Understanding Virtualization

Containers vs Virtual Machines

Lisa Zangrando INFN-PD

Introduzione ai container e all'ecosistema Docker

What is Virtualization?

- **Virtualization** is the process of creating a **software-based representation** of a computing resource.
- The virtual entity behaves like the original
- Many forms exist
- Common categories: Servers, Storage, Networking

Focus today: Server Virtualization → Containers

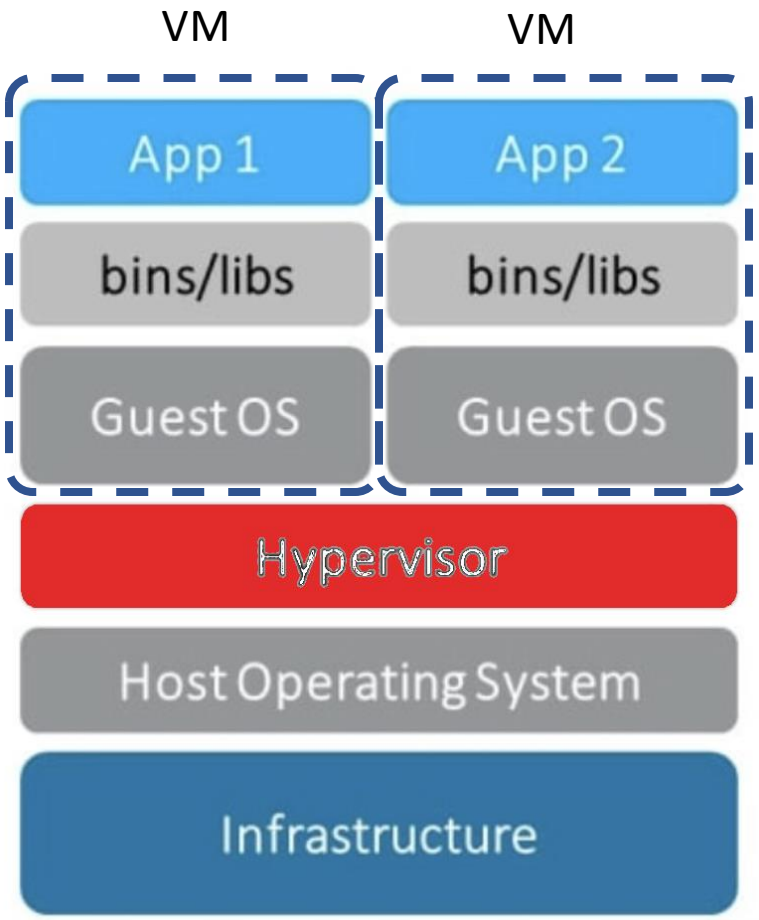
Server Virtualization

- Creates multiple isolated environments (virtual servers) to run different applications and operating systems on the same physical host
- To the user, each virtual server appears as a real, dedicated machine.
- Introduced to solve challenges of scalability, isolation, and hardware efficiency
- Two main approaches:
  - **Traditional virtualization (hardware-level)** → uses hypervisors and Virtual Machines (VMs)
  - **OS-level virtualization** → more recent, leads to containers

- Both aim to create isolated execution environments, but use different techniques.

