Cloud Platform Fundamentals

A Complete 12-Chapter Cloud Computing Course

Topics covered:

What the cloud actually is & the shared responsibility model · AWS/Azure/GCP compared

Compute, storage & networking fundamentals · Identity & access management

Databases, monitoring & logging · Cost management & billing

Security & compliance basics · Infrastructure as Code · Choosing a provider

Exercises: 36 hands-on exercises with worked solutions

Format: A4 · Dark-theme code examples · cross-provider throughout, framed for support engineers

Table of Contents

01 What "The Cloud" Actually Is

02 The Big Three, Compared

03 Compute Fundamentals

04 Storage Fundamentals

05 Networking Fundamentals

06 Identity & Access Management

07 Databases in the Cloud

08 Monitoring & Logging

09 Cost Management & Billing

10 Security & Compliance Basics

11 Infrastructure as Code — A First Look

12 Choosing & Comparing Providers, and Where to Go Next

CHAPTER 1 OF 12

What "The Cloud" Actually Is

Cloud Platform Fundamentals

Chapter 1 · What "The Cloud" Actually Is

This course is deliberately **cross-provider** rather than picking one vendor to specialize in — every concept chapter maps AWS, Azure, and GCP terminology side by side, because in real support work you rarely get to choose which cloud the customer or company happens to be running on. Course 2 goes further, into the specifically operational and troubleshooting skills a support engineer actually needs day to day. This chapter starts with the foundation everything else builds on: what "the cloud" actually means underneath the marketing term.

From On-Premises to "The Cloud"

Running software "on-premises" (on-prem) means an organization owns and operates its own physical servers, in its own building or a rented data center rack, handling everything from buying the hardware to replacing failed disks. "The cloud" replaces that with **renting** compute, storage, and networking from a provider's own massive data centers — Amazon (AWS), Microsoft (Azure), and Google (GCP) being the three dominant providers today.

Modern cloud computing is generally dated to AWS's 2006 launch of S3 (storage) and EC2 (virtual servers) — the first time renting raw infrastructure by the hour, at scale, became a practical, self-service option rather than something requiring a lengthy contract with a hosting provider.

The Service Models — IaaS, PaaS, SaaS

Cloud services are usually described by **how much of the stack you manage yourself** versus how much the provider manages for you:

MODEL	YOU MANAGE	PROVIDER MANAGES	EXAMPLES
IaaS	OS, runtime, app, data, config	Physical hardware, virtualization, network	EC2, Azure VMs, Compute Engine
PaaS	App code, data	OS, runtime, scaling, patching	Elastic Beanstalk, Azure App Service, App Engine

MODEL	YOU MANAGE	PROVIDER MANAGES	EXAMPLES
SaaS	Your own data and user access	Everything else, end to end	Gmail, Microsoft 365, Salesforce

Moving down this table (IaaS → PaaS → SaaS), you give up more control in exchange for less operational burden. Chapter 3 (Compute) goes deep specifically on the IaaS layer, since that's where most of the hands-on troubleshooting work in this course actually happens.

The Shared Responsibility Model

Every major provider frames security using the same underlying idea, usually summarized as security **"of"** the cloud versus security **"in"** the cloud:

- **The provider's responsibility ("of" the cloud)** — physical data center security, the underlying hardware, the virtualization/hypervisor layer, and the global network infrastructure.
- **The customer's responsibility ("in" the cloud)** — your own data, your IAM configuration (Chapter 6), your network configuration (security groups, firewall rules), and — depending on the service model above — your OS patching and application code.

Exactly where that line sits **shifts** depending on the service model: with IaaS, you're responsible for almost everything above the hypervisor; with SaaS, you're mainly responsible for your own data and who has access to it.

ARGUABLY THE SINGLE MOST USEFUL CONCEPT IN THIS CHAPTER FOR SUPPORT WORK

A large share of "the cloud is down" tickets turn out, on investigation, to be a misconfiguration sitting entirely on the customer's own side of the responsibility line — an overly permissive security group, a misconfigured IAM policy, an application bug — not an actual provider outage. Knowing where that line sits, and checking it first, is one of the fastest ways to correctly triage an incident before escalating it. Chapter 5's troubleshooting flowchart in Course 2 builds directly on this distinction.

"MANAGED SERVICE" DOESN'T MEAN "ZERO RESPONSIBILITY"

Even at the SaaS end of the spectrum, where the provider manages essentially everything technical, you're still fully responsible for your own data and access management. An employee's leaked SaaS login, or a document shared with the wrong permissions, is still your organization's problem to handle — the provider secured the platform, not your usage of it.

Why Organizations Migrate to the Cloud

- **Elasticity** — scaling capacity up or down on demand, rather than provisioning for peak load year-round.

- **CapEx to OpEx** — converting large upfront hardware purchases into an ongoing operating expense, paid as you use it.
- **Global reach** — running infrastructure close to customers worldwide without building physical data centers in every region.
- **Speed of provisioning** — a new server ready in minutes, rather than the weeks or months on-prem hardware procurement often takes.
- **Managed services** — offloading operational burden (patching, backups, scaling) onto the provider.

