Virtualization Overview

1 Introduction

Virtualization is the enabling technology that underpins today's cloud computing platforms and the elastic, service-oriented approach that dominates modern datacenter design. By inserting a software layer between the workload and the hardware, virtualization enables many independent execution environments to coexist on the same physical infrastructure. This improves the utilization of expensive resources and provides the operational flexibility on which large-scale services rely.

2 Architectural Foundations

The instruction-set architecture (ISA) lies at the border between hardware and software. It is the contract that specifies which binary instructions a processor can execute and the side effects that these instructions cause. Although every program ultimately resolves into ISA instructions, developers typically write in high-level languages and rely on compilers, runtime libraries, and operating system services to emit the correct machine instructions on their behalf. From the bottom up, the full stack includes digital logic, micro-architecture, the ISA, the operating-system (OS) interface, compilers and high-level languages.
Two interfaces play a central role in virtualization:

- The System ISA, which is the privileged portion used by an OS to configure and control the platform,
- The Application Binary Interface (ABI), which is the unprivileged interface that user processes rely on and is mediated by the OS through system calls and libraries.

Virtualization works by reimplementing one of these interfaces in software, making a guest believe that it owns a real machine even though its instructions are caught, redirected, or transformed before reaching the underlying hardware.

3 Virtual Machines and the Hypervisor

Virtualization succeeds by presenting the illusion of one or more independent machines, each of which behaves as though it has sole ownership of the physical platform. This illusion is achieved through two complementary concepts: the virtual machine, which is the abstraction exported to the guest, and the hypervisor, which is the privileged control program that implements and enforces this abstraction. It is essential to understand their taxonomy, as they must consider performance, isolation, portability and operational cost.

3.1 Process-Level versus System-Level Virtual Machines

A process virtual machine operates at the application binary interface (ABI), translating or mediating instructions from user space programs so that they can run unmodified on any underlying hardware or operating system. Examples include the Java Virtual Machine (JVM), Microsoft's Common Language Runtime (CLR) and WebAssembly engines. Their purpose is to enable portability and safety rather than the strong workload isolation required in data centres;

indeed, many JVMs can coexist in the same Linux process table. Nevertheless, some of the same principles anticipate later techniques in system virtualization, e.g. byte code interpretation, ahead-of-time (AOT) or just-in-time (JIT) compilation, managed memory, and sandboxed system call emulation.
By contrast, a system virtual machine recreates an entire computer at the hardware interface, i.e. the instruction set architecture (ISA). An unmodified operating system boots into that virtualized environment exactly as it would on bare metal, loading its own kernel modules, network stack and userspace utilities. System VMs therefore offer two features that process VMs cannot: (1) strong security isolation based on separate kernel address spaces and (2) the ability to combine different guest kernel types (Linux, Windows, BSD and proprietary RTOSes) on the same server. These qualities explain why public clouds primarily offer system virtual machine (VM) abstraction as a building block for infrastructure as a service (IaaS).

3.2 Hypervisor Deployment Models

A hypervisor, also known as a virtual machine monitor (VMM), is a software layer that manages resources, enforces isolation, and handles input/output (I/O) on behalf of each guest. It is the actual software that manages virtual machines (VMs) and enables virtualisation. Hypervisors are traditionally categorized according to their position in the system stack; however, practical deployments reveal additional architectural distinctions.

Type 1 (bare-metal) hypervisors such as VMware ESXi, Microsoft Hyper-V or KVM run directly on the host processor, and claims the highest CPU privilege level. All guest requests for CPU cycles, memory or I/O cross the hypervisor boundary first. Their common virtues are:

- **Minimal attack surface** – there is no large, general-purpose OS beneath the hypervisor that might leak guest data.
- **Predictable latency** – direct control of interrupt routing and CPU scheduling avoids the double indirection present in Type 2 designs.
- **Hardware homogeneity** – data center operators can qualify a single hypervisor kernel and driver set for thousands of nodes.

Within Type 1, two internal architectures exist:

- **Monolithic kernels** bundle device drivers and control code into a single privileged image, giving tight, low-latency/low-overhead access at the cost of lower flexibility if a new device appears. Device drivers should be within the original kernel.
- **Micro-kernel or service-VM approaches** (e.g. Xen) outsource complex drivers to a specialised, privileged guest (Service-VM) while keeping the core hypervisor minimal. This extra indirection adds latency but greatly eases portability/flexibility.

