

Computing & Virtualization

Reading time: ~10 min · Prereqs: Lessons 01–08 · *Snapshot 2026 — hypervisor technology is stable; orchestration tooling, especially Kubernetes, evolves quickly.*

Why this matters to you

Every conversation about “the cloud,” every Jira ticket that says “we need more compute,” every blog post about Kubernetes eventually arrives at the same question: where, physically and logically, does your software actually run? The answer is almost never “on one computer.” It is some combination of rented servers, virtual machines, and lightweight containers managed by software that decides, moment to moment, which hardware to use. If you work anywhere near a product that has users, understanding this layer tells you why deployments take the time they do, why the monthly cloud bill looks the way it does, what a “container” is when a developer mentions it, and what an SRE actually does when they say they’re “on call.” This lesson is the mental model: no configuration, no commands, just the map.

1. The spectrum from cupboard to hyperscaler

Where a company runs its software is a spectrum, not a binary choice. At one extreme: the dusty desktop-class server humming in a locked cupboard, managed by whoever still knows the root password. At the other: a data center the size of several city blocks, owned by Amazon or Google, with its own power substation, fiber entry points, and 24-hour physical security. Most real organizations sit somewhere in between.

On-premises (on-prem) means the hardware is in a building you control — your office, your data-center room, your leased cage in a co-location facility. You bought the servers (capital expenditure, **CapEx**), you rack them, you patch them, and your team gets the pager at 3 AM when a disk fails. The upside is control: every setting, every cable, every security policy is yours. The downside is that the hardware sits idle at 5–15% average utilization on a busy day because you had to buy enough capacity for your peak load.

Public cloud (AWS, Azure, GCP) flips that equation. You pay by the second for exactly as much compute as you use (operating expenditure, **OpEx**). Spin up a thousand servers for a one-hour batch job, pay for one hour, shut them down. No hardware to buy, no data center to cool. The downside: you have no control over the physical layer, costs can surprise you spectacularly if an engineer forgets to turn something off, and at very large sustained loads the bill often exceeds what owned hardware would have cost. Many large companies discovered this in the early 2020s and “repatriated” significant workloads back on-prem after the initial cloud euphoria faded.

Neither answer is universally correct. The industry as of 2026 runs on a **hybrid** mix: regulated data (banking, healthcare) often stays on-prem or in a private cloud for compliance reasons; burst capacity and experimental workloads go to the public cloud; the teams managing both speak the same conceptual language.

2. What virtualization actually is

Until around 2000, the default model was one operating system per physical server. If you needed to run a web server and a database server, you bought two machines. Both sat mostly idle. Then **virtualization** arrived with a deceptively simple idea: what if the hardware could pretend to be multiple independent computers at once?

The analogy that works best is an apartment building. Imagine a huge house with one owner. Now imagine a developer buys it, installs internal walls, separate front doors, and separate meter boxes, and rents each flat independently. Each tenant has their own kitchen, their own bathroom, their own locks. They cannot walk into a neighbour’s flat. From a tenant’s point of view, they live in a complete, self-contained home. They have no idea how many other tenants are in the building, and they do not care. The building owner — the **hypervisor** — is the one who manages the shared structure underneath: the foundations, the electrical wiring, the plumbing.

In computing terms: the physical server is the building. The hypervisor is the landlord software that divides the hardware into isolated slices. Each slice is a **virtual machine** (VM): it has its own virtual CPU, virtual RAM, virtual disk, and its own complete operating system. The guest OS inside the VM has no idea it is sharing a physical host with ten other VMs. From inside, it looks and behaves like a bare-metal server.

Why did this change everything? Four reasons. First, **utilization**: one physical host doing the work of ten idle servers dramatically reduces hardware and electricity costs.

Second, **isolation**: a runaway process in VM 3 cannot eat the memory of VM 7; the hypervisor enforces hard limits. Third, **mobility**: a VM is ultimately just a large file on disk. You can pause a running VM, copy it to another physical host, and resume it — hardware maintenance stops being an emergency that requires a service window. Fourth, **snapshots**: before applying a risky patch, take a snapshot of the VM's current state. If the patch breaks things, revert to the snapshot in seconds. Sysadmins who had previously dreaded change windows started saying yes to upgrades much more readily.

3. Hypervisor types and where containers fit

Not all hypervisors work the same way. **Type 1 hypervisors** run directly on bare metal: the server boots into the hypervisor software, not into a general-purpose operating system. There is no host OS underneath — the hypervisor is the OS. This is the architecture of the big names used in production data centers: **VMware ESXi**, **Microsoft Hyper-V**, **KVM** (the Linux kernel's built-in hypervisor), and **Xen** (which Amazon Web Services used under the hood for a decade). Type 1 has the best performance because there is one fewer software layer between your VM and the hardware.

Type 2 hypervisors run inside a host operating system as a regular application. You install **VirtualBox** on your Windows laptop, click "New Virtual Machine," and a few minutes later you have a Linux VM running in a window. **VMware Workstation** and **Parallels** (popular on macOS for running Windows) work the same way. Type 2 is convenient and perfectly fine for development and testing; the added overhead is noticeable but rarely a problem when you just need to try something quickly.

