributed to CPUs that reside within the grid.

# Grid Computing



# Where are we ?

- Introduction
- The Foundation Technologies (Enablers)
  - Virtualization
  - SOA
  - Data Center
  - Grid Computing
  - Utility Computing & pay-as-you-go model 
- The Core Technologies (Building Blocks)
  - **Virtualization**
  - **Containers**
  - **Orchestration**
  - Storage Systems
  - **Networking**
- Cloud Security Technologies
- Emerging & Advanced Cloud Tech

# Utility Computing

- **Utility computing** is a **computing model** in which computing resources—such as processing power, storage, and networking—are **provided to users on demand** and **billed based on usage**, similar to how public utilities (like electricity, water, or gas) are consumed.
- In simpler terms, **it treats computing as a utility**: you don't buy and maintain the infrastructure; instead, you use what you need and pay only for what you consume.
- **Key Characteristics**
  - On-demand access
  - Pay-per-use model
  - Scalability
  - Resource pooling
  - Service-based delivery

# Relationship to Cloud Computing

- Utility computing is a **precursor** and **foundation concept** of **cloud computing**.

- While cloud computing adds features like self-service portals, automation, virtualization, and multi-tenancy, it still relies on the **utility computing principle**: *“Pay only for what you use.”*

# Where are we ?

- Introduction
- The Foundation Technologies (Enablers)
  - Virtualization
  - SOA
  - Data Center
  - Grid Computing
  - Utility Computing & pay-as-you-go model
- The Core Technologies (Building Blocks) 
  - **Virtualization**
  - **Containers**
  - **Orchestration**
  - Storage Systems
  - **Networking**
- Cloud Security Technologies
- Emerging & Advanced Cloud Tech

# Where are we ?

- Introduction
- The Foundation Technologies (Enablers)
  - Virtualization
  - SOA
  - Data Center
  - Grid Computing
  - Utility Computing & pay-as-you-go model
- The Core Technologies (Building Blocks)
  - **Virtualization**
  - Containers 
  - **Orchestration**
  - Storage Systems
  - **Networking**
- Cloud Security Technologies
- Emerging & Advanced Cloud Tech

# Motivation for Containerization

- Traditionally, to run any application on your computer, you had to install the version that matched your machine's operating system. For example, you needed to install the Windows version of a software package on a Windows machine.

- VMs are great, but have high runtime overhead
  - Can we scale faster and more easily?

# What is containerization?

- Containerization is the **packaging of software code** with just the operating system (OS) libraries and dependencies required to run the code to create a **single lightweight executable**—called a container—that runs consistently on **any** infrastructure. (**IBM**)
- Containerization is the packaging together of software code with all it's necessary components like libraries, frameworks, and other dependencies so that they are isolated in their own "container." (**Red Hat**)
- Containerization is a software deployment process that bundles an application's code with all the files and libraries it needs to run on any infrastructure. With containerization, you can create a single software package, or container, that runs on all types of devices and operating systems. (**Amazon**)

# What is containerization?

- OS-level virtualization.
- Deploy and Run distributed applications without VMs
- Isolated Environment
  - Each container can have different files, environment variables, libraries, and different OS.
  - Multiple isolated application can run on a single host and access the same OS kernel

# What is a Container

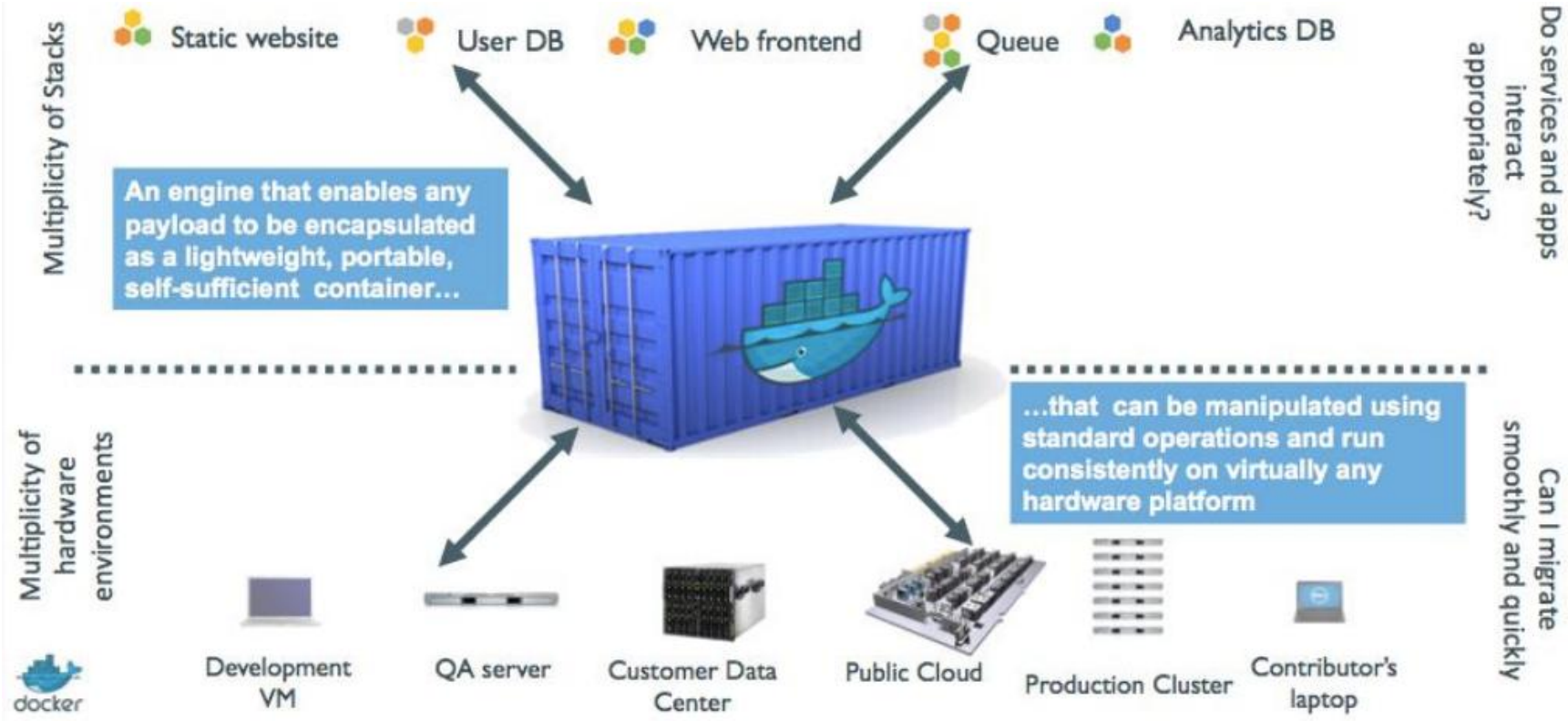
Shipping containers

- One size fits most loads
- Massive efficiency improvements
- Standard size and configuration

# What is a Container

Containers:

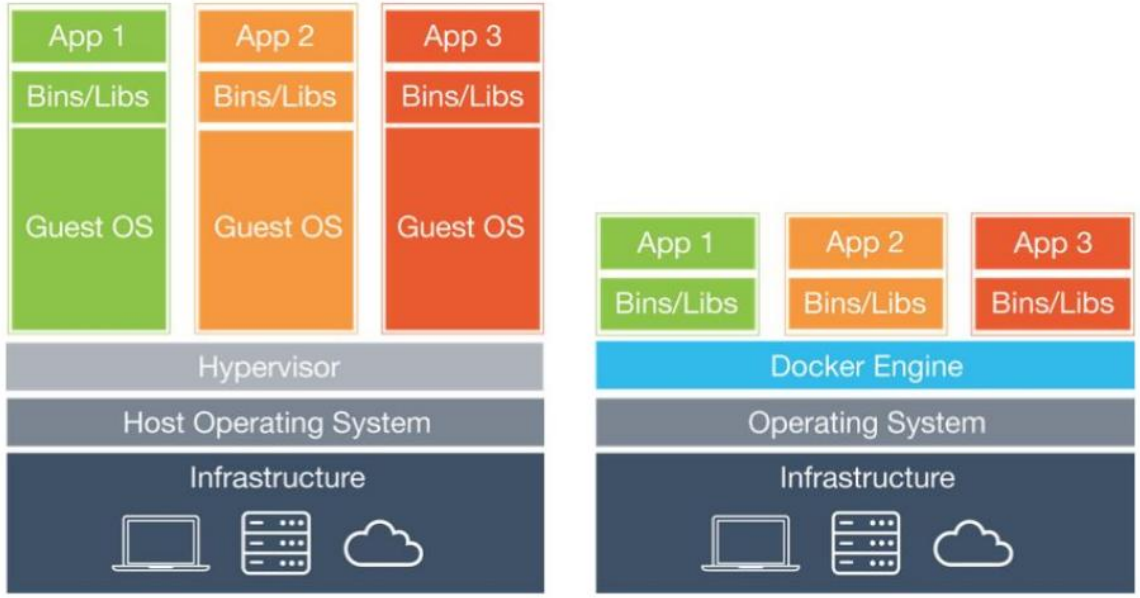
- Portability for our applications
- Standard format
- Includes all the dependencies for the application
- Workload isolation at runtime




- Is a lightweight, portable box that holds everything an application needs to run smoothly, such as code, libraries, and system tools .

# Big Idea: Containers have less OS overhead

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




Traditional VM-based Infrastructure

Container Infrastructure

# Containerization architecture

Containerization architecture consists of four essential component layers.

- Underlying IT infrastructure
- Host operating system
- Container image
- Containerized applications

# How does containerization work?

- Containers encapsulate an application as a single executable package of software that bundles application code together with all of the related configuration files, libraries and dependencies required for it to run.
- Containerized applications are “isolated”.
- An open-source container runtime or container engine (like Docker runtime engine) is installed on the host’s operating system and becomes the conduit for containers to share an operating system with other containers on the same computing system.
- Other container layers, like common binaries (bins) and libraries, can be shared among multiple containers. This feature eliminates the overhead of running an operating system within each application and makes containers smaller in capacity and faster to start up than VMs, driving higher server efficiencies
- Containers can be easily transported from a desktop computer to a virtual machine (VM) or from a Linux to a Windows operating system.

# Containerization vs virtualization

- Containers are often compared to virtual machines (VMs) because both technologies enable significant compute efficiencies by allowing multiple types of software (Linux- or Windows-based) to run in a single environment.
- Virtualization utilizes a hypervisor, a software layer placed on a physical computer or server that allows the physical computer to separate its operating system and applications from its hardware.
- Virtualization technology allows multiple operating systems and software applications to run simultaneously and share a single physical computer or host machine’s resources
- Each application and its related file system, libraries and other dependencies—including a copy of the operating system (OS)—are packaged together as a VM.

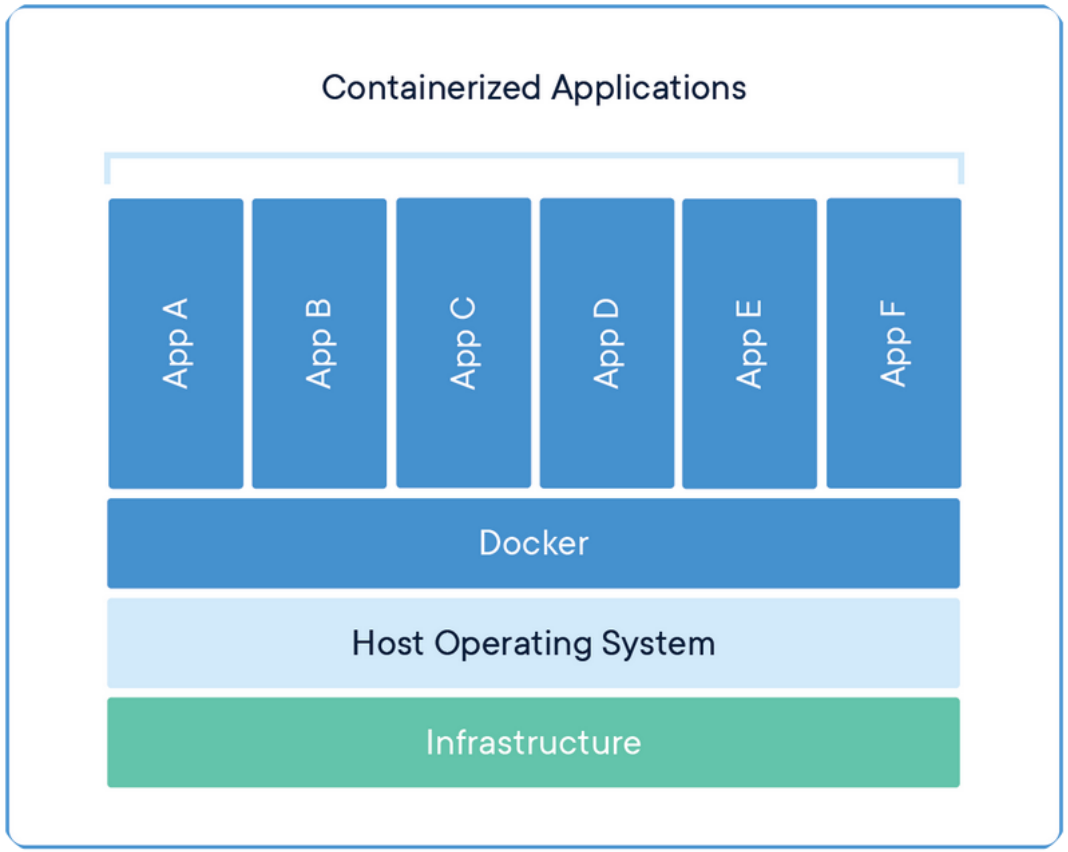
# Containerization vs virtualization

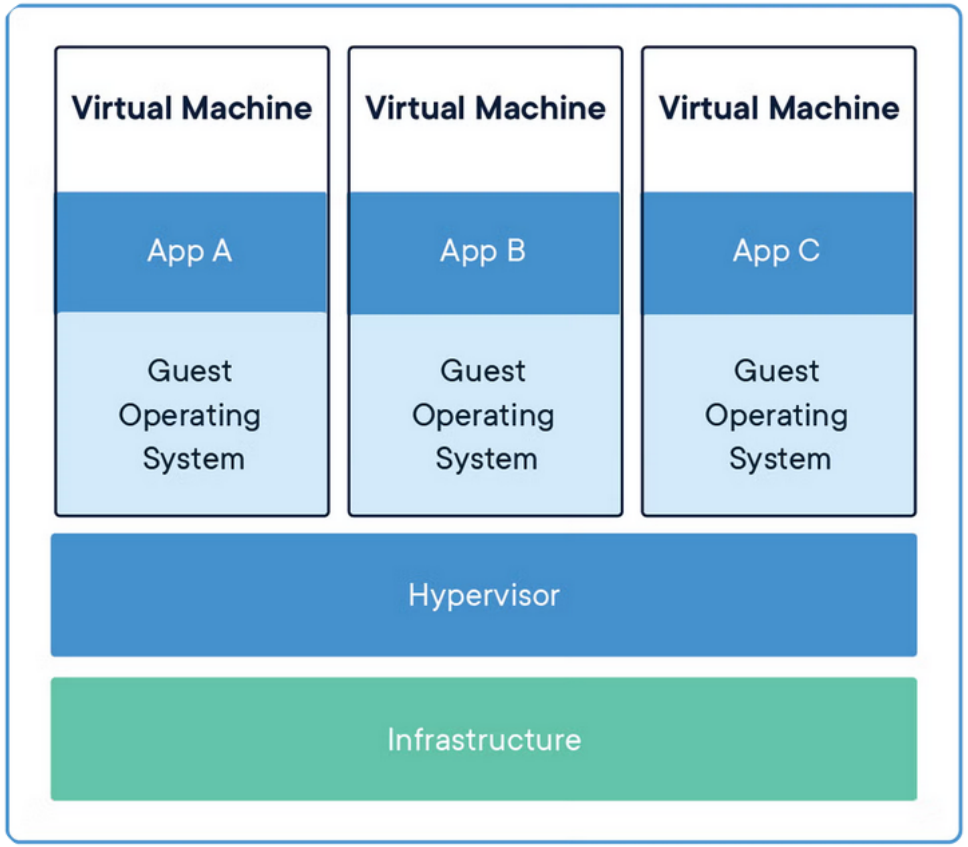
- Containerization, on the other hand, creates a single executable package of software that bundles application code together with all of its dependencies required for it to run.
- Unlike VMs containers do *not* bundle in a copy of the OS. Instead, the container runtime engine is installed on the host system's operating system, or "host OS," becoming the conduit through which all containers on the computing system share the same OS.
- Containers are often called "lightweight"—they share the machine's OS kernel and do not require the overhead of associating an OS within each application (as is the case with a VM).
- Other container layers (common bins and libraries) can also be shared among multiple containers, making containers inherently smaller in capacity than a VM and faster to start up. Multiple containers can run on the same compute capacity as a single VM

