# Lecture 1 (INT 332) BASICS OF DEVOPS

## Revolutionizing Containerization

# Why Devops

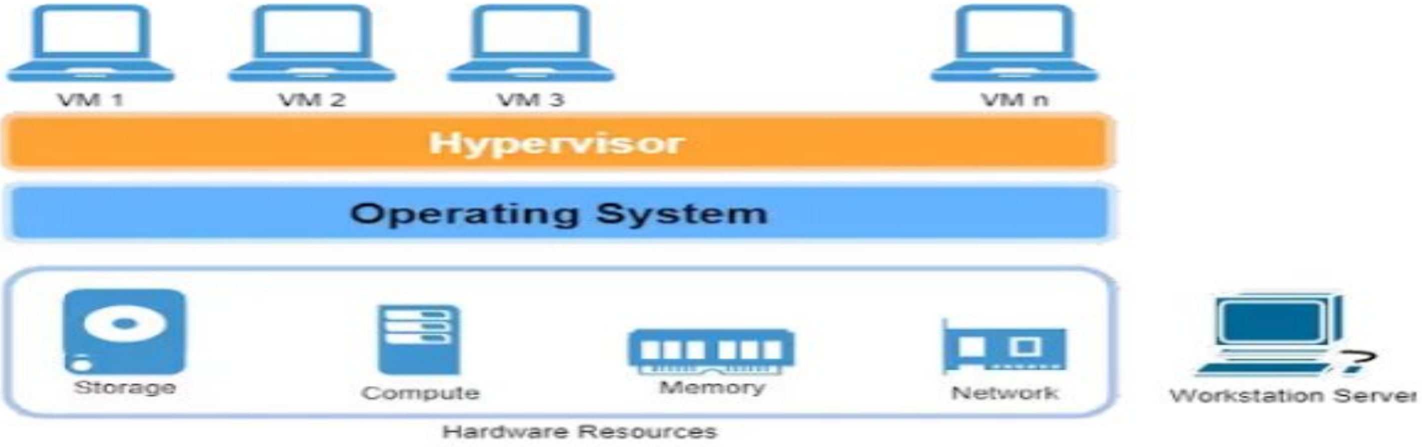

urban {code}
test.ai
Dev
Ops
JFrog
Rx
ACCELQ
Comprehensive List of DevOps Tools 2024
appvance.ai
A Gamut of DevOps Tools
testim

# Understanding the concepts of Virtulization and Containerization

Imagine a peaceful independent house.

# The Apartment Era (Virtualization)

Infrastructure followed the same path.




One physical server was divided into multiple **virtual machines.**
Each VM felt like its own house — with its own OS, memory, and CPU.

Utilization improved.
Costs reduced.

# Virtualization

- **With the changing times, businesses started looking for solutions to reduce overhead costs, enhance scalability, and standardize application deployment process.**

- **They started considering the following two approaches to reduce costs −**

1. **Virtualization**
2. **Containerization**

# What is Virtualization?

- Virtualization is the process of partitioning a physical server into multiple virtual servers.

- The process of partitioning is carried out using a software called 'hypervisor'.

- After partitioning, the virtual servers act and perform just as a physical server.

- Essentially, it means using the same hardware set-up more efficiently, thereby freeing the resources for other tasks or retiring the resources altogether.

# Advantages of Virtualization

- Enhanced performance
- Promotes agile IT infrastructure
- Promotes use of resources in optimum manner
- Better disaster management if any physical hazards take place to resources
- Better security as the infected VM can be isolated from other VMs and the host server
- Space saving
- The capital investment cost on hardware is saved, and thereby maintenance cost is saved, hence overall cost savings for a business.

# Drawbacks of Virtulaization which lead to concept of containerization

1. **Heavy Resource Overhead**
2. **Slow Boot Time and Poor Agility**
3. **Inefficient Scalability**
4. **Limited Application Portability**
5. **Higher Management and Maintenance Complexity**
6. **Increased Infrastructure Cost**

**These drawbacks directly led to the evolution of containerization, which offers lightweight isolation, faster deployment, better portability, and seamless integration with DevOps and microservices architectures.**

**PG Living (Containerization)**

High rent.
Fast life.
Limited space.