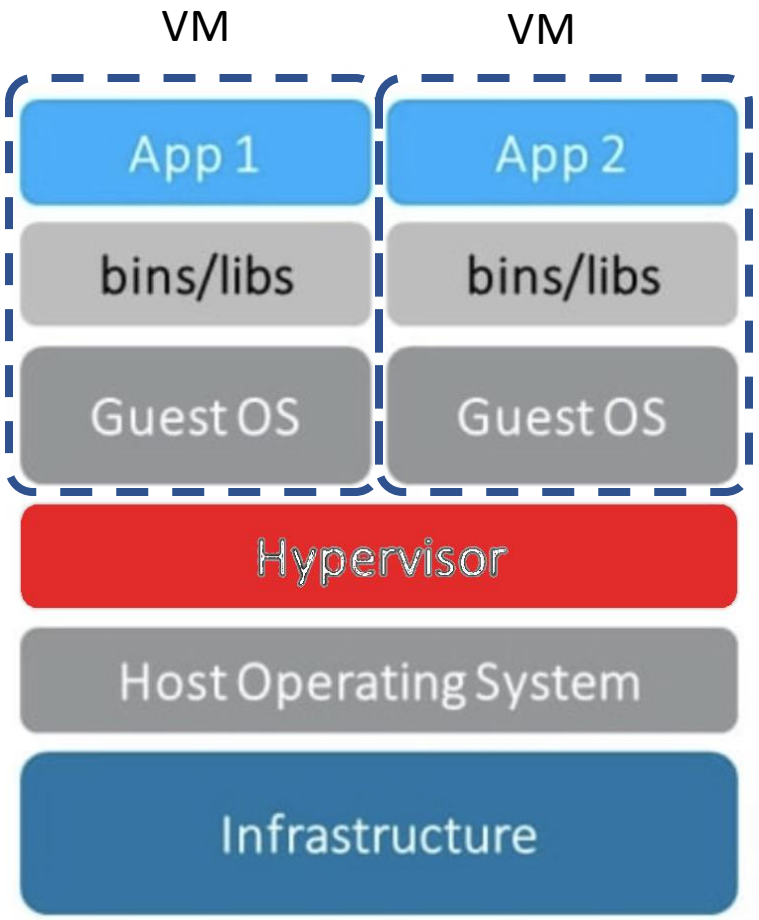
Hardware Virtualization

- Also known as traditional virtualization, introduced in the 1960s during the IBM mainframe era
- Still widely used today in modern cloud infrastructures
- Concept:
  - One physical server (**host**) runs multiple operating systems (**guests**)
  - Each guest runs inside an isolated environment called a Virtual Machine (**VM**)



