**🔹 1. Containerization: Concept & Purpose**

**✅ Purpose:**

**Containerization** is a lightweight method to **package**, **deploy**, and **run applications** in a consistent environment across various systems.

* Ensures **application + all dependencies** run the same regardless of environment (dev/test/prod/cloud/on-prem).
* Solves the "it works on my machine" problem.
* Optimized for **scalability**, **portability**, and **resource efficiency**.

**🔹 2. Evolution: Physical → Virtualization → Containers**

**🖥️ A. Physical Servers (Bare Metal)**

* Single OS running directly on hardware.
* All apps share the same OS and resources.
* Issues:
  + Poor resource utilization.
  + Difficult to isolate apps.
  + Scalability is rigid.

**🧱 B. Virtualization (e.g., VMware, VirtualBox, Hyper-V)**

* Hypervisor (like **VMware ESXi** or **KVM**) abstracts physical resources.
* Multiple **Virtual Machines (VMs)** run on a host, each with its **own OS**.

**Benefits:**

* OS-level isolation.
* Can run different OSes (Windows + Linux on same host).

**Drawbacks:**

* **Heavyweight**: each VM includes a full guest OS.
* High memory and disk usage.
* **Slow boot** times (OS needs to boot).
* Lower density: fewer VMs per host.

**📦 C. Containerization (e.g., Docker, Podman, containerd)**

* Containers share the **same OS kernel** but are isolated via namespaces and cgroups.
* Only ship the **app + its dependencies** (not full OS).
* Use a **Container Runtime** instead of a hypervisor.

**Benefits:**

* **Lightweight**: no guest OS per container.
* **Faster startup** (milliseconds).
* **High density**: run hundreds of containers per host.
* **Easier portability** (build once, run anywhere).
* **Lower resource usage** per app.

**Drawbacks:**

* Share the same kernel → all containers must run compatible binaries.
* Slightly less isolation than VMs (although improving with tools like gVisor, Kata Containers).

**🔄 Visual Summary:**

| **Property** | **Virtual Machines** | **Containers** |
| --- | --- | --- |
| Abstraction Level | Hardware | OS (Kernel) |
| Startup Time | Minutes | Seconds or less |
| OS per unit | Each VM has full OS | Shared OS Kernel |
| Resource Usage | High | Low |
| Density per Host | Few | Many |
| Portability | Moderate (hypervisor-specific) | High (runs anywhere with runtime) |
| Isolation | Strong | Medium to Strong (namespace-based) |

**🔹 3. Container Tooling: Container Runtime**

**🔧 Common Container Runtimes:**

| **Runtime** | **Description** |
| --- | --- |
| **Docker** | Most popular; includes CLI, image builder, container runtime. |
| **containerd** | Core runtime (used by Docker under the hood). Supports Kubernetes CRI. |
| **CRI-O** | Lightweight runtime optimized for Kubernetes (OpenShift uses this). |
| **Podman** | Daemonless, rootless containers, compatible with Docker CLI. |
| **runc** | Low-level OCI runtime, used by containerd & Docker. |
| **Rocket (rkt)** | Now deprecated; was a competitor to Docker. |

**🔹 4. Lifecycle of a Containerized Application (Using Docker)**

Here’s how your application goes from code to running as a container:

css

CopyEdit

Source Code (Python, Node.js, Java, etc.)

↓

Dockerfile

↓

docker build

↓

Custom Docker Image

↓

docker run / container runtime

↓

Container (Running App)

**📘 Example Dockerfile (Node.js):**

dockerfile

CopyEdit

# Step 1: Base Image

FROM node:18-alpine

# Step 2: Working Directory

WORKDIR /app

# Step 3: Copy App Code

COPY . .

# Step 4: Install Dependencies

RUN npm install

# Step 5: Start App

CMD ["node", "app.js"]

**🔹 5. Real-World Benefits of Containers**

**✅ Why containers are preferred for modern app deployment:**

* **CI/CD-friendly**: Integrates well into DevOps pipelines.
* **Cloud-native**: Containers are building blocks for Kubernetes and serverless platforms.
* **Immutable infrastructure**: You ship containers, not code — consistency guaranteed.
* **Rapid scaling**: Containers can be created and destroyed quickly to meet traffic needs.
* **Microservices ready**: Ideal for decomposing monolithic apps into manageable services.