# Containerization vs virtualization

| | Container | Virtualization |
|---|---|---|
| OS Level | Shares host OS & Kernel | Independent OS & Kernel |
| Resource Utilization | Lightweight | High Overload |
| Isolation | Only process level | Stronger Isolation |
| Portability & Consistency | Highly Consistent and Portable | Less Standardized and less Portable |
| Use Cases | Microservices, Faster deployments | Legacy Apps, Full isolation |

# Containerization vs virtualization

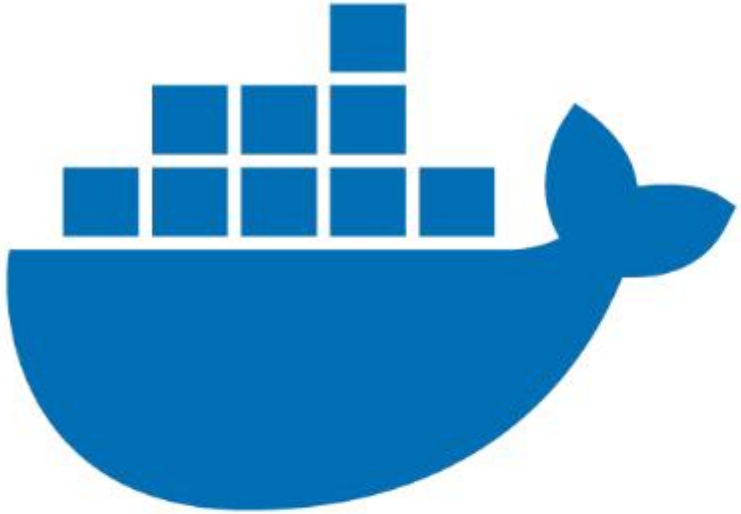


# Benefits of containerization

- Portability
- Agility
- Speed
- Fault isolation
- Efficiency
- Ease of management
- Security

# Popular Containerization Platforms



docker

kubernetes

# What is Docker?

- Open-source platform to develop, deploy, and run applications with containers.
- Abstracts hardware virtualization for containers.
- Client-Server Architecture

# Docker Components

- Docker engine
  - The core software that runs and manages containers.
- Docker Image
  - A **blueprint** or **template** that defines what's inside the container (like a recipe).
- Docker Container
  - A **running instance** of an image (similar to VMs)
- Dockerfile
- Docker Hub
  - A public repository (like GitHub, but for Docker images). You can download ready-made images
  - Available on : **https://hub.docker.com/**

# Docker Image

- A **Docker image** is a **read-only template** that contains everything needed to create (or run) a container:
  - ✓The operating system files (like Ubuntu or Alpine Linux)
  - ✓Installed packages and dependencies
  - ✓The application code
  - ✓Configuration files and environment variables

- When you “run” an image, Docker creates a **container** — a live, writable instance of that image.

# The Dockerfile

- Dockerfile
  - ➢A simple text file with instructions to build an image.
  - ➢Describes everything your container needs:
    - Dependencies
    - Source Code / Binaries

```
FROM ubuntu
RUN apt-get install python3
COPY app.py /app/
CMD ["python3", "/app/app.py"]
```

- Purpose
  - Standardized Builds
  - Automated Image Creation
  - Efficient Deployment

# The Dockerfile

Components of a Dockerfile

- Base Image
- RUN Instruction
- COPY and ADD instruction
- Execution and Runtime Configuration
- Expose Instruction
- ENV Instruction
- WORKDIR Instruction
- User Configuration

# The Dockerfile

Essential Commands:

- FROM - Inherit from a parent container
  - i.e. “FROM ubuntu”
- RUN - Runs a command during the build process
  - i.e. “RUN apt-get install python3”
- ADD - Copies files from the build directory into the image
  - i.e. “ADD hello_world.py /usr”
- EXPOSE - Register a port that the image will listen on
  - i.e. “EXPOSE 80”
- CMD - Set the default command to be executed on startup
  - i.e. “CMD python /usr/hello_world.py”

# How to run code in Docker

1. Inherit from a parent OS/platform container
2. Install any packages / libraries you need
3. Add any source code you need
4. Attach any volumes you need for data persistence
5. Set a command to be run at startup

# How to run code in Docker
## Example

- Let's say you have a web app built with Python Flask.
- Normally, you would have to:
  1. Install Python on the server
  2. Install Flask and other dependencies
  3. Configure everything manually
- With Docker, you can create a **Dockerfile** that describes your app environment.
- Then, you build an **image** and run it as a **container**:

```
docker build -t myflaskapp .
docker run -p 5000:5000 myflaskapp
```

