Docker

26 July 2025 14:06

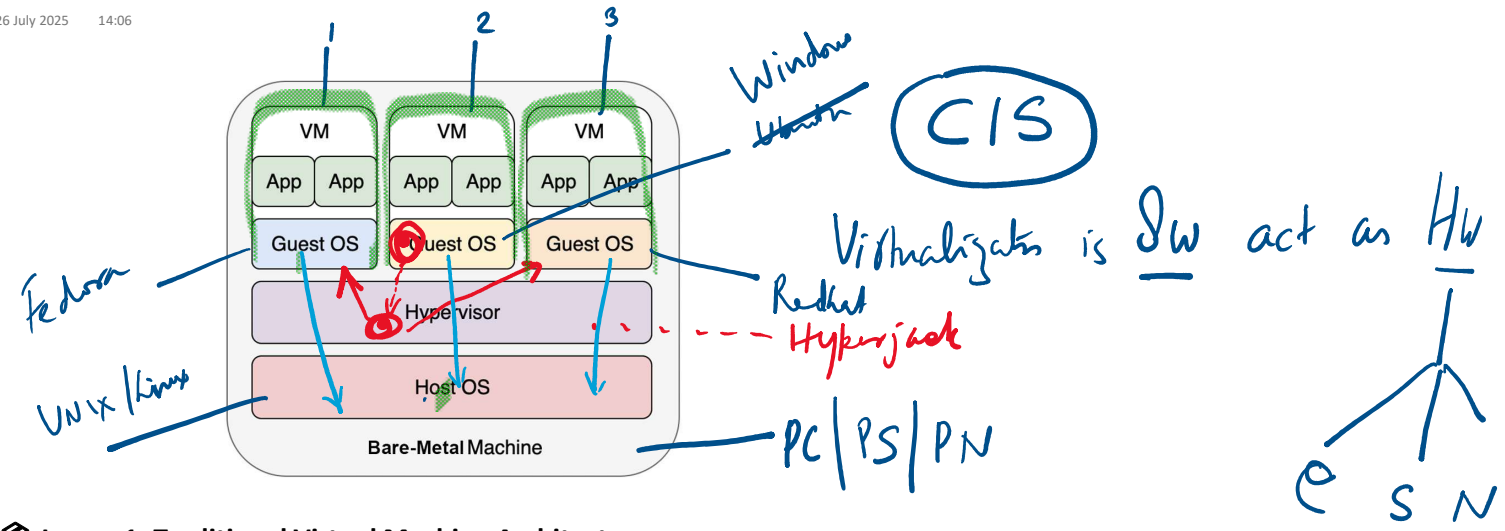


Image 1: Traditional Virtual Machine Architecture

Description:

This architecture represents **traditional virtualization** using a **hypervisor** to run multiple VMs on a **bare-metal machine**.

Key Components:

- **Host OS**: Base operating system on the physical machine (e.g., Linux, Windows).
- **Hypervisor**: Virtualization layer (e.g., VMware ESXi, Hyper-V, KVM) that allocates resources to each VM.
- **Guest OS**: Each VM runs its own OS instance, independent of the host or other VMs.
- **Apps**: Applications run within their respective VMs.

Pros:

- Strong isolation between environments.
- Mature ecosystem (e.g., vSphere, Hyper-V).

Cons:

- **Heavy resource usage** — every VM includes a full OS.
- Slower startup and scaling.
- Limited portability across environments.

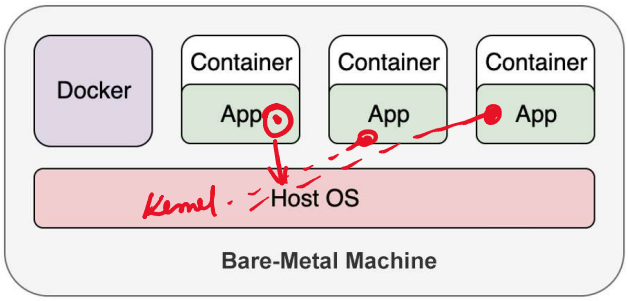


Image 2: Docker Container Architecture

Description:

This image shows a **containerized application stack** using Docker directly on a host OS.

🧠 **Key Components:**

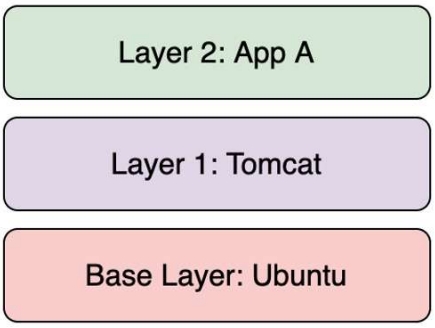
- **Host OS**: The underlying operating system (e.g., Linux).
- **Docker Engine**: Manages container lifecycle, networking, storage, and execution.
- **Containers**: Lightweight, isolated environments that share the host OS kernel but run apps independently.

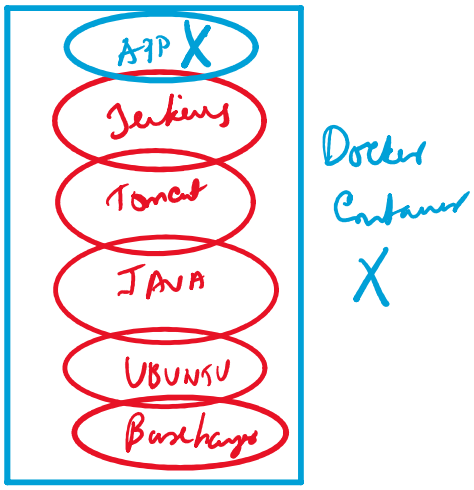
☑ **Pros:**

- **Lightweight and fast** — containers don't need full guest OS.
- Very fast startup time.
- Ideal for microservices, CI/CD, and dynamic scaling.
- Easy to version, package, and ship.