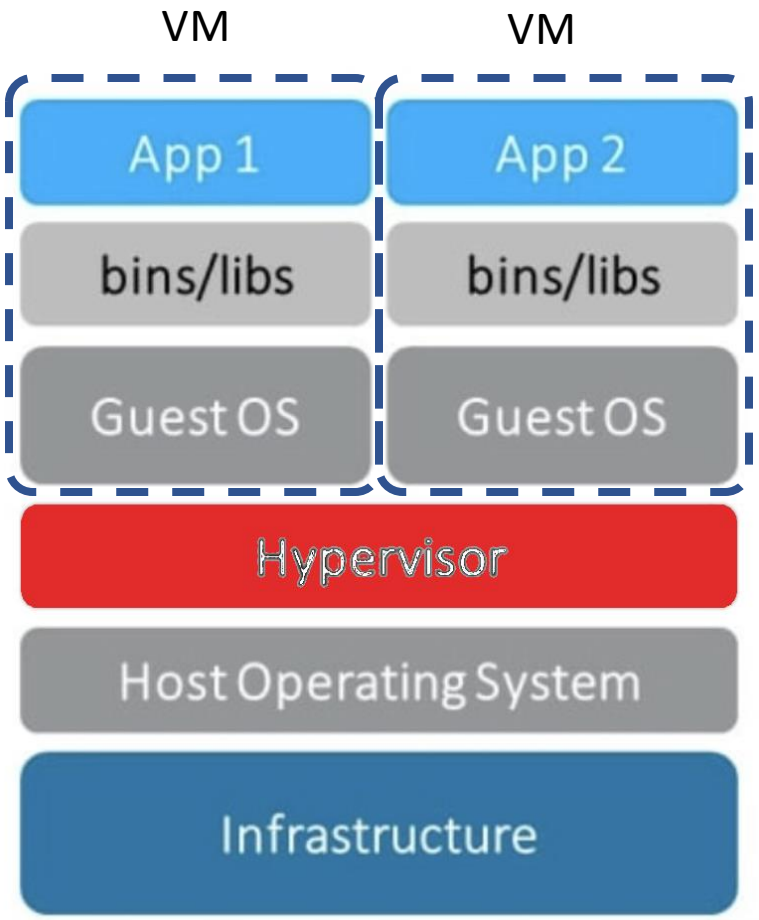
Hardware Virtualization architecture

- Hypervisor is a thin software layer (sometimes hardware-assisted)
- Sits between the host hardware and guest systems
- Abstracts physical resources (CPU, memory, storage, network) and presents them as dedicated virtual resources
- A VM behaves like a real computer:
  - Own virtual CPU, RAM, disks, and network interfaces
  - Can install an OS and applications just like on a physical server



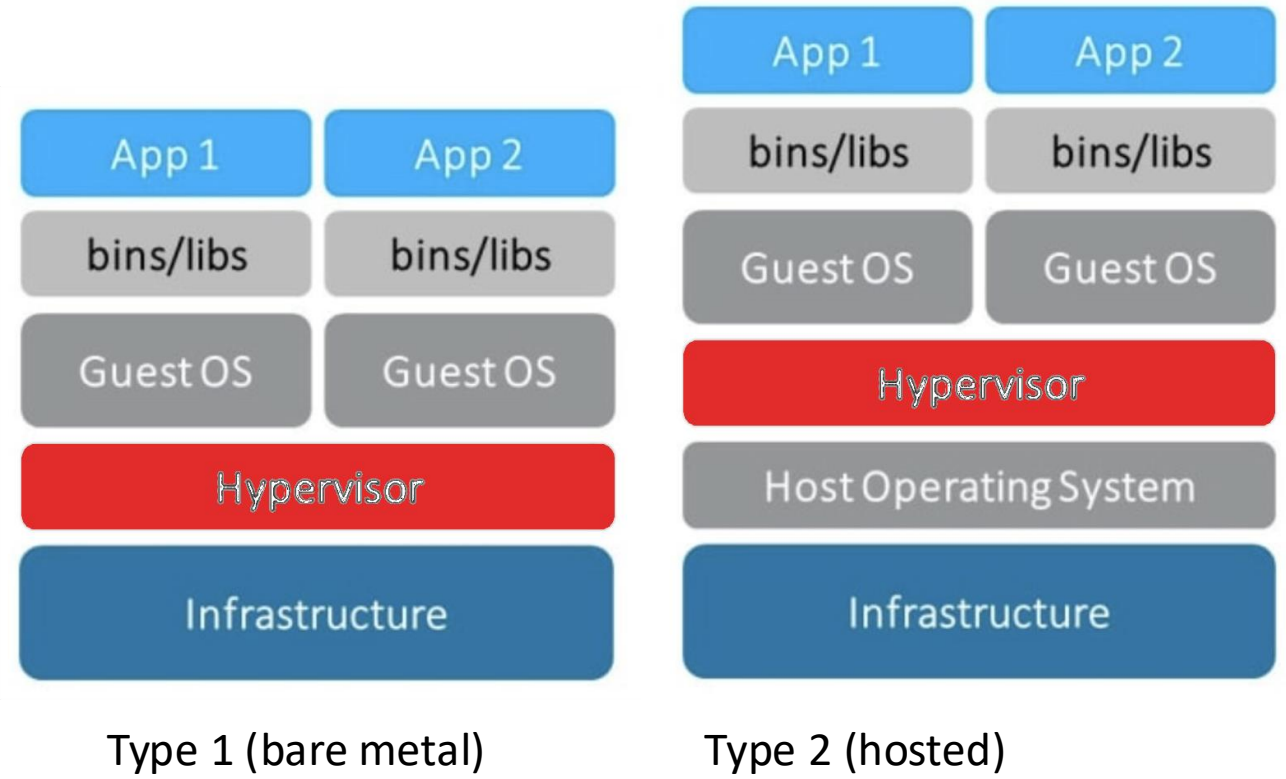
Capabilities of the Hypervisor

- The hypervisor is the core component enabling hardware virtualization. Its main roles include:
  - Hardware Abstraction
  - Resource Management
  - Isolation
  - Security Enforcement
  - VM Life-Cycle Management
  - Performance Monitoring
  - Device and I/O Virtualization
  - Live Migration Support
  - Fault Tolerance & High Availability (optional)