# Tools

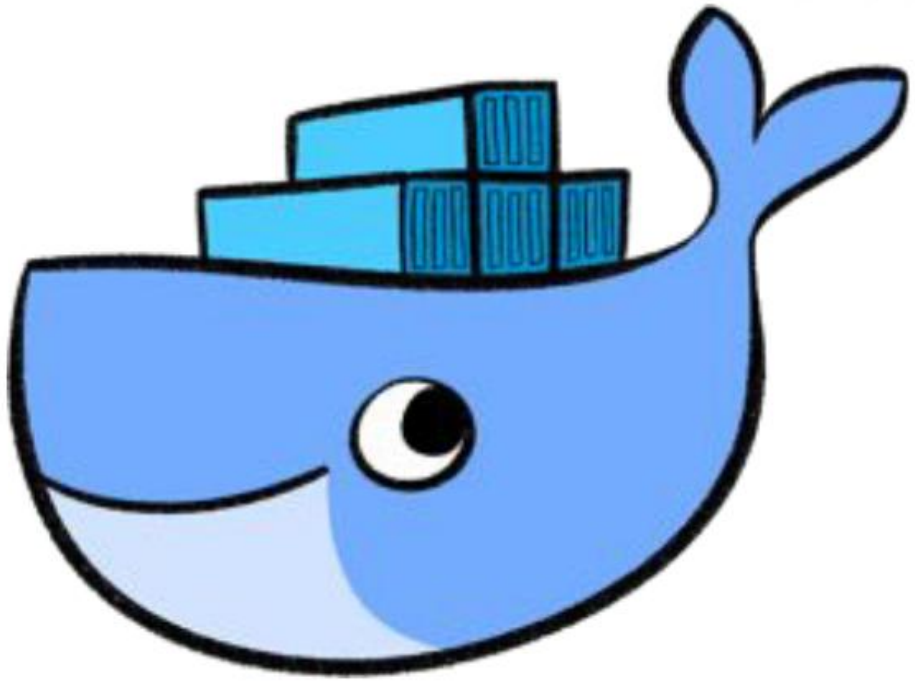
Tools to scan Docker containers

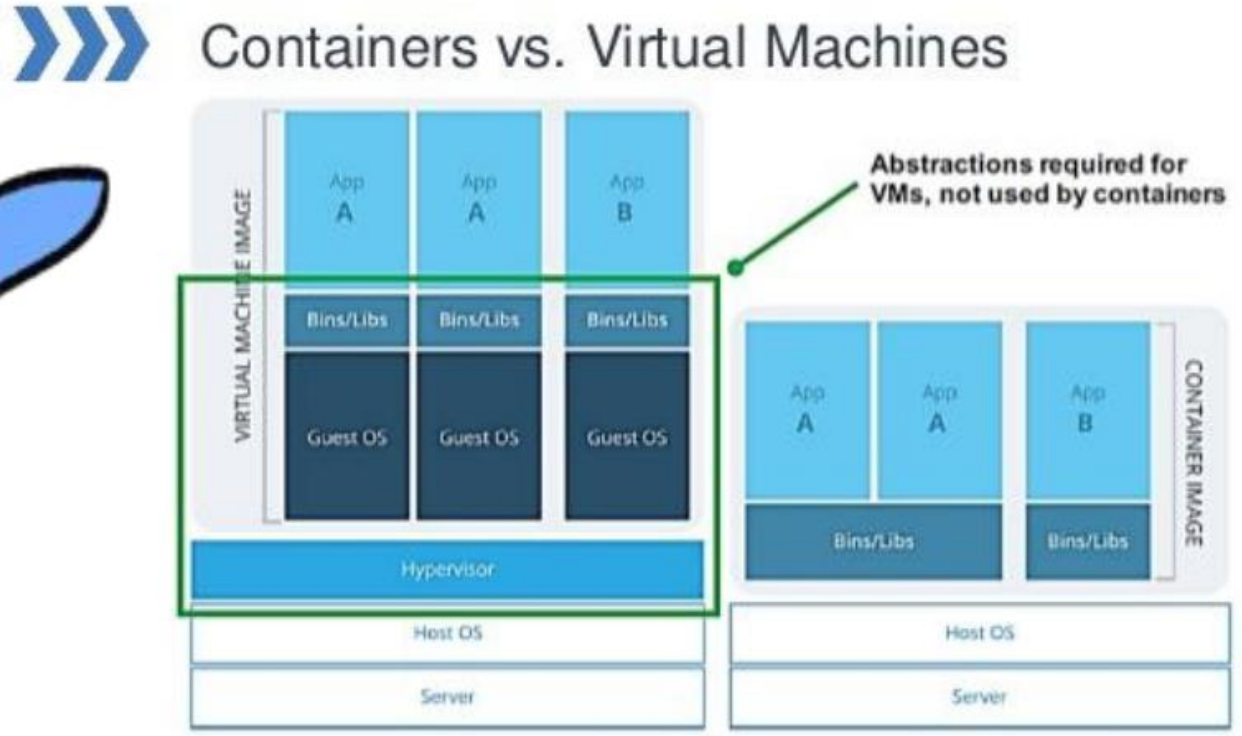
- Clair
- Trivy
- Anchore Enginer
- Synk Container
- OpenSCAP

# Where are we ?

- Introduction
- The Foundation Technologies (Enablers)
  - Virtualization
  - SOA
  - Data Center
  - Grid Computing
  - Utility Computing & pay-as-you-go model
- The Core Technologies (Building Blocks)
  - **Virtualization**
  - Containers
    - Docker
    - Kubernetes 
  - **Orchestration**
  - Storage Systems
  - **Networking**
- Cloud Security Technologies
- Emerging & Advanced Cloud Tech

# Recap on containerization





# Limitations of Docker alone

- While **Docker** is excellent for **creating and running individual containers**, it has **limitations** when managing **multiple containers across multiple machines** :
  1. **No Built-in Orchestration**
     - If one container fails, Docker doesn't automatically restart or replace it on another host.
     - There's no automated load balancing, scaling, or scheduling across multiple servers.
  2. Manual Scaling
     - There's no concept of **auto-scaling** based on traffic or CPU load.
  3. **Limited Self-Healing**
     - Docker can't **recreate the container on a healthy node** or ensure a **desired number of replicas** are always running.
  4. **No Load Balancing Across Hosts**
  5. Poor Multi-Host Management
     - Docker was not designed to manage multiple machines as one logical cluster.
     - You must manually configure networking, security, and storage across hosts.

# Limitations of Docker alone

- Docker is great for **containerization** (building and running containers), but not for **orchestration** (managing containers in production at scale).

- Kubernetes was created to **solve these challenges** — offering automation, resilience, and scalability for containerized applications.

# What is Kubernetes ?

- **Kubernetes** (often abbreviated as **K8s**) is an **open-source platform for automating the deployment, scaling, and management of containerized applications**.
- It was originally designed by **Google** (based on their internal system *Borg*) in 2014 and is now maintained by the **Cloud Native Computing Foundation (CNCF)**.
- The name comes from the **Greek word for "helmsman" or "pilot"**, symbolizing how Kubernetes steers your application infrastructure.
- The ship wheel  in the Kubernetes logo has **seven spokes**, referencing the abbreviation **K8s** (the 8 letters between K and s).

# What is Kubernetes ?

- For a developer, Kubernetes provides a manageable execution environment for deploying, running, managing, and orchestrating containers across clusters or clusters of hosts.
- For devops and administrators, Kubernetes provides a complete set of building blocks that allow the automation of many operations for managing development, test, and production environments.
- Kubernetes is like an **operating system for your data center or cloud**.
  It manages containers (like Docker containers) running across multiple servers, ensuring that:
  - Applications **run reliably**,
  - **Resources are balanced**,
  - And services remain **available** even if some machines fail.

# Key responsabilities

1. **Container Orchestration**
   Automatically schedules and runs containers on available machines.
2. **Load Balancing & Service Discovery**
   Exposes your containers to the network and distributes traffic evenly.
3. **Scaling**
   Increases or decreases the number of running containers based on load.
4. **Self-Healing**
   Restarts or replaces failed containers automatically.
5. **Automated Rollouts & Rollbacks**
   Deploys new versions of applications safely and reverts if something fails.
6. **Configuration & Secret Management**
   Manages application configuration without rebuilding container images.

# Kubernetes vs Docker

- **Kubernetes and Docker are not direct competitors** — they **work together and complement each other**, but solve **different problems**.

**Docker: The Container Platform**

- Builds and runs containers.
- Ensures consistency between development and production.
- Provides the docker CLI and Docker Engine.
- Manages containers on a single host (local or server).