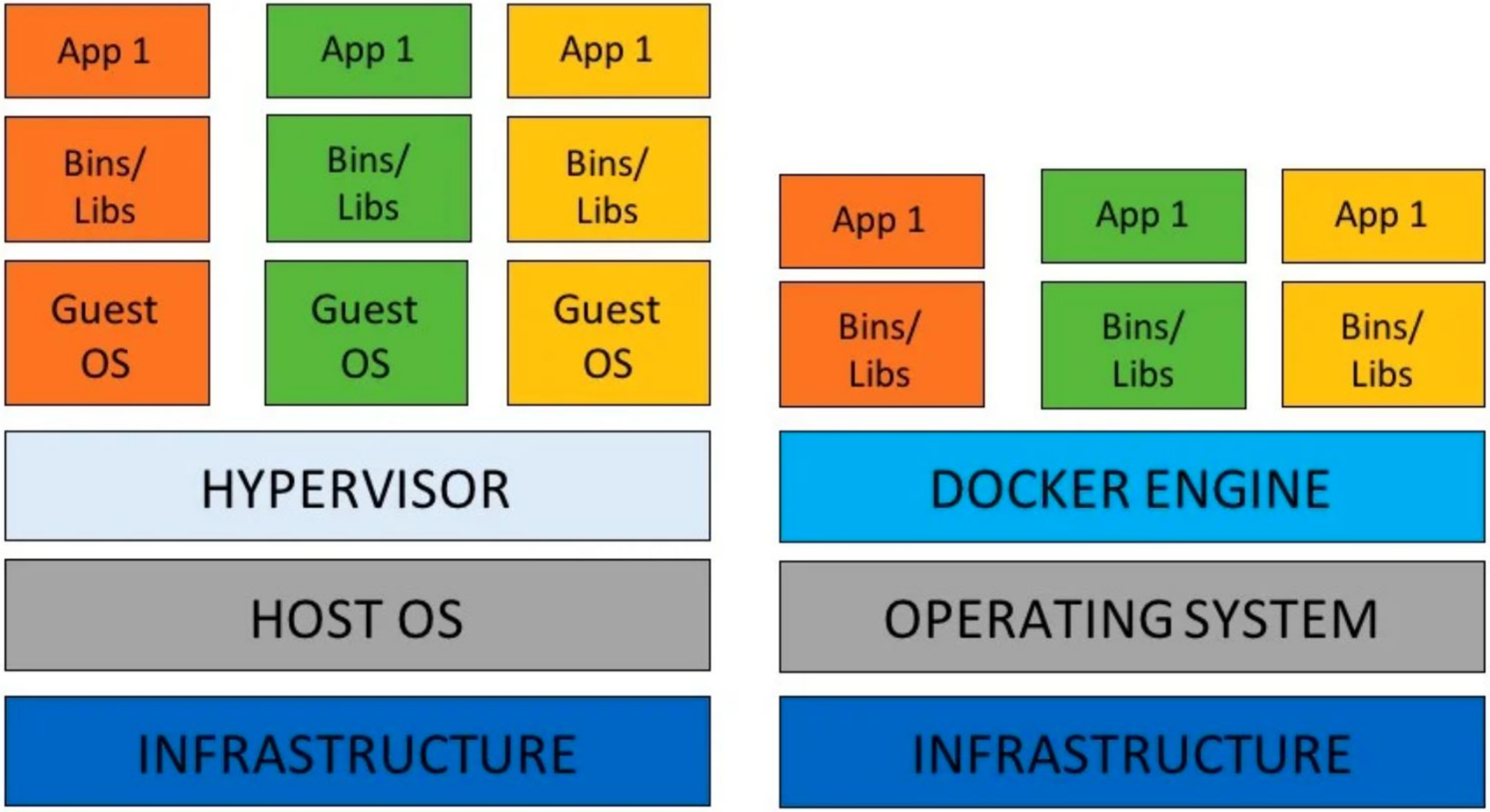
People started choosing **PGs and hostels**.

You don't own the building.
You don't manage electricity.
You don't worry about maintenance.

# That’s exactly what containers do.

- Containers don’t carry an entire operating system. They use bare minimum OS.
- They share the host OS and only bring what the application needs.
- They start in seconds. They consume minimal resources.
- You can run dozens — even hundreds — on the same machine.
- Earlier the applications were monolithic , so physical servers were apt,
- later there were components split like backend, frontend, db ,so VM’s took care of hosting.
- Now all the applications are turning to microservices, so the containerization tools like docker will serve the purpose.
- Also the portability problem is also solved using the containerization

# Need of containers



## Container Runtime

- **A container runtime is the software responsible for creating, starting, stopping, and managing containers on a host system.**

- **It acts as the execution engine that turns a container image into a running container.**

**In simple words:**

***"Container runtime is what actually runs containers."***

# Need for Container Runtime

- A container image is just a read-only package (application + dependencies).

To run it, the system needs a runtime to:

- Create an isolated environment
- Apply resource limits
- Start the application process
- Monitor and stop it when required

# Types of Container Runtimes

## 1. High-Level Runtimes

**Used by users and DevOps tools.**

**Docker Engine**

**containerd**

**CRI-O (Kubernetes-focused)**

**These handle:**

- **Image pulling**
- **Networking**
- **Storage**
- **Container lifecycle**

# 2. Low-Level Runtimes

- Responsible for actually running the container process.
- runc (OCI-compliant)

These directly interact with:

- Linux kernel
- Namespaces
- cgroups

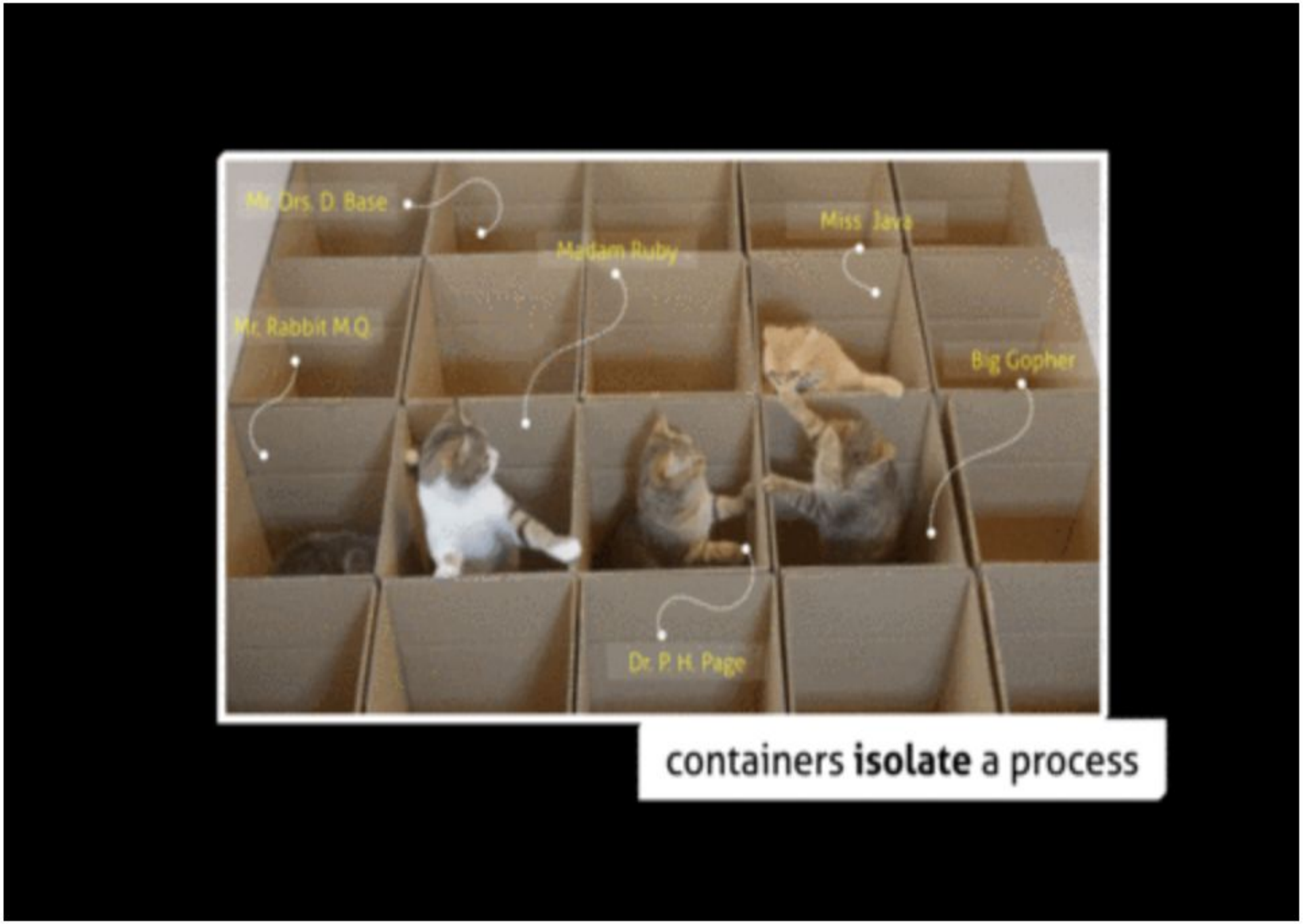
## Real-World Analogy

- Container Image → Packed lunch box
- Container Runtime → Person who opens it, serves food, and cleans up

# Process Isolation & Namespaces

- **Process isolation ensures that each container runs as if it is alone on the system, even though multiple containers share the same OS kernel.**

- **This isolation is achieved using Linux namespaces.**



Mr. Drs. D. Base
Madam Ruby
Miss Java
Mr. Rabbit M.Q.
Big Gopher
Dr. P. H. Page
containers **isolate** a process

## Interpretation of Picture

- As shown is this little cat picture, containers are isolated but using networking they still **discover** each other and consume the services if needed.

# Namespaces

- **Namespaces are Linux kernel features that partition system resources so that processes see only what belongs to them.**

- ***"Namespaces create the illusion of a separate system for each container."***

# 1. PID Namespace (Process Isolation)

- Each container has its own process tree
- Process IDs inside container start from 1
- A container cannot see host or other container processes

E.g: Inside a container, the main app runs as PID 1, just like init/systemd in a VM.

# 2. Network Namespace

- **Each container gets:**
  - Its own IP address
  - Its own ports
- **Containers can run apps on the same port without conflict**

- **Example:**
  **Multiple Nginx containers all listen on port 80 internally.**

## 3. Mount Namespace

- Each container has its own filesystem view
- File changes inside container do not affect host

## 4. UTS Namespace

- Allows each container to have its own hostname

## 5. User Namespace (Advanced)

- Maps container users to non-root users on host
- Improves security

# Real-World Analogy behind Namespaces

**Namespaces are like separate cabins in a train:**

- Same engine (kernel)
- Different passengers
- No visibility between cabins

# Control Groups (cgroups) for Resource Limits

**Control Groups (cgroups) are Linux kernel features that limit, control, and monitor resource usage of processes.**

**They ensure that:**

- **No container can monopolize system resources**
- **Fair sharing of CPU, memory, and I/O**

***"Namespaces isolate, cgroups control."***

# Need of cgroups

**Without cgroups:**

- **One container could consume all CPU or RAM**
- **Host and other containers could crash**