**🔹 6. Common Misconceptions**

| **Myth** | **Reality** |
| --- | --- |
| Containers are VMs | No, containers share the OS kernel — they are process-level isolation, not hardware-level. |
| Docker is required for containers | Not anymore. Kubernetes prefers containerd or CRI-O. |
| Containers are less secure | Isolation has matured; with proper security measures (AppArmor, SELinux, seccomp), they are secure. |

**🔹 Summary Table**

| **Feature** | **Virtual Machine** | **Container** |
| --- | --- | --- |
| OS Abstraction | Full OS per instance | Shared OS kernel |
| Resource Overhead | High | Low |
| Portability | Moderate | Very High |
| Startup Time | Minutes | Seconds or less |
| Scalability | Limited | High |
| Tools | VMware, Hyper-V, KVM | Docker, Podman, containerd, CRI-O |

**🧪 Lab Guide: Docker to Kubernetes (with kubectl)**

**🎯 Goal:**

Deploy a simple HTTP application (httpd) using Docker and then manage it using Kubernetes with kubectl. Understand how the command flows through the control plane (API server, etcd), and observe resource behavior.

**🧱 Prerequisites:**

1. **Kubernetes cluster** (local via Minikube, Kind, kubeadm, or cloud)
2. **kubectl** configured and connected to the cluster
3. **Docker** installed (to see docker ps interactions)
4. Basic terminal knowledge

**🗂️ Section 1: Concept - Docker Alone vs Docker + Orchestrator**

| **Feature** | **Docker Only** | **Docker + Kubernetes** |
| --- | --- | --- |
| Deployment | Manual (docker run) | Declarative (kubectl apply, run) |
| Scaling | Manual (docker run N times) | Automatic (kubectl scale, ReplicaSets) |
| Networking | Host-based | Cluster-managed (Services) |
| Monitoring | docker ps, logs, stats | kubectl get, describe, metrics |
| Self-healing | None | Yes (Pods restart on failure) |

**🚀 Section 2: Deploying httpd on Kubernetes**

**🔸 Step 1: Run your application**

bash

CopyEdit

kubectl run ramanapp --image=httpd

This creates a **Pod** with a single container running the httpd image.

* Behind the scenes:

scss

CopyEdit

kubectl → kube-apiserver → etcd (stores the state) → scheduler → kubelet → container runtime (Docker/containerd) → Pod runs

**🔍 Step 2: Observe Pod Lifecycle**

bash

CopyEdit

kubectl get pods

kubectl get pods -o wide

* View pod IP, node assignment, container image, and status.

**🧰 Step 3: Debug and Explore Resources**

**Check all available Kubernetes resources:**

bash

CopyEdit

kubectl api-resources

**Filter for container-related APIs:**

bash

CopyEdit

kubectl api-resources | grep -i con

**Note:**

There is no direct kubectl get containers because "containers" are part of **Pods**. Use:

bash

CopyEdit

kubectl describe pod ramanapp

**🧹 Section 3: Cleaning up**

bash

CopyEdit

kubectl delete pods --all

kubectl delete deploy --all

This deletes all Pods and Deployments from the current namespace (default).

💡 Pro Tip: Use -A (or --all-namespaces) to view/delete across namespaces.

**🌐 Section 4: System-Wide Observations**

**Get all Pods in all namespaces:**

bash

CopyEdit

kubectl get pods -A

kubectl get pods -A -o wide

This is useful to:

* Check if control plane Pods are running (kube-system namespace).
* Inspect CNI plugins, CoreDNS, Metrics server, etc.

**🐳 Step 5: View Docker Containers (Optional)**

Only relevant if the cluster uses Docker as the container runtime (like Minikube):

bash

CopyEdit

docker ps

This shows running containers spawned by kubelet.

**🧾 Reference Command Summary**

bash

CopyEdit

# Deployment

kubectl run ramanapp --image httpd

# Pod inspection

kubectl get pods

kubectl get pods -o wide

kubectl describe pod ramanapp

# Cluster-wide info

kubectl get pods -A

kubectl get pods -A -o wide

# Kubernetes resource types

kubectl api-resources

kubectl api-resources | grep -i con

# Cleanup

kubectl delete pods --all

kubectl delete deploy --all

# Docker container info (optional)

docker ps

**📊 Bonus: Diagram (Conceptual Flow)**

arduino

CopyEdit

kubectl run → API Server → etcd (store) → Scheduler → Controller Manager

↓

Kubelet

↓

Docker / containerd

↓

Pod (httpd)

**🧪 Optional Lab Extensions**

* Convert to Deployment:

bash

CopyEdit

kubectl create deployment ramanapp --image=httpd

kubectl expose deployment ramanapp --port=80 --type=NodePort

* Add labels and scale:

bash

CopyEdit

kubectl label pods ramanapp app=web

kubectl scale deployment ramanapp --replicas=3

* Monitor live changes:

bash

CopyEdit

watch kubectl get pods