Scope : Single machine

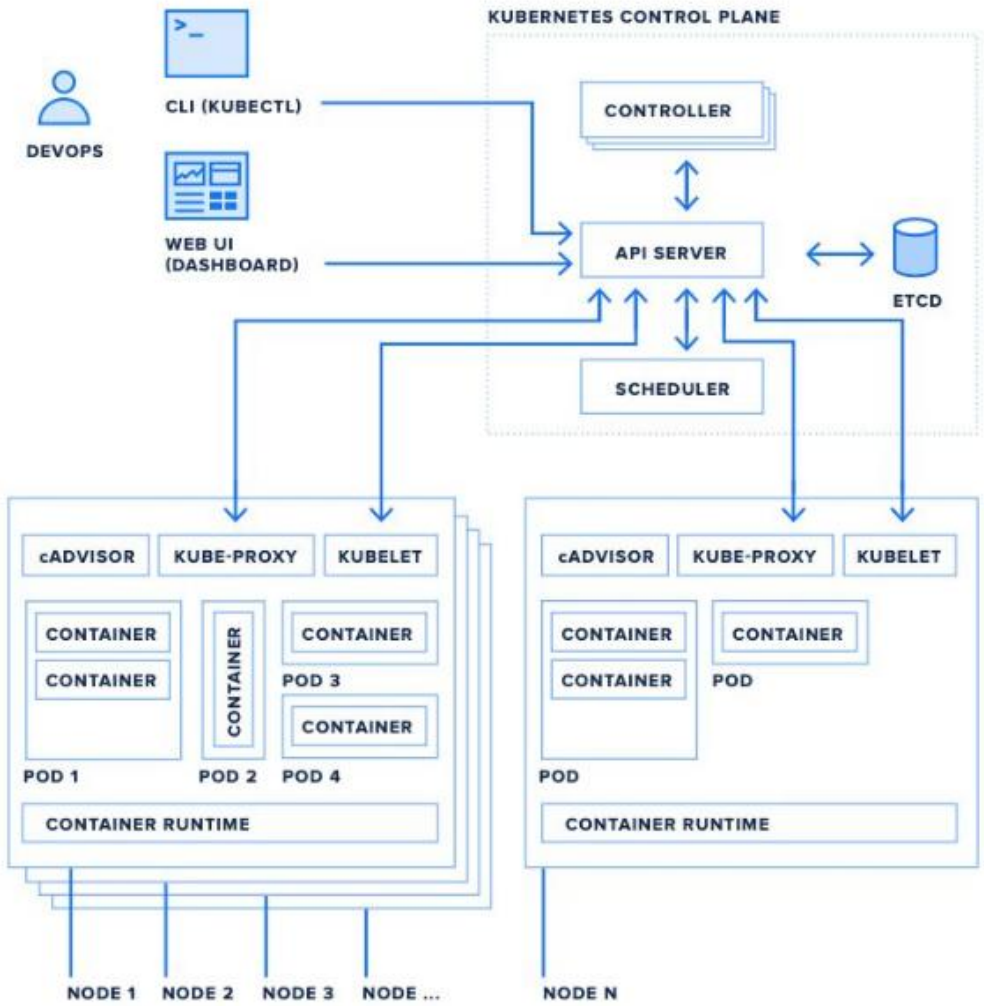
**Kubernetes: The Orchestration Platform**

- Scheduling and placement of containers across a cluster.
- Load balancing and service discovery.
- Scaling (up/down).
- Health monitoring and self-healing.
- Rolling updates and rollbacks.

Scope : Cluster of many machine

# Kubernetes Architecture

- K8s is organized as a cluster
- Client-Server architecture
- With 2 main components
  - Server: Control Plane (master node)
  - Clients: Worker Nodes



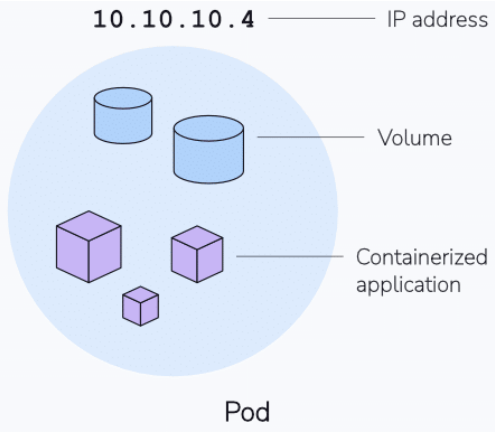
# Kubernetes Architecture
## Workloads

Kubernetes includes several abstraction layers that you use to define your application :

- Pod
- Deployment
- Service
- Job
- DaemonSets and StatefulSets

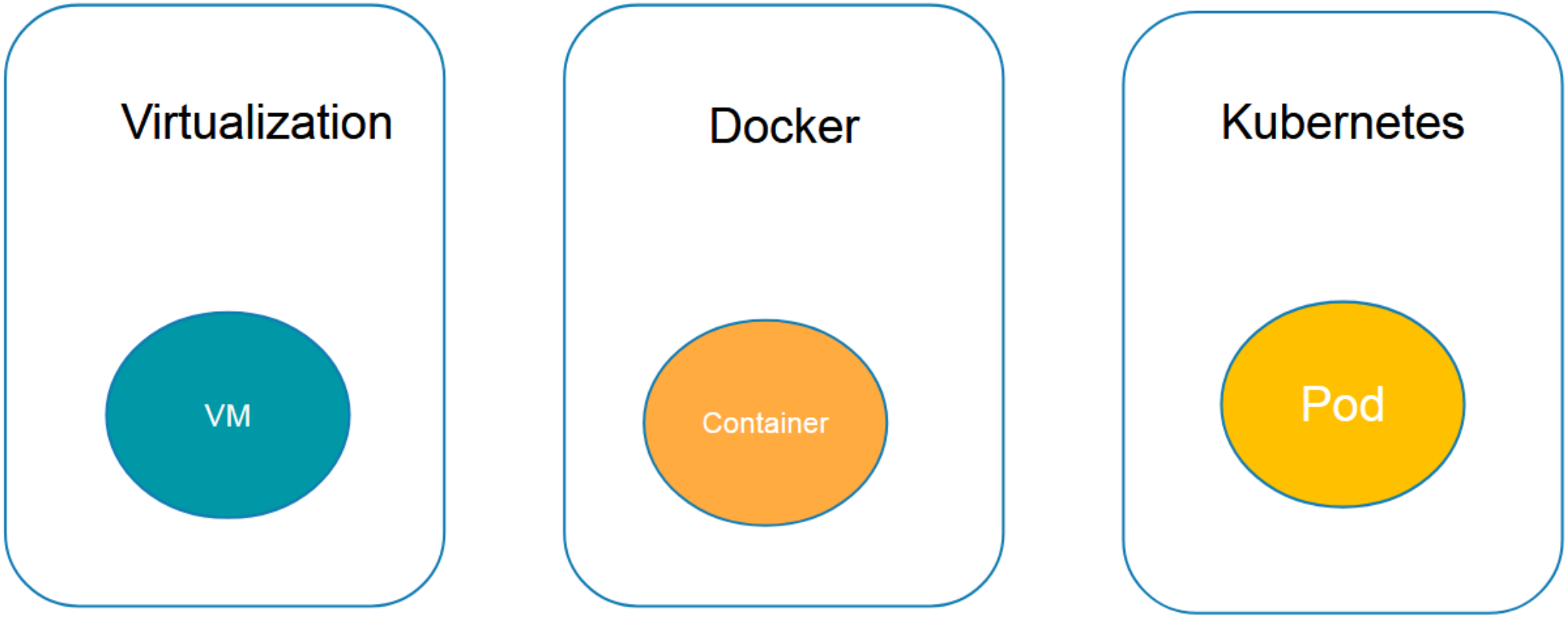
# Kubernetes Architecture
## Pods

- Kubernetes does **not run containers directly** — it runs **Pods**, which encapsulate containers.