# Types of Resources Controlled by cgroups

1.CPU

- Limit CPU usage
- Assign CPU shares

2. Memory

- Set maximum memory usage
- Kill container if limit exceeded

3. Disk I/O

- Limit read/write speeds

4. Network (Indirectly)

- Through traffic shaping mechanisms

# Real-World Analogy

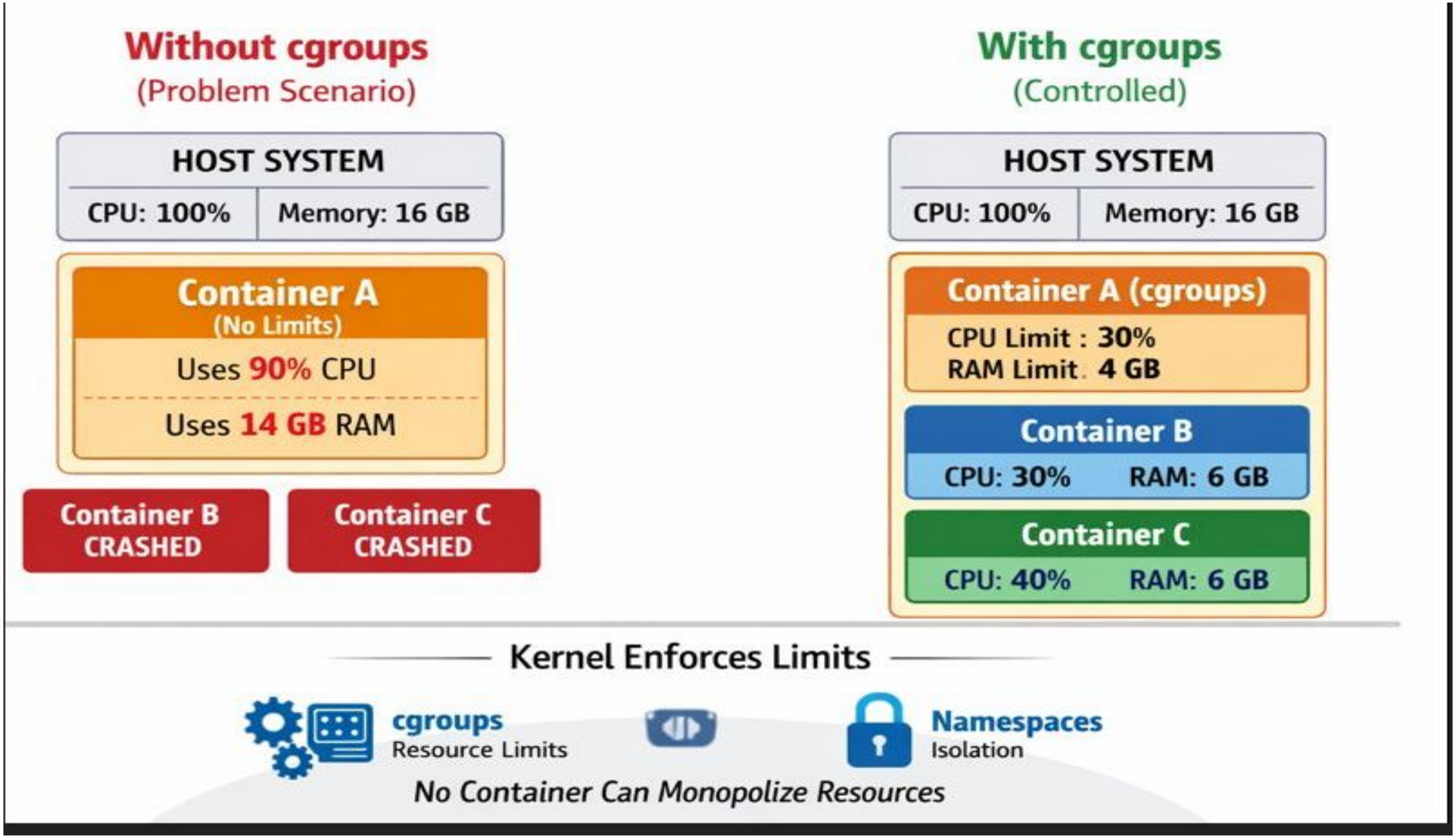
**cgroups are like electricity meters in apartments:**

- **Each flat has a limit**
- **One tenant cannot use all power**

# Finally for namespaces and cgroups

Containers are lightweight because they do not virtualize hardware.

Instead, a container runtime uses Linux namespaces for process isolation and cgroups for resource control, allowing multiple isolated applications to run efficiently on a shared operating system kernel.