Containers (popularised by **Docker** starting around 2013) solve an overlapping problem by different means. Instead of virtualizing the hardware and running a full guest OS, containers virtualize only the operating system layer — they share the host OS's kernel and isolate only the process, filesystem, and network. The result is dramatically smaller and faster: a container image is megabytes rather than gigabytes; it starts in milliseconds rather than seconds. The trade-off is weaker isolation — containers share a kernel, so a kernel vulnerability can affect all containers on the host, whereas VMs each have their own kernel and are harder to escape. In practice, containers and VMs coexist: Kubernetes, the dominant container orchestrator, most commonly runs its container workloads on top of virtual machines, not bare metal. The apartment-building analogy extended: VMs are the individually walled flats; containers

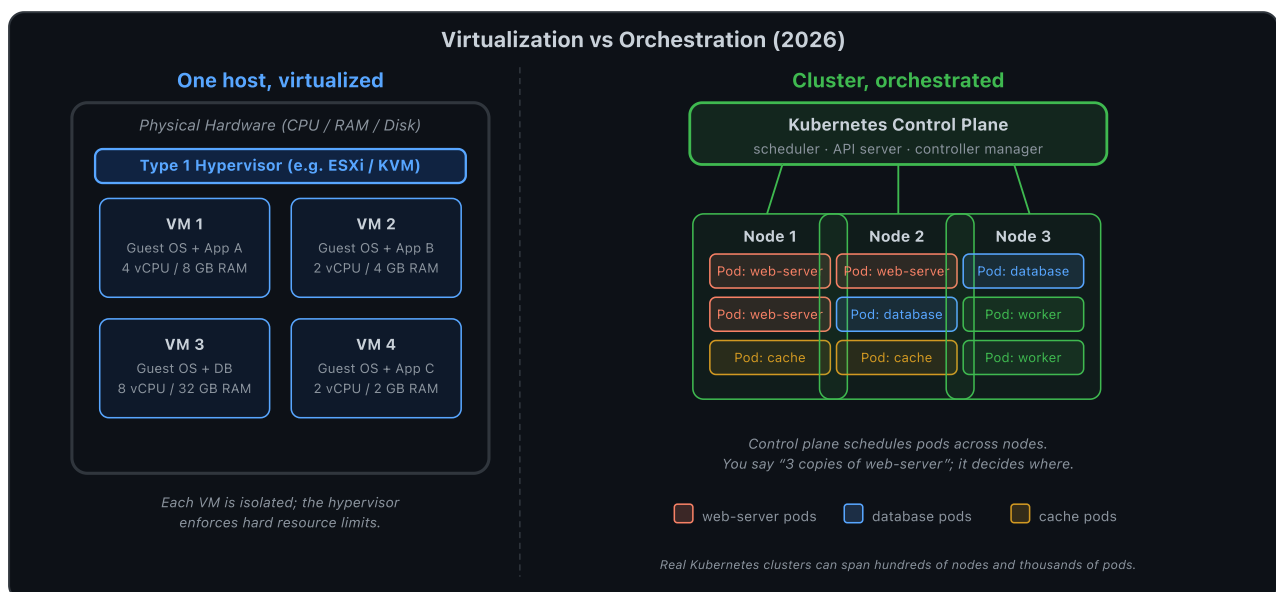
are more like office suites within one flat — shared hallway, but each suite has its own locked door.

4. Orchestration: the conductor for a compute orchestra

When you have one VM, you log into it. When you have ten, you probably have a spreadsheet. When you have a thousand, you need software to track where everything is, decide where to place new workloads, restart anything that dies, and distribute traffic evenly. That software is called an **orchestrator**, and the word is exactly right: an orchestrator is the conductor who tells every musician what to play and when, without touching an instrument itself.

5. The mental model in one diagram

Here is a side-by-side picture of the three layers: the physical host, the hypervisor that divides it, and the orchestrator that manages a cluster of hosts. The left side shows a single host sliced into VMs; the right side shows an orchestrator (Kubernetes) scheduling containers across a pool of nodes.



The left panel shows one physical host divided by a hypervisor into four independent virtual machines, each with its own guest OS. The right panel shows a Kubernetes cluster: a control plane telling three nodes where to run which containers (pods). In production, those nodes are often themselves VMs — so the two models stack, rather than replace each other.

The four major orchestrators each occupy a slightly different niche. **VMware vCenter** manages fleets of ESXi hypervisors and their VMs; it is the dashboard that a traditional enterprise VMware shop uses to see all its virtual infrastructure in one place.

OpenStack is the open-source alternative, popular in telecom operators and universities that want to build a private cloud without paying VMware licensing fees.

Kubernetes (originally from Google, now a CNCF project) became the de-facto standard for running containerized applications and commands a huge ecosystem of supporting tools. **Nomad** (from HashiCorp) is lighter-weight and notably handles both containers and plain executable jobs without the complexity of Kubernetes — often the right answer for organizations that find Kubernetes operationally overwhelming.