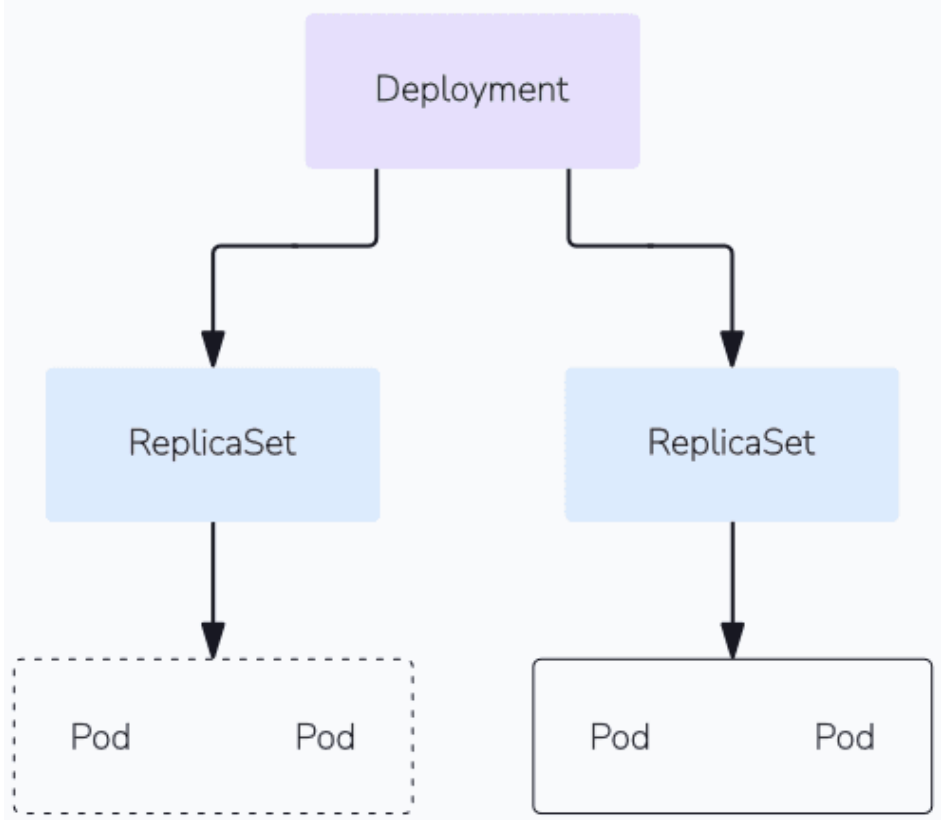

- A **Pod** is the smallest unit in Kubernetes.
- It can contain **one or more containers** that share the same network and storage.
- All the containers in the Pod are scheduled to the same host.

# Atomic Unit of Scheduling



# Kubernetes Architecture
## Deployment

- A Deployment wraps the lower-level ReplicaSet object.
- It guarantees a certain number of replicas of a Pod will be running in your cluster.
- Deployments also provide declarative updates for Pods;
  - you describe the desired state, and the Deployment will automatically add, replace, and remove Pods to achieve it.

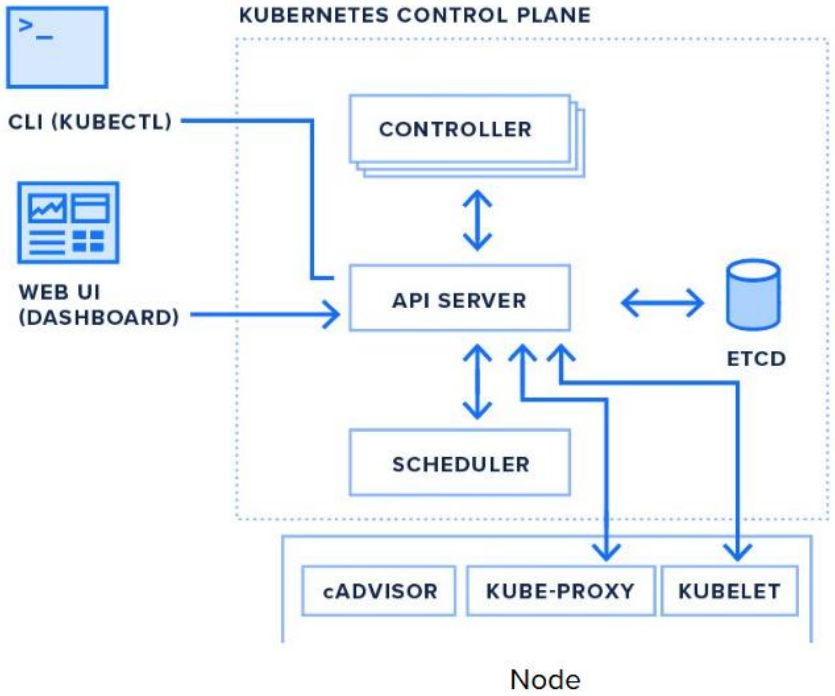


# Kubernetes Architecture
## Services, Jobs, DaemonSets and StatefulSets

- **Service :** is an entity that represents a set of pods that run an application or functional component.
  - Services expose Pods as a network service.
  - You use services to permit access to Pods, either within your cluster via automatic service discovery, or externally through an Ingress.
- **Job**
  - A Job starts one or more Pods and waits for them to successfully terminate.
  - Kubernetes also provides CronJobs to automatically create Jobs on a recurring schedule.
- **DaemonSets and StatefulSets**
  - Other kinds of workloads include DaemonSets and StatefulSets.
  - DaemonSets replicate a Pod to every Node in your cluster, while StatefulSets provide persistent replica identities.

# Kubernetes Architecture
## Control Plane

- Responsible for **managing the cluster**, deciding **what** should run, **where**, and **when**.
- Maintains the desired state of the cluster (continuously compares the current state of the cluster e.g., the number of pods running)
- **Components :**
  - API server
  - Scheduler
  - Controllers
    - Kubernetes
    - Cloud
  - Etcd