Types of Hypervisors

- Type 1 (Bare-metal):
  - Runs directly on physical hardware
  - High performance, strong isolation, minimal overhead
  - Examples: VMware ESXi, Microsoft Hyper-V, KVM
- Type 2 (Hosted):
  - Runs on top of an existing OS
  - Easy to install, ideal for desktops and testing
  - Lower performance due to extra layer
  - Examples: VirtualBox, VMware Workstation, Parallels Desktop



Type 1 (bare metal) Type 2 (hosted)

Approaches to HW Virtualization

- **Emulation**
  - Runs guest OS not compatible with host hardware
  - High flexibility (e.g., ARM on x86)
  - Performance penalty: hypervisor translates instructions
  - Examples: Bochs, VirtualPC, QEMU (without acceleration)
- **Full Virtualization**
  - Guest OS compatible with host architecture
  - Non-critical instructions run directly on hardware
  - Critical instructions handled by hypervisor:
    - Software traps
    - Hardware-assisted (Intel VT, AMD-V)
  - Examples: VMware, VirtualBox, Parallels, Xen, KVM/QEMU
- **Para-Virtualization**
  - Guest OS aware of virtualization
  - Communicates with hypervisor via APIs
  - Lower overhead, better performance
  - Requires OS guest modifications
  - Examples: Xen, KVM/QEMU, User-mode Linux

Advantages of HW Virtualization

- **Complete Isolation:** each VM operates independently
- **Flexibility:** ability to run different OS, even those not compatible with the host hardware.
- **Resource Optimization:** multiple VMs share server resources, increasing efficiency.
- **Advanced Features:** support for snapshots, live migration, backup, and high availability...
- **Full Compatibility:** run unmodified operating systems with full virtualization.
- **Cost Reduction:** lower hardware and operational costs through server consolidation.
- **Scalability:** easily add new VMs to handle variable workloads.
- **Rapid Provisioning:** fast creation and deployment of VMs and applications.
- **Business Continuity & Disaster Recovery:** supports continuity strategies and rapid recovery.
- **Simplified Management:** centralized administration and automation of data center resources.