Honestly, though: cloud isn't automatically cheaper, and unmanaged usage can grow costs unpredictably — Chapter 9 covers cost management as its own dedicated topic, not an afterthought. Vendor lock-in (services and APIs specific to one provider, making a future migration harder) is also a genuine, real trade-off worth weighing rather than ignoring.

Public, Private & Hybrid Cloud

- **Public cloud** — shared infrastructure operated by AWS/Azure/GCP, used by many different customers (this course's main focus).
- **Private cloud** — dedicated infrastructure for a single organization, whether on-prem or hosted by a third party.
- **Hybrid cloud** — a deliberate mix of both, often for compliance requirements, latency-sensitive workloads, or legacy systems that aren't practical to migrate.

Hybrid and multi-cloud environments come with their own real support challenges — connectivity between environments, consistent monitoring across both — covered properly in Course 2's `cloud2-9`.

What This Course Covers

CHAPTER	TOPIC
2	The Big Three, Compared
3	Compute Fundamentals
4	Storage Fundamentals
5	Networking Fundamentals
6	Identity & Access Management
7	Databases in the Cloud
8	Monitoring & Logging

CHAPTER	TOPIC
9	Cost Management & Billing
10	Security & Compliance Basics
11	Infrastructure as Code — A First Look
12	Choosing & Comparing Providers, and Where to Go Next

Hands-On Exercises

EXERCISE 1

Classify each scenario as IaaS, PaaS, or SaaS: (a) "we rent virtual machines and manage our own OS patching and web server config," (b) "we just push our application code and the platform handles scaling and runtime patching automatically," (c) "we use a web-based CRM tool and never think about servers at all."

View solution

EXERCISE 2

A customer's cloud storage bucket was left publicly readable by mistake, and data was accessed by an unauthorized party. Using the shared responsibility model, whose side of the line does this fall on — the provider's or the customer's — and why?

View solution

EXERCISE 3

Give at least two legitimate reasons an organization might deliberately choose a hybrid cloud setup rather than moving everything to the public cloud.

View solution

CHAPTER 1 QUICK REFERENCE

- **On-prem** = you own the hardware; **cloud** = you rent it, at scale, self-service, by the hour
- **IaaS/PaaS/SaaS** — how much of the stack you manage vs. the provider does; moving down the list trades control for less operational burden
- **Shared responsibility model** — provider secures "of" the cloud (hardware, hypervisor, network); customer secures "in" the cloud (data, IAM, config); the line shifts by service model

- A huge share of "outage" tickets are actually customer-side misconfigurations — check the responsibility line first
- Migration drivers: elasticity, CapEx→OpEx, global reach, provisioning speed, managed services — with real trade-offs (cost unpredictability, vendor lock-in)
- **Public/private/hybrid cloud** — hybrid exists for real reasons (compliance, latency, legacy systems), not indecision
- **Next chapter:** The Big Three, Compared — AWS/Azure/GCP terminology mapping, the habit this whole course relies on

CHAPTER 2 OF 12

The Big Three, Compared

Cloud Platform Fundamentals

Chapter 2 · The Big Three, Compared

This chapter builds the single habit the rest of this course leans on constantly: mapping the same underlying concept across AWS, Azure, and GCP's three different sets of names for it.

Why Terminology Mapping Matters for Support Work

A support engineer doesn't get to pick which cloud a given customer or team happens to run on. The good news: the underlying concepts across all three major providers are remarkably similar — a virtual machine is a virtual machine everywhere. The friction is almost entirely in the **naming**. Being able to quickly translate "this sounds like an EC2 problem" into the Azure or GCP equivalent, without re-learning the concept from scratch each time, is a genuinely practical skill this chapter is built around.

Market Landscape & History

- **AWS** — launched 2006 (S3, EC2), the first mover, and still the largest by market share today.
- **Azure** — launched 2010, with especially deep integration into the Microsoft enterprise ecosystem (Active Directory, Microsoft 365), a major draw for organizations already invested there.
- **GCP** — launched 2008 (App Engine) and ramped up significantly after 2011, with particular strength in data/ML tooling — and, notably, GCP is where Kubernetes itself originated internally before Google open-sourced it.

Rough market share ordering is AWS > Azure > GCP, but that ranking isn't the whole story — plenty of organizations use more than one provider, and "market leader" doesn't mean "the only one worth knowing."

The Core Terminology Map

The single most useful practical artifact in this chapter — worth bookmarking and referring back to throughout the rest of this course:

CONCEPT	AWS	AZURE	GCP
Virtual machine	EC2	Azure Virtual Machines	Compute Engine

CONCEPT	AWS	AZURE	GCP
Object storage	S3	Blob Storage	Cloud Storage
Block storage	EBS	Managed Disks	Persistent Disk
Virtual network	VPC	Virtual Network (VNet)	VPC
Identity service	IAM	Microsoft Entra ID (Azure AD)	Cloud IAM
Managed relational DB	RDS	Azure SQL Database	Cloud SQL
Managed NoSQL DB	DynamoDB	Cosmos DB	Firestore / Bigtable
Serverless functions	Lambda	Azure Functions	Cloud Functions
Load balancer	ELB / ALB	Azure Load Balancer	Cloud Load Balancing
DNS service	Route 53	Azure DNS	Cloud DNS
Monitoring/logging