Without cgroups
(Problem Scenario)
HOST SYSTEM
CPU: 100%
Memory: 16 GB
Container A
(No Limits)
Uses 90% CPU
Uses 14 GB RAM
Container B CRASHED
Container C CRASHED
With cgroups
(Controlled)
HOST SYSTEM
CPU: 100%
Memory: 16 GB
Container A (cgroups)
CPU Limit : 30%
RAM Limit. 4 GB
Container B
CPU: 30% RAM: 6 GB
Container C
CPU: 40% RAM: 6 GB
Kernel Enforces Limits
cgroups
Resource Limits
Namespaces
Isolation
No Container Can Monopolize Resources

# Today’s Task

**Task 1: A company uses containers for microservices. One faulty container crashes the host due to high memory usage.**

**Questions:**

- **Which container feature was misconfigured or missing?**
- **Which kernel mechanism should have prevented this?**
- **Would namespaces alone solve this issue? Why?**

**Task 2: A server is running three containers:**

- **Container A runs a CPU-intensive task**
- **Container B runs a web server**
- **Container C runs a database**

**Questions:**

- **Which Linux feature prevents Container A from consuming all CPU?**
- **Which feature ensures Container B cannot see processes of Container C?**
- **Which namespace allows all three containers to use port 80 internally?**

# Container Image

## A container image is:

- A read-only blueprint
- Contains:
  - Base OS (minimal)
  - Application code
  - Libraries & dependencies
  - Runtime
  - Configuration

**Images are immutable – once built, they do not change.**

# Example

**python:3.11-slim is the image**

**This image already includes:**

- **Linux OS**
- **Python 3.11**
- **Standard Python libraries**

# Image Layers

**Each image is built as a stack of layers.**

**Characteristics of Layers**

- **Each instruction in a Dockerfile = one layer**
- **Layers are:**
  - **Immutable**
  - **Cached**
  - **Reused across images**

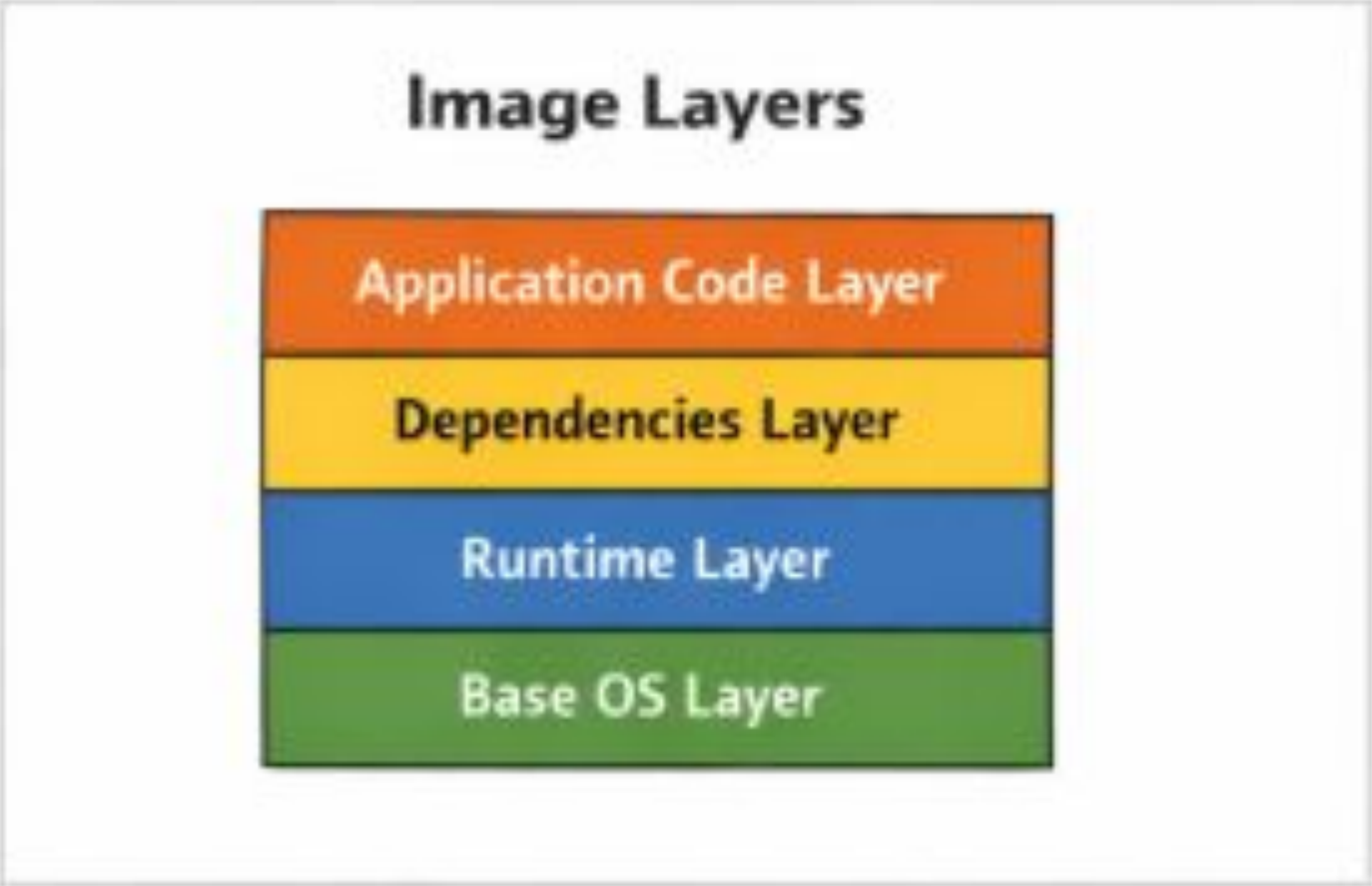


Image Layers
Application Code Layer
Dependencies Layer
Runtime Layer
Base OS Layer