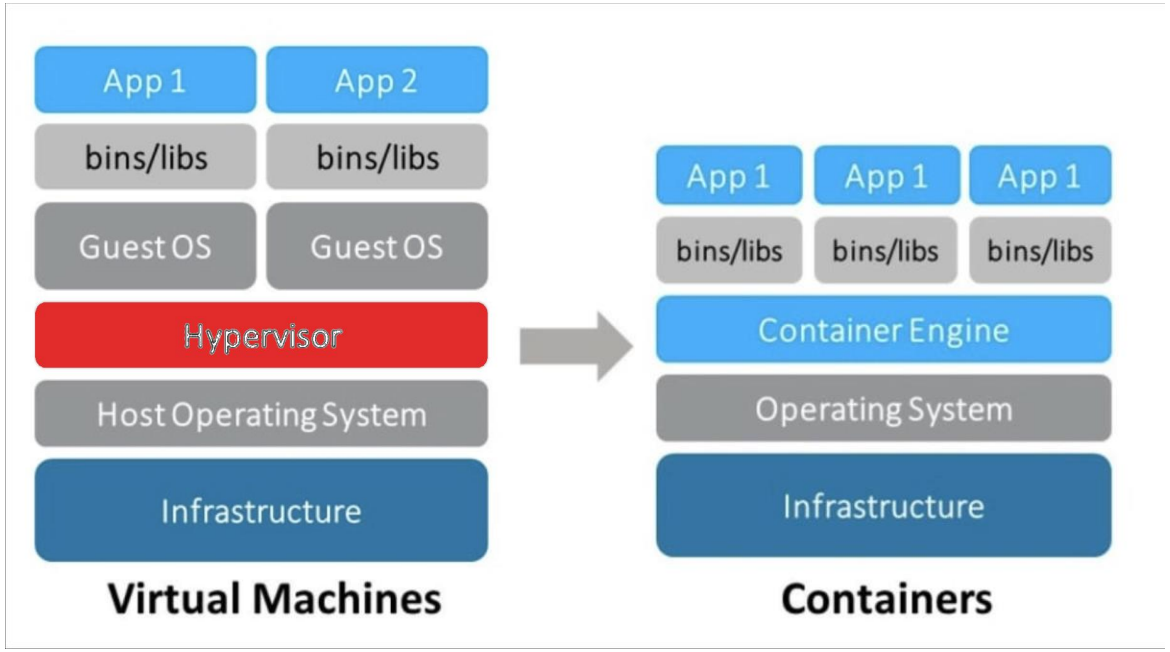
Disadvantages of HW Virtualization

- **High overhead:** each VM requires a full operating system, consuming CPU, memory, and storage.
- **Slow startup:** booting a VM is comparable to a physical computer (several seconds or minutes)
- **Complex management:** multiple VMs mean more OS images to maintain, update, and monitor.
- **Large size:** virtual disks and full OS occupy significant storage, making VMs heavier than containers.
- **Low density:** a physical host can run only a limited number of VMs before resources are saturated.
- **Reduced flexibility:** reducing overhead or scaling quickly is more difficult compared to lighter solutions like containers.
- **Higher hardware and operational costs:** greater resource consumption requires more investment in servers and infrastructure.

OS-Level Virtualization

- Container: lightweight package with app + dependencies (no full OS).
- Idea: run multiple isolated environments (containers) on a single kernel.
- No hypervisor: isolation provided directly by the Linux OS.
- Linux mechanisms (since 2008):
  - **Namespaces**: isolate processes, network, mounts, IPC, users.
  - **cgroups**: limit and control CPU, memory, I/O.

VMs vs containers



- Since the kernel provides process isolation and resource control, a hypervisor is no longer needed. However, a management layer is still required — the **container engine**.

- VMs virtualize hardware and run a full OS per instance → require a hypervisor.
- Containers share the host kernel → faster, lighter, more efficient.

Why we need a Container Engine?

- Linux kernel: provides containerization primitives, not a full runtime
- Container engine: makes them practical, reproducible, scalable
- Provides:
  - **Image management:** build, pull, layers
  - **Filesystem support:** OverlayFS
  - **Networking:** virtual networks, port mapping, CNM/CNI integration
  - **Lifecycle:** start, stop, restart containers
  - **Isolation enforcement:** applies namespaces, cgroups, seccomp, AppArmor/SELinux profiles

Container Engines overview

- Common engines: **Docker**, containerd, CRI-O, Podman, Apptainer (formerly Singularity).
- **Docker**: first (2013) and most popular engine, simplifies image build, networking, and container lifecycle
- **containerd / CRI-O**: lightweight runtimes, often used with Kubernetes
- **Podman**: daemonless, rootless, secure execution
- **Apptainer**: designed for HPC and scientific workflows

- **Course focus**: we will use Docker to explore OS-level virtualization and containers

Advantages of OS-level Virtualization

- **Lightweight & fast**
  - Containers share the host kernel → start almost instantly and use fewer resources.
  - Higher workload density on the same server compared to VMs.
- **Portability — build it once, run it anywhere**
  - Container images bundle the app and all dependencies.
  - Applications can run consistently across local desktops, physical servers, virtual servers, production environments, and public or private clouds.
- **Improved developer productivity**
  - Containers create predictable runtime environments, minimizing debugging due to environmental differences.
  - Developers and operations teams can focus on building and delivering new features rather than solving environment issues.