Type 2 (hosted) hypervisors are ordinary processes inside a conventional OS; they delegate many hardware duties to that host OS. Hosted designs are easier to install but suffer extra overhead and are uncommon in production clouds. In detail, they inherit device drivers, memory management and scheduling from the host OS, which simplifies portability (no private driver stack is needed) but introduces three unavoidable penalties:

- **Performance overhead** due to nested scheduler decisions and extra context switches.
- **Security coupling**, a guest escape vulnerability can compromise the host user account, which in turn may access sensitive data or networks.
- **Inconsistent latency** under high host load, limiting suitability for real-time or low-jitter applications.

Consequently, Type 2 hypervisors are widely used on laptops and developer workstations, and not (or rarely) in production clouds. VMware Workstation, Oracle VirtualBox, and Parallels Desktop fall into this category.

Interestingly, some modern systems blur the line between Type 1 and Type 2 by adopting a **hybrid approach**. For example, Apple's *Hypervisor.framework* exposes hardware virtualization to the user space, while macOS retains full device control, effectively implementing a minimalist hosted model. Similarly, Linux's KVM module runs in kernel space (showing Type 1 characteristics), but relies on the rest of the Linux kernel for scheduling and memory management (showing Type 2 characteristics). In practice, the landscape of hypervisors is more complex than a simple Type1-versus-Type2 table that appears in many textbooks. Each approach occupies a point in the multidimensional space of performance, isolation strength, device compatibility, management complexity, and boot latency. Understanding these trade-offs is important to make the right choice considering the workload demands and needs.

4 Virtualization Techniques

Full virtualization and paravirtualization are two approaches to system-level virtualization, each with distinct characteristics.

- **Full virtualization** uses *trap-and-emulate* or *binary translation* so that every sensitive instruction executed by the guest is intercepted and either emulated or rewritten on the fly. Early x86 processors lacked a clean privilege model for virtualization. However, today's hardware extensions, such as Intel VT-x/VT-d and AMD-V, provide additional modes in which sensitive guest instructions (e.g., those that access control registers or I/O ports) automatically trigger a trap into the VMM. Memory management units expose second-level address translation (Intel EPT and AMD RVI), enabling guest page tables to be used with minimal shadowing overhead.
- **Paravirtualization** can be used where hardware assistance is absent or where maximum performance is required. In this case, the guest OS can be modified so that privileged operations are made explicit via *hypercalls* to the hypervisor. This paravirtualization removes many expensive traps, but it ties the guest to a particular VMM interface. Xen popularized this model, and modern Linux distributions still ship paravirtual drivers (VirtIO) for storage and networking, even when the rest of the kernel remains unmodified.

To provide further insight into virtualization techniques, it is worth mentioning something more on I/O virtualization. While CPU and memory virtualization are well established, I/O remains challenging because high-performance devices require direct, DMA-based access to physical memory. Technologies such as SR-IOV enable a single NIC or GPU to expose numerous lightweight *virtual functions* that the hypervisor can allocate directly to guests, thereby bypassing much of the emulation overhead while retaining the isolation enforced by an IOMMU.

5 Key Properties Desired by Datacenter Operators

In the context of datacenters, virtualization platforms are evaluated less on raw speed than on how reliably they deliver four core properties:

- **Partitioning** – subdivision of a server's CPU time, memory space and devices into self-contained slices;
- **Isolation** – faults or attacks in one VM must not propagate; performance interference must be predictable and, where possible, bounded;
- **Encapsulation** – a VM's entire state can be packaged, paused, snapshot or migrated ("live migration") between hosts without service interruption;
- **Hardware independence** – the same VM image can boot on any compatible hypervisor regardless of server vendor, enabling cloud elasticity and disaster recovery.

These characteristics are useful in the context of a datacenter to permit the following use:

- **Server consolidation**: workloads that once required separate physical machines can cohabitate, shrinking resource footprints and power consumption;
- **Elastic provisioning**: cloud providers spin VMs up or down in seconds to track customer demand, paying only for resources that are actually consumed;
- **High availability**: encapsulation lets orchestration software restart or migrate VMs away from failing hardware with little or no downtime;
- **Security and multitenancy**: isolation mechanisms, together with attestation and encrypted memory (e.g., AMD SEV, Secure Encrypted Virtualization, and Intel TDX, Trust Domain Extensions), protect customers who share the same rack;
- **Test and development**: snapshots create repeatable, sandboxed environments; dev-ops workflows treat VM images as immutable artefacts that flow through CI/CD pipelines.

6 Containers: Lightweight Virtualization at the OS Level

Virtual machines virtualize hardware; containers virtualize the operating-system. Running on Linux, container engines such as Docker or Podman combine namespaces (to give each container its own view of processes, filesystems and networks) with cgroups (to police resource usage). Because every container shares the host kernel, there is no need to bundle guest OS images, so start-up is near-instantaneous and density is extremely high.

More in detail, containers are lightweight, portable, and self-sufficient software packages that contain everything needed to run an application, including code, runtime, system tools, libraries, and dependencies. They encapsulate applications in a way that ensures consistency and reproducibility across different computing environments. Docker is one of the most popular containerization platforms, but there are others like Podman, LXC (Linux Containers), and container orchestration tools such as Kubernetes.

When we talk about containers, we have to refer to the following main components: (i) the container Image, which is a read-only template containing the application and its dependencies; (ii) the Container Runtime and the Container Engine, respectively, the software that manages containers - including starting, stopping, and interacting with them, and the software responsible for building, running, and distributing containers, such as Docker Engine; (iii) Container Instances, the running instances of a container image. Each instance is isolated from others, having its own filesystem, CPU, memory, and network resources. They can be managed by container orchestration tools like Kubernetes, which handle scaling, failover, and deployment patterns. (iv) Containers are described by a text file that contains all the commands needed to build a container image. It's like a recipe for your application, specifying

the base image to start from, the application code to include, and the dependencies to install. The Container Engine reads this file and builds an image accordingly.

The main benefits of the containerization are the following:

- Isolation: Containers use process isolation at the OS level, where each container shares the host operating system's kernel but runs in an isolated user space.
- Flexibility: even the most complex applications can be containerized;
- Lightweight: Containers consume fewer resources compared to virtual machines. Containers exploit and share the host kernel.
- Fast Startup: Containers start almost instantly since they do not require booting an entire operating system.
- Portability: Containers can run consistently across different environments, from development to production.
- Scalability: Containers are easily scalable, allowing for efficient resource utilization.

Containers and virtual machines serve different purposes and have distinct architectural approaches. Containers are lightweight, portable, and efficient for running applications with minimal overhead, while virtual machines provide stronger isolation and encapsulation but with higher resource consumption and longer startup times. In particular,

- Architecture: VMs virtualize hardware resources and run a complete guest operating system on top of a hypervisor, while Containers virtualize at the operating system level, sharing the host OS kernel and libraries while isolating applications and their dependencies.
- Resource Consumption: VMs are heavier as they include an entire guest OS, consuming more resources (CPU, memory, storage). Containers are lightweight since they share the host OS kernel and resources, resulting in lower resource consumption.
- Isolation: VMs offer strong isolation as each VM has its own kernel, complete OS, and resources. Containers provide process-level isolation, sharing the kernel but isolating file systems, networking, and process space.
- Boot Time: VMs take longer to start as they need to boot a complete operating system. Containers start quickly as they don't require OS booting, making them ideal for microservices and rapid application deployment.
- Deployment: VMs are deployed as separate instances with their own OS, requiring more management overhead. Containers are deployed as instances of images, making deployment and scaling more streamlined and manageable. On the other hand, virtualization systems allow heterogeneous operating systems, whereas containers share the same kernel.

7 Emerging Horizons on Virtualization

Virtualization is a moving target rather than a static achievement. Chipmakers, cloud operators and open-source communities continue to extend the abstraction so that new classes of workload can be hosted efficiently and securely. The trends below capture some of the most influential directions in which the field is evolving.

- **GPU and accelerator virtualization** – NVIDIA’s vGPU, AMD’s MxGPU and Intel’s GVT-g lend virtual functions of a single accelerator to multiple tenants; workloads such as AI inference or VDI can now be sold “as a service”;

- **Edge and IoT** – micro-VM frameworks (Firecracker, Cloud Hypervisor) boot in milliseconds with MB-scale footprints, suiting latency-sensitive edge nodes where a full hypervisor would be too heavy;
- **Confidential computing** – hardware-encrypted VM memory regions prevent even the cloud operator's privileged code from snooping on guest data, an important step for regulated industries;
- **Serverless runtimes** – in the smallest unit of virtualisation yet, functions may execute in micro-VMs that live for only a few seconds, blurring the line between IaaS VMs and PaaS abstractions.

8 Conclusion

Virtualization has evolved from its origin back on IBM mainframes in the 60s into a pervasive foundation for cloud computing, with sophisticated hardware support and a rich software ecosystem. Whether delivered through classic hypervisors, paravirtual interfaces, containers, or emerging micro-VMs, the central promise remains unchanged: to decouple the logic of software from the constraints of hardware. These concepts allow engineers to design resilient, efficient, and secure data centers and to understand the complex trade-offs among performance, flexibility, and isolation that shape every modern computing platform.