# Benefits of Layered Images

## Faster Builds

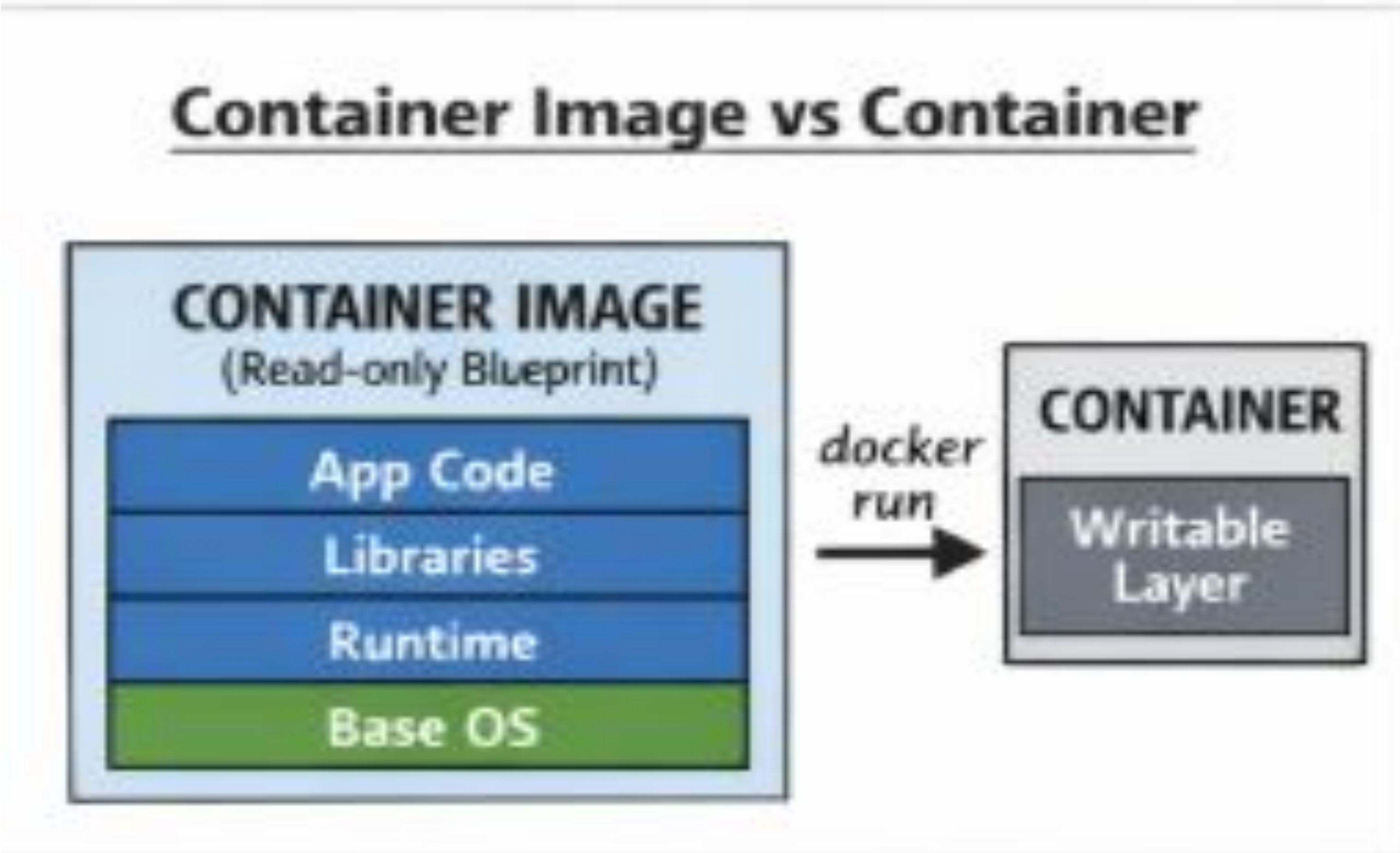
- Docker reuses unchanged layers (cache)

## Smaller Storage

- Same layers shared between images

## Faster Deployment

- Only missing layers are downloaded


Container Image vs Container
CONTAINER IMAGE
(Read-only Blueprint)
App Code
Libraries
Runtime
Base OS
docker run
CONTAINER
Writable Layer

# Image Registry

**An image registry is a centralized repository used to store, manage, version, and distribute container images.**

**It enables:**

- **Sharing images across teams and environments**
- **Consistent deployment in development, testing, and production**
- **Integration with CI/CD pipelines**

**In DevOps, an image registry plays the same role as:**

- **GitHub → for source code**
- **Maven Central → for Java libraries**

# Why Image Registries are Required

## Without a registry:

- **Each system must build images independently**
- **High inconsistency across environments**
- **Manual software installation**
- **No version control of application environments**

## With a registry:

- **Build once, deploy everywhere**
- **Faster deployments**
- **Version-controlled environments**
- **Rollback capability**
-

# Types of Image Registries

## 3.1 Public Image Registries

**Public registries allow open access to images.**

**Characteristics:**

- Images are publicly visible
- Anyone can pull images
- Often free
- Used for base images and open-source applications

# Private Image Registries

Private registries restrict access using authentication and authorization.