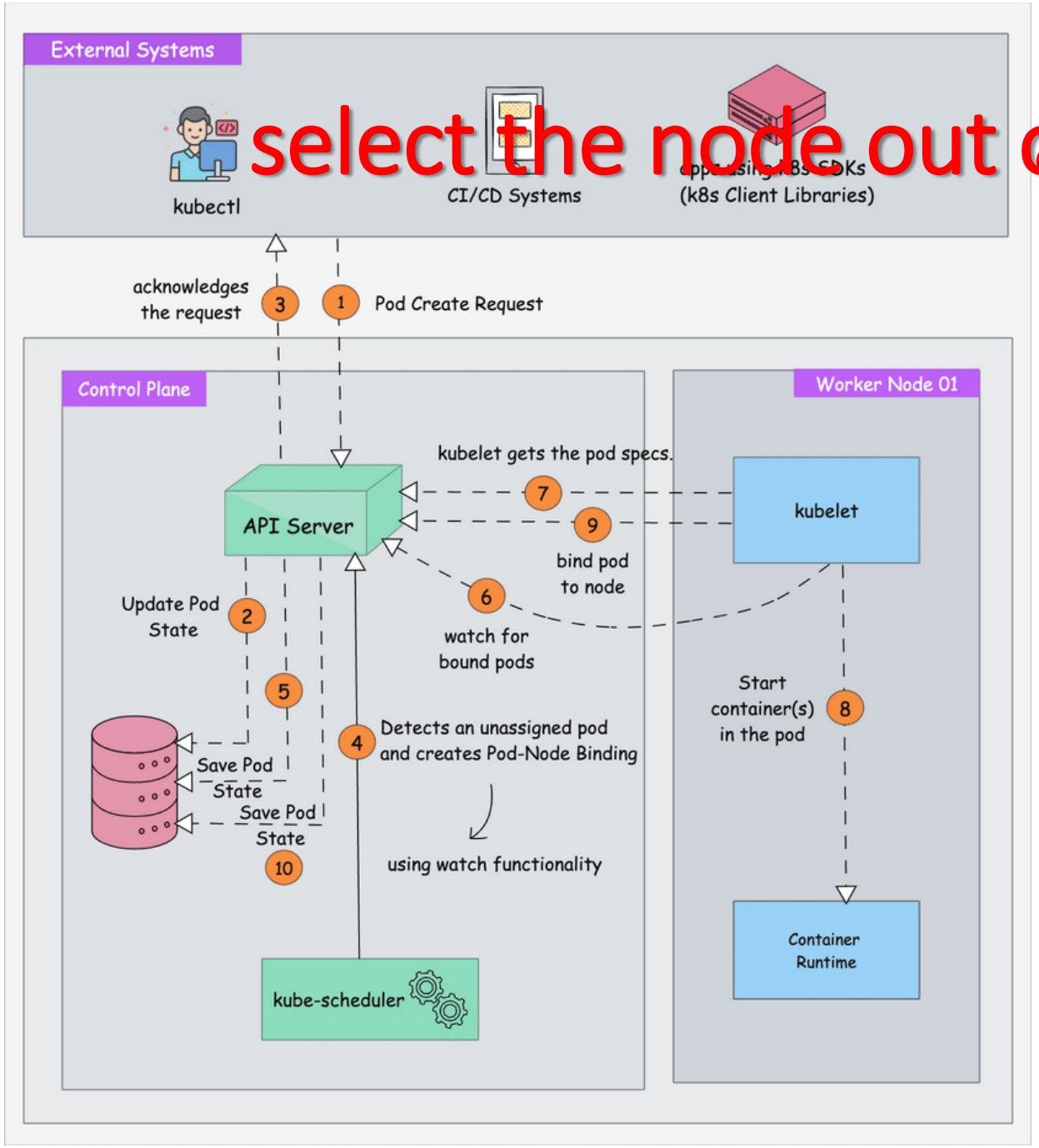


# Scheduling Process

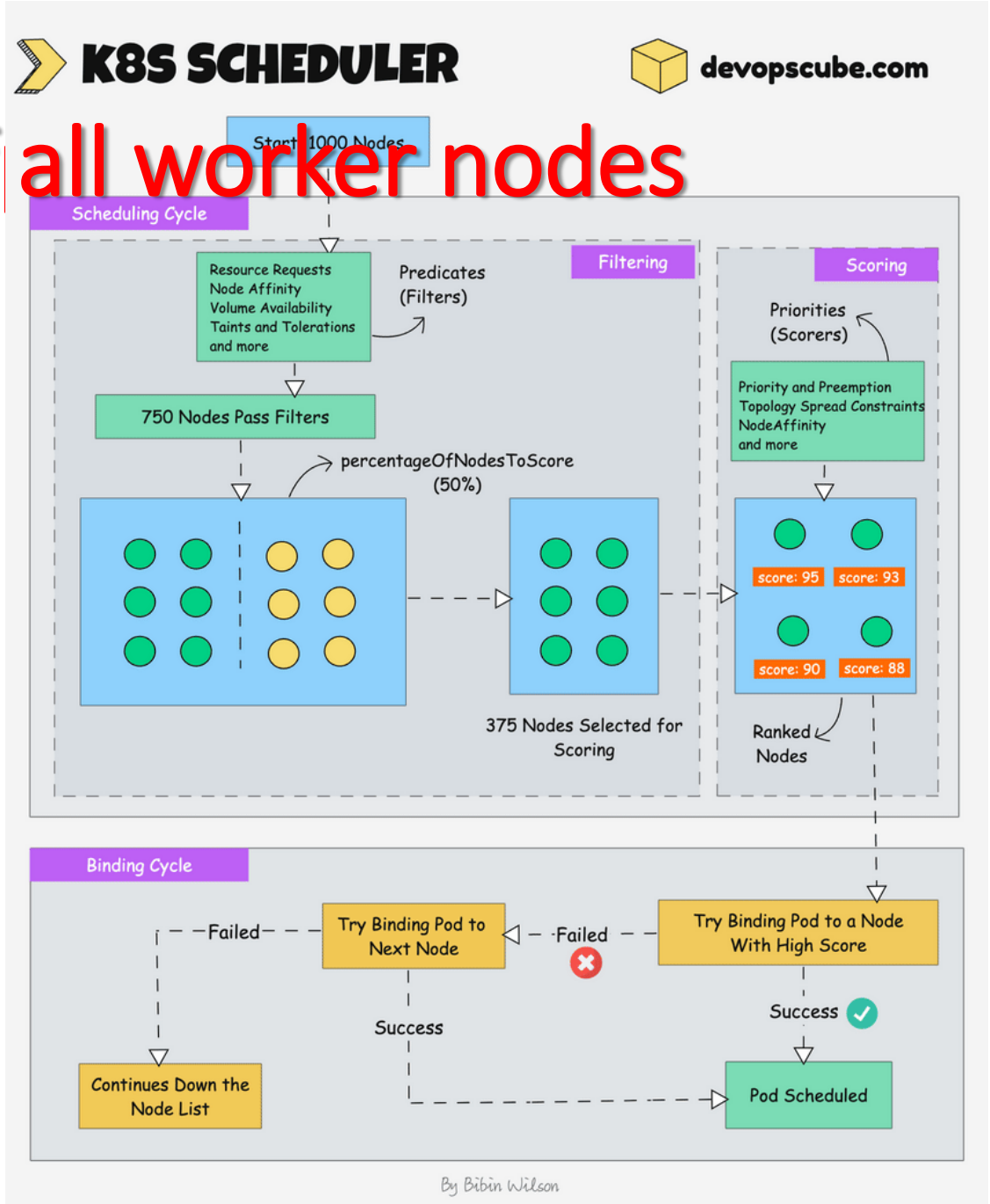
1. When a pod is created, it enters the pending state if no nodes are available for scheduling.

2. The scheduler filters out nodes that do not meet the pod's requirements (e.g., resource limits)

3. The remaining nodes are ranked based on various factors (uses a priority function to assign a score to the nodes on a scale from 0 to 10)

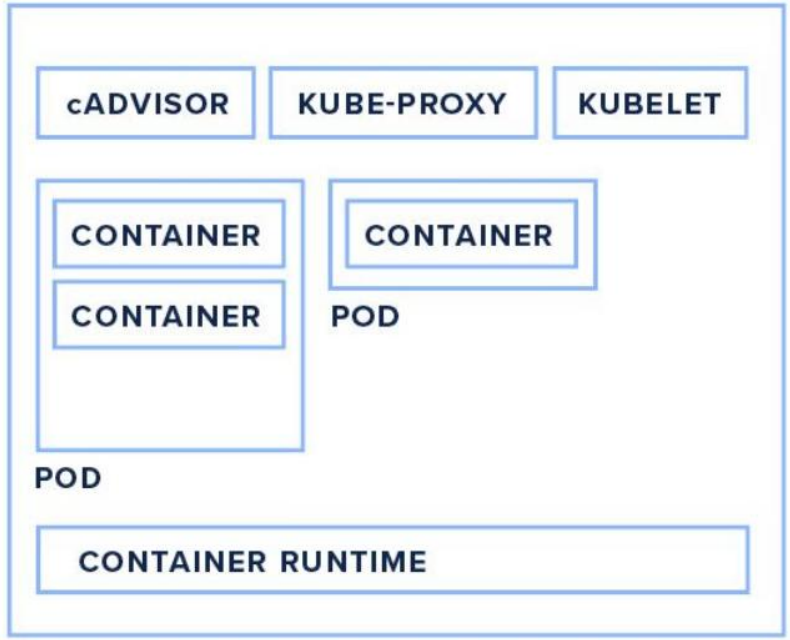



select the node out of all worker nodes



# Kubernetes Architecture
## Nodes

- The **machines** (physical or virtual) that **run the application containers**.
- Each worker node is managed by the control plane.
- They provide the necessary resources (CPU, memory, storage) for executing these applications.
- **Components** :
  - Kubelet : agent for the control plane.
  - Kube-proxy : the network communication point
  - cAdvisor
  - Container runtime : like Docker Engine



# How It Works
## Example

1. You submit a deployment file (YAML/JSON) using :
   kubectl apply -f app.yaml
2. The API Server receives the request.
3. The Scheduler selects the most appropriate node(s) for the Pods.
4. The Controller Manager ensures the Pods are running as desired.
5. The Kubelet on each node communicates with the control plane and starts the containers via the container runtime.
6. The Kube Proxy manages network routing between Pods and Services.