5. Watching the herd: monitoring and the people who do it

Running thousands of VMs and containers is only possible if you have real-time visibility into what they are doing. The industry has settled on three complementary signals, which together are called **observability**.

Metrics are numbers sampled at regular intervals: CPU utilization is at 72%, memory consumption is 4.2 GB, the web service is handling 340 requests per second.

Prometheus (open-source) and **Datadog** (commercial SaaS) are the dominant tools for collecting and storing metrics. Metrics are cheap to store and excellent for alerting: “if CPU stays above 90% for 5 minutes, page the on-call engineer.” They are less good for understanding *why* something is slow.

Logs are the text lines that running processes emit: timestamps, error messages, warnings, HTTP request records. They are verbose (a busy service might emit millions of lines per minute) but rich in context. Log aggregation tools like **Elasticsearch** (the “E” in the ELK stack) and **Grafana Loki** ingest these streams and make them searchable. When an alert fires, the engineer opens the logs for the offending service to find the specific error that triggered it.

Traces capture the path one request takes through a distributed system. When a user clicks “Buy” on an e-commerce site, that request might touch a frontend service, an inventory service, a payment service, and a database — all in under 200 milliseconds. A trace records each hop, its duration, and any errors. **Jaeger** and **Zipkin** are the common open-source options; Datadog and New Relic include tracing in their commercial platforms. Without traces, diagnosing a slow request in a microservice

architecture feels like trying to solve a murder mystery with only the last page of the novel.

Grafana deserves a special mention as the most popular open-source dashboard layer: it connects to Prometheus, Loki, and many other data sources, and lets teams build visual dashboards showing all three signal types on one screen. When something breaks at 3 AM, the on-call engineer opens Grafana before touching anything else.

The people who operate this stack carry several job titles. A **sysadmin** (systems administrator) is the original title: keeping servers alive, managing OS updates, user accounts. An **SRE** (Site Reliability Engineer, a title Google coined) treats reliability as a software engineering problem: they write code to automate toil, set **SLOs** (service-level objectives like “99.9% of requests complete in under 200 ms”), and use error budgets to decide when it is safe to ship new features versus when to focus on stability. A **platform engineer** builds the internal tools — the deployment pipelines, the Kubernetes clusters, the secrets management systems — that product developers use every day without thinking about them. All three roles exist along a continuum; at a small company, one person does all three.

What people argue about

- **Cloud repatriation: is the cloud overpriced at scale?** Since roughly 2022, a number of high-profile companies (Basecamp/HEY, Dropbox partially, several financial-services firms) published analyses showing that at consistent, predictable loads, owning hardware is substantially cheaper than renting it. The counter-argument from cloud vendors and their advocates is that the analysis ignores the engineering headcount required to manage on-prem infrastructure, the opportunity cost of capital tied up in hardware, and the value of cloud-native services (managed databases, serverless, AI APIs) that have no easy on-prem equivalent. There is no universal answer; the calculus differs by company size, load profile, and engineering capacity.
- **Kubernetes: essential platform or complexity tax?** Kubernetes is genuinely powerful and has become the industry standard for container orchestration. It is also genuinely complex to operate correctly. A vocal contingent argues that most organizations adopting Kubernetes are not large enough or fast-moving enough to justify the operational overhead, and that simpler platforms (Fly.io, Railway, managed container services) deliver 80% of the benefit at 20% of

the complexity. The pro-Kubernetes camp responds that the ecosystem lock-in risk with simpler platforms is real, and that Kubernetes skills are transferable in a way that proprietary platforms are not.

- **eBPF: the next wave in observability?** A technology called **eBPF** (extended Berkeley Packet Filter) allows code to run safely inside the Linux kernel without modifying it. This has enabled a new generation of observability and security tools (Cilium, Tetragon, Pixie) that can trace exactly what every process is doing at kernel level with minimal overhead. Whether this represents a genuine paradigm shift or “just another observability layer” is a live debate among SREs and platform engineers as of 2026.

Reading on

- **The Phoenix Project** by Gene Kim, Kevin Behr, and George Spafford — a novel about an IT organization in crisis, widely credited with making DevOps and SRE thinking accessible to non-technical readers. The compute and virtualization concepts in this lesson are the backdrop for the story.
- **Google’s Site Reliability Engineering book** — sre.google/sre-book/table-of-contents. Free online. The first five chapters cover the philosophy of SRE and why reliability is an engineering discipline, not a job title. Readable without an engineering background.
- **Kubernetes.io “Concepts” overview** — kubernetes.io/docs/concepts/overview. The official documentation is surprisingly approachable. The “What is Kubernetes?” page gives the mental model in plain prose; the “Kubernetes Components” page maps exactly to the right panel of the diagram above.
- **Corey Quinn’s “Last Week in AWS”** — lastweekinaws.com. A newsletter that covers AWS news with healthy scepticism and dry humor. The back-catalogue is an excellent way to understand what cloud providers actually sell, why their pricing is the way it is, and where the genuine trade-offs lie.

LAST-REVIEWED: 2026-06-02