## Characteristics:

- Secure and controlled access
- Used in enterprises
- Stores proprietary applications
- Integrated with IAM and RBAC

## Examples:

- AWS Elastic Container Registry (ECR)
- Azure Container Registry (ACR)
- GitHub Container Registry (GHCR)
- On-premise private registry

# Image Naming Convention

**Images follow a standard naming**

**format:<registry>/<namespace>/<image>:<tag>**

**Example:docker.io/harpreet/webapp:v1**

| Component | Meaning |
|---|---|
| registry | Location of image |
| namespace | User or organization |
| image | Application name |
| tag | Version or label |

# Tags and Versioning

## Tags identify different versions of the same image.

### Common Tagging Strategies

| Tag | Purpose |
|---|---|
| latest | Default (not recommended in production) |
| v1, v2 | Versioned releases |
| dev | Development build |
| qa | Testing build |
| prod | Stable production build |

# Image Distribution Process

Image distribution refers to how images move from developers to production systems.

## Distribution Flow:

- Developer builds an image locally
- Image is pushed to a registry
- Registry stores the image securely
- Servers pull the image when required
- Containers are created from the image

# Push and Pull Operations

## Push Operation

- Uploads image layers from local system to registry
- Requires authentication
- Only new layers are pushed

## Pull Operation

- Downloads image layers from registry
- Uses caching to avoid re-downloading existing layers
- Faster on repeated deployments

# Role of Image Registries in CI/CD Pipelines

**In modern DevOps pipelines:**

- **Code is committed to Git**
- **CI system builds the container image**
- **Image is pushed to registry**
- **CD system pulls the image**
- **Application is deployed automatically**

**In Actual, Registry acts as the bridge between CI and CD.**

# Comparison Between Public and Private Registry

| Feature | Public Registry | Private Registry |
|---|---|---|
| Access | Open | Restricted |
| Security | Lower | High |
| Cost | Mostly free | Paid |
| Use Case | Learning, open source | Enterprise, production |

# QUIZ

1.What is the primary purpose of an image registry?

A. To execute containers
B. To store and distribute container images
C. To build container images
D. To monitor container performance

2. Which component of an image name represents the version of the image?

A. Registry
B. Namespace
C. Image name
D. Tag

3. Which type of image registry is typically used in enterprise environments?

A. Public registry
B. Open registry
C. Private registry
D. Anonymous registry

4. In CI/CD pipelines, the image registry mainly acts as:

A. Source code repository
B. Testing environment
C. Bridge between CI and CD
D. Container runtime

# Introduction to Docker Hub

Centralized repository for Docker images

Features:

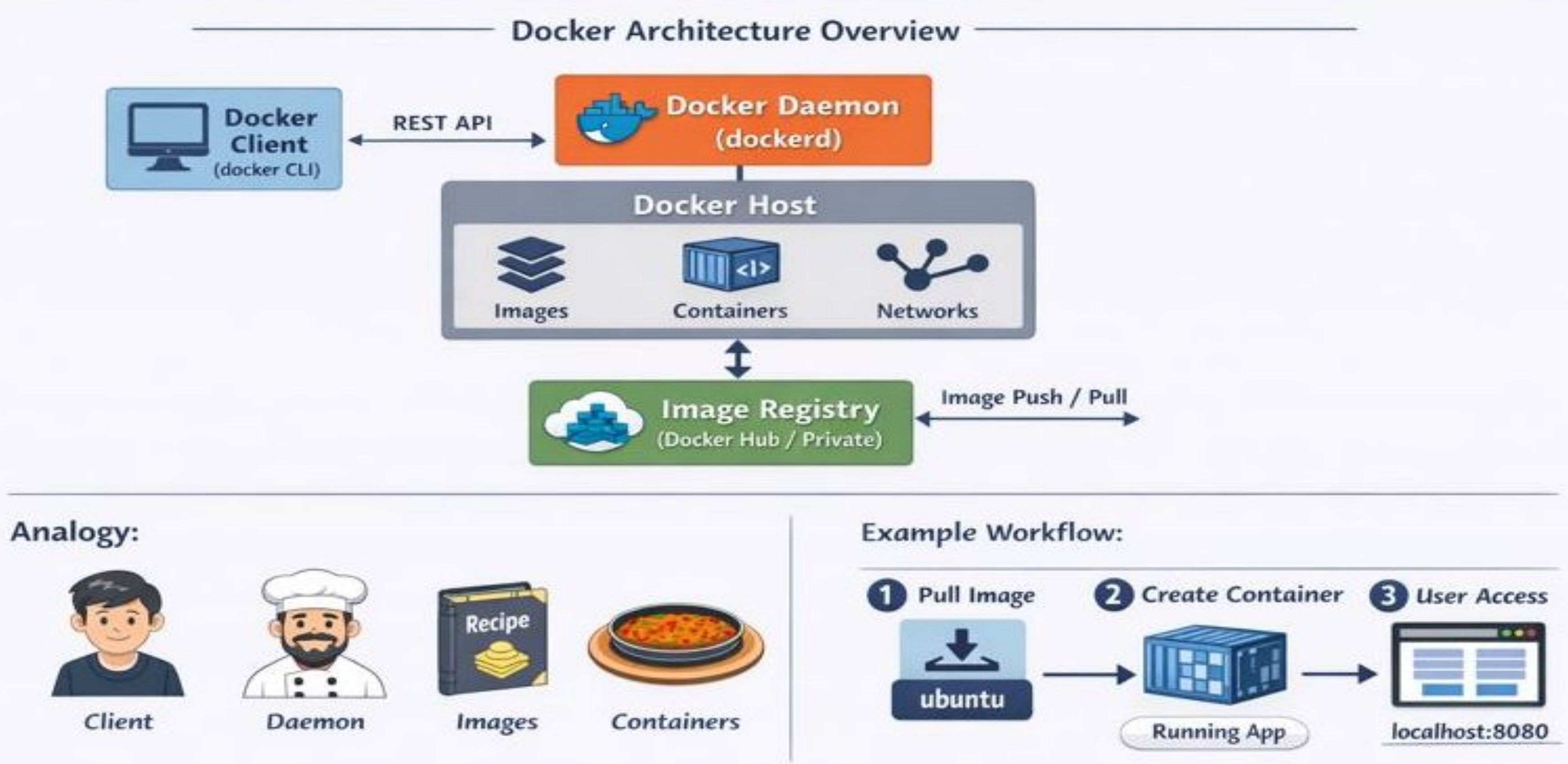
- Public and private repositories
- Official images
- Automated builds
- Importance in DevOps