# Example

- To better understand Kubernetes architecture, imagine a restaurant where various roles and operations come together to provide a smooth dining experience.

- In this analogy, the Kubernetes cluster functions like a restaurant, with its management and staff working together to deliver services efficiently.

# Example
## The Analogy

1. **Kubernetes Cluster = Restaurant**
2. **Master Node (Control Plane) = Restaurant Management**
3. **API Server = Front Desk**
   - *Role in the Restaurant:* The front desk is the first point of contact, where customer orders are taken and directed to the right place.
   - *Role in Kubernetes:* The API server is the main entry point to the Kubernetes cluster. It validates, processes, and distributes requests to the appropriate components.
4. **Controller Manager = Restaurant Manager**
   - *Role in the Restaurant:* The restaurant manager monitors the workflow, ensuring enough food is prepared, and adjusting staffing levels as needed.
   - *Role in Kubernetes:* The controller manager monitors the cluster and makes sure the desired state is maintained. It adds or removes resources to keep everything running smoothly.

# Example
## The Analogy

5. **Scheduler = Kitchen Dispatcher**
   - *Role in the Restaurant:* The dispatcher assigns orders to chefs based on the complexity of dishes and the chef's current workload.
   - *Role in Kubernetes:* The scheduler assigns new workloads (pods) to the most appropriate nodes, optimizing resource use and balancing the load.
6. **Etcd = Restaurant Details**
   *Role in the Restaurant:* Tracks the cash details, staff details, dishes details, supplies details and all.
   *Role in Kubernetes:* Etcd stores the cluster's configuration and state, keeping a record of all desired states and enabling consistency across nodes.
7. **Worker Nodes = Restaurant Staff**
   - Each worker node represents the restaurant's staff who carry out tasks like preparing food and serving customers.

# Example
## The Analogy

**8. Kubelet = Chef**

- *Role in the Restaurant:* The chef follows instructions from the front desk and prepares dishes according to the orders.
- *Role in Kubernetes:* Kubelet ensures that the containers (workloads) assigned to the worker node are running and reports their status back to the API server.

**9. Kube-Proxy = Waitstaff**

- *Role in the Restaurant:* The waitstaff ensures that food is delivered to the right table and manages the flow of orders.
- *Role in Kubernetes:* Kube-Proxy routes network traffic to the correct pods, managing internal communication and external access to services.

# Example
## The Analogy

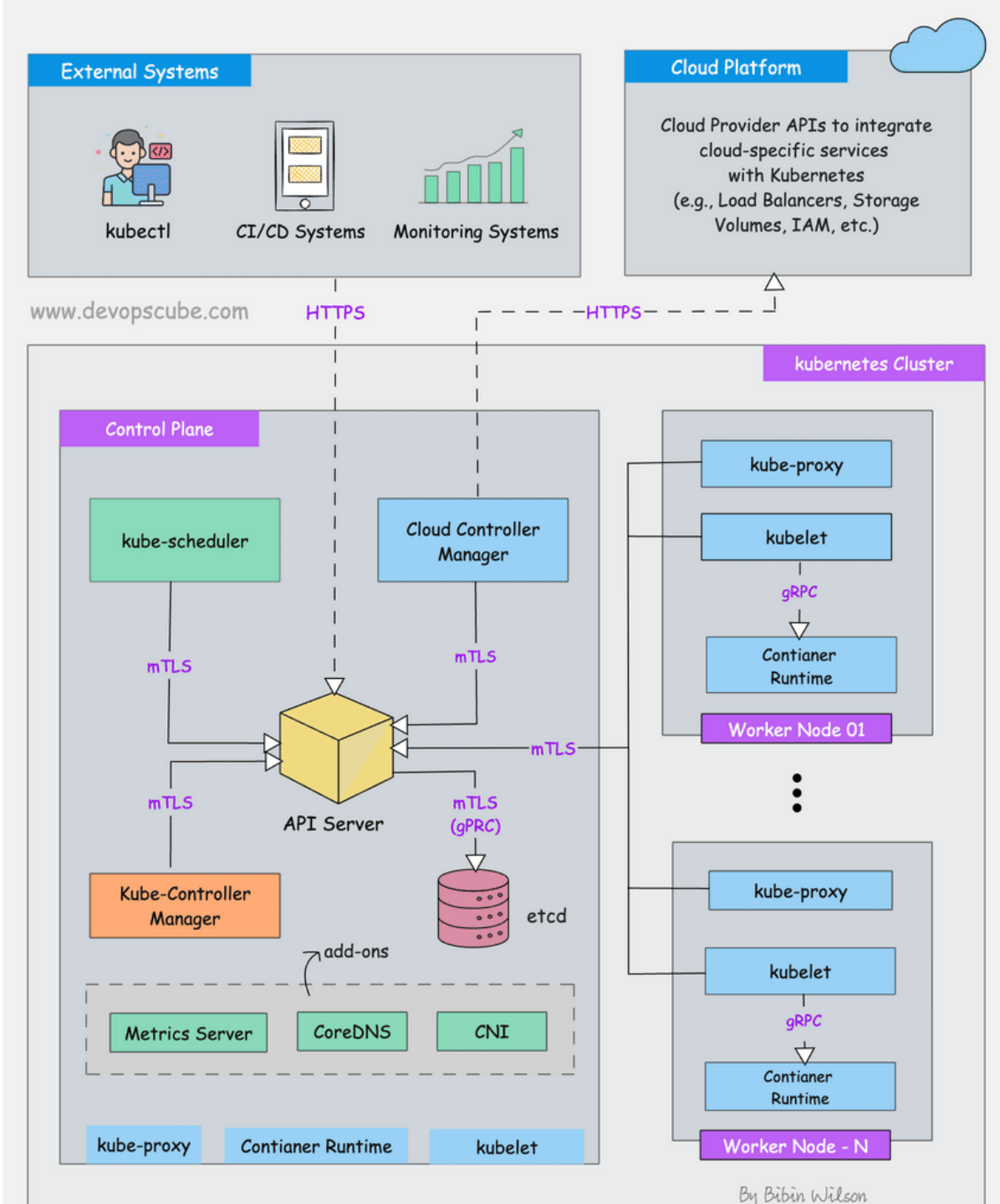
**10. Container Runtime = Kitchen**

- *Role in the Restaurant:* The kitchen is where the ingredients are transformed into dishes.
- *Role in Kubernetes:* The container runtime (like Docker) runs containers on the worker nodes, executing workloads based on the instructions from the API server.

**11. Services = Menu**

- *Role in the Restaurant:* The menu provides customers with a list of available dishes and directs them to what they can order.
- *Role in Kubernetes:* Services abstract a group of pods, providing stable endpoints for accessing them. They facilitate load balancing and routing, similar to how the menu organizes available dishes.

# Kubernetes Architecture :
## A Summary



# Common kubectl Commands

- Kubectl is a command line tool with which one runs commands against Kubernetes clusters.
- It is used for deploying applications, viewing logs, and managing and inspecting cluster resources.

| PURPOSE | COMMAND |
|---|---|
| List of nodes and pods | kubectl get pod -o wide |
| Show labeled nodes | kubectl get nodes -show-labels |
| Obtain pod resource use | kubectl top pod |
| Add a label by hand to a pod. | kubectl label pods <podname> owner=gfg |
| Obtain a certain hidden field of secret. | kubectl get secret GFG-cluster-kubeconfig |
| List all services | kubectl get services |
| List storage class | kubectl get storageclass |
| Get service detail | kubectl get service <servicename> -o yaml |