Advantages of OS-level Virtualization

- **DevOps & Orchestration friendly**
  - Integrates naturally with CI/CD pipelines and orchestrators like Kubernetes.
  - Simplifies deployment, scaling, and automation.
  - Supports microservices and cloud-native architectures.
- **Efficient Isolation & Resource Control**
  - namespaces provide process, network, and filesystem isolation.
  - cgroups allow fine-grained control of CPU, memory, and I/O.
  - All this without the overhead of a full OS per instance.

Disadvantages of OS-level Virtualization

- **Isolation & security**
  - Containers share the host kernel → a kernel bug or vulnerability can affect multiple containers.
  - Requires careful security: policies, kernel updates, and isolation tools like SELinux or AppArmor.
- **OS dependency**
  - Containers rely on the host OS kernel → Linux containers run natively only on Linux hosts.
  - Cannot run different OS containers without extra virtualization.
- **Application compatibility**
  - Some legacy or monolithic applications may not containerize easily.
  - Dependencies on specific OS versions, libraries, or kernel configurations can cause issues.

Disadvantages of OS-level Virtualization

- **Complexity at scale**
  - Orchestrating thousands of containers requires advanced tools (Kubernetes) and skills.
  - Monitoring, logging, and troubleshooting can be challenging
- **Resource sharing & interference**
  - Shared kernel can lead to performance interference if resource limits (cgroups) are not properly configured.

Extended comparison: Virtual Machines vs Containers

Aspect	Virtual Machines	Containers
Architecture	Virtualize hardware and include a full OS for each instance.	Share the host kernel; include only the application and its dependencies.
Weight & size	Heavy: tens of GB; startup in minutes.	Lightweight: a few MB; startup in milliseconds.
Security	Very strong isolation; VM escapes are rare.	Require extra attention: kernel vulnerabilities affect all containers.
Portability	Limited to the same hypervisor or image format.	Very high: “build once, run anywhere” with a compatible kernel.
Performance	Higher overhead due to hypervisor and dedicated OS.	Near-native performance thanks to kernel sharing.
Resource consumption	Each VM needs memory and CPU for its OS.	Very efficient: no OS per instance.
Scalability	Slower and more expensive to scale.	Fast, automated, horizontal scaling.
Application model	Less suitable for microservices or cloud-native architectures.	Ideal for microservices and cloud-native apps.

Extended comparison: Virtual Machines vs Containers

Aspect	Virtual Machines	Containers
Application compatibility	Runs any OS and legacy app.	Limited to the host kernel; some legacy apps may not work.
Updates & patching	Heavier: entire OS must be maintained per VM.	Immutable images: replace containers rather than updating in place.
Time to recovery	Slower: rebooting a VM takes seconds/minutes.	Fast: restarting a container takes milliseconds.
Orchestration	Rarely orchestrated; rely on hypervisors and traditional tools.	Highly orchestratable (Kubernetes).
Large-scale management	Easier to monitor due to fewer instances.	Complex without orchestration: requires logging, monitoring, and distributed tracing.
Typical use cases	Legacy apps, multi-OS environments, strong isolation needs, heavy databases, traditional middleware.	Microservices, CI/CD, cloud-native apps, elastic scaling, stateless workloads.
Operational cost	Higher resource usage → higher infrastructure costs.	Higher density → lower operational costs.
Learning curve	Moderate	Can be steep, especially with Kubernetes.

Conclusions

- VMs remain essential when you need full isolation, support for different operating systems, or compatibility with legacy applications.
- Containers excel in lightness, speed, portability, and scalability, making them ideal for modern cloud-native and microservices architectures.
- The best approach is often a combination: use VMs to provide strong isolation and security, and run containers on top for agility, efficiency, and rapid deployment.

Thanks !!!