# Docker Architecture

**Key Components:**

- \- Docker Client
- \- Docker Daemon
- \- Docker Images
- \- Containers
- \- Docker Registry (Docker Hub)

# Docker Architecture



# Docker Life Cycle

**Stages:**

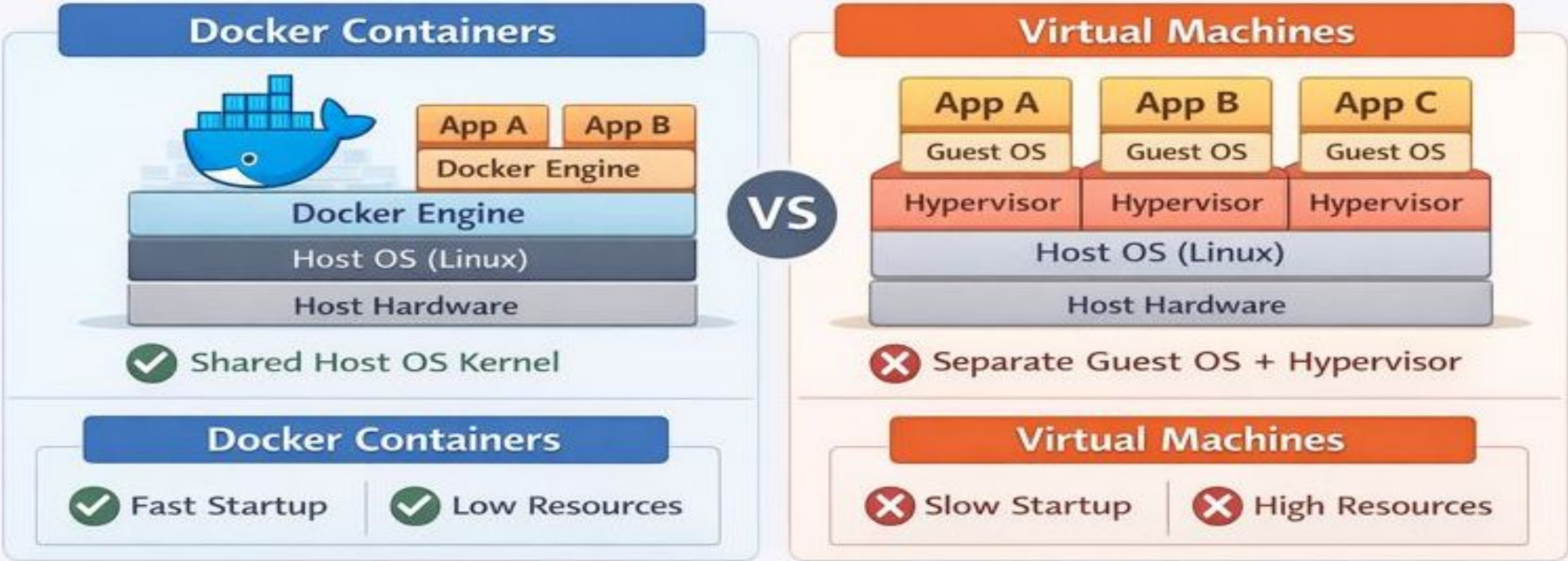
- \- Build: Create an image
- \- Ship: Share images via Docker Hub
- \- Run: Start containers

**Container states:**

- \- Created
- \- Running
- \- Paused
- \- Stopped

# Why is Docker faster than VM?

- Containers share the Host OS Kernel
- No Guest OS boot needed
- Less resource usage & faster startup



# Conclusion

**Key takeaways:**

- **Docker simplifies software deployment**
- **Docker Hub enhances collaboration**
- **Docker’s architecture and lifecycle enable efficient workflows**
- **Hyper-V integration broadens Docker’s capabilities**