❌ **Cons:**

- Shared kernel → less isolation than VMs (though mitigated via cgroups, seccomp, AppArmor, etc.).
- Persistent storage and networking require deliberate handling.





An Image = R / O
A container = the top layer
R / W

🧱 **Image 3: Docker Image Layering**

🔍 **Description:**

This shows the **layered nature of Docker images**, critical for both build efficiency and reusability.

🧠 **Layer Explanation:**

- **Base Layer (Ubuntu)**: Starting point of the image — a minimal OS like Ubuntu, Alpine, etc.
- **Layer 1 (Tomcat)**: Application server added on top (e.g., installing Tomcat on Ubuntu).
- **Layer 2 (App A)**: Custom application layer — your deployed .war, .jsp, or any app code.

☑ **Pros of Layering:**

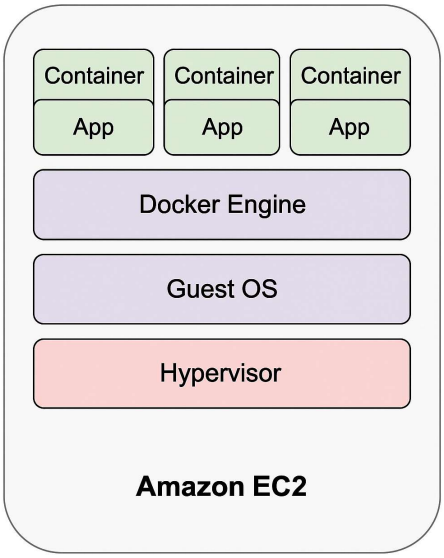
- Efficient caching — unchanged layers don't rebuild.
- Multiple images can share the same base layers → storage savings.
- Promotes image composition and reuse.

❌ **Pitfall to Avoid:**

- Changes high in the layer stack (e.g., app code) should be kept separate to avoid invalidating cache for lower layers.

🧩 **Expert Summary**

Topic	VM Architecture	Docker Containers	Image Layers
Isolation	Strong (hypervisor-level)	Process-level (namespaces)	Not applicable
Performance	Higher overhead	Lightweight	Very efficient reuse
Portability	OS-dependent VMs	Highly portable	Version-controlled
Use Case	Legacy, monoliths, full OS	Microservices, CI/CD, cloud	Docker build optimization



🖼️ **Image 04 – Docker Engine Running on Amazon EC2 (with Containers)**

📊 **Visual Overview:**

This diagram represents how Docker-based containerization operates within an **Amazon EC2 instance**—specifically illustrating the **virtualization and container layers** in a cloud-hosted environment.

🧠 **Layer-by-Layer Breakdown:**

Layer Description

◇ **Bare-Metal Machine**

- Represents the **physical AWS server** inside an AWS data center.
- AWS users don't interact directly with this layer, but it's where virtualization begins.

◇ **Hypervisor**

- AWS runs the **Nitro Hypervisor**, a lightweight and secure

hypervisor.

- Its job is to create and manage isolated virtual machines (EC2 instances) on top of physical infrastructure.

◇ **Guest OS**

- The **operating system running on your EC2 instance** (e.g., Amazon Linux, Ubuntu, RHEL).
- It is a fully functional OS and serves as the base for Docker installation.
- From this point onward, the environment is entirely under your control.

◇ **Docker Engine**

- Installed on the EC2 instance.
- Acts as the container runtime: responsible for **building, managing, networking, and running containers**.
- Directly interfaces with the Linux kernel to isolate process namespaces and file systems.

◇ **Containers + Applications**

- Each container runs a single application (e.g., web server, microservice, database).
- These containers share the same kernel (from the Guest OS) but run isolated environments.
- They are launched via the Docker Engine and may communicate via Docker networks.

🚀 **Execution Flow Summary**

1. You launch an EC2 instance (e.g., t3.medium) on AWS.
2. That EC2 runs on a virtual machine managed by the **Nitro Hypervisor**.
3. On the Guest OS (e.g., Amazon Linux 2023), you install Docker.
4. Docker Engine manages your application containers.
5. Each container runs an app, isolated, but lightweight and fast compared to VMs.

📌 **Real-World Use Case**

This setup is typical for:

- Hosting multiple services (e.g., frontend, backend, database) on a single EC2 during prototyping or testing
- Hosting self-managed Docker registries, CI runners, or utility containers
- Teaching environments and workshops (like yours)

  ⚠ **Important:** This is *not* Kubernetes — no orchestration or self-healing, and you're responsible for managing scaling, updates, and networking.

☑ **Benefits**

- **Lightweight** compared to running multiple EC2s or VMs
- Full control over Docker, networking, and volumes
- Ideal for **small-scale** deployments and prototyping

⚠ **Limitations**

- **Single-node** architecture — no orchestration, autoscaling, or HA

- Containers depend on the health of the EC2 instance
- Lacks dynamic service discovery, secrets management, and policy enforcement compared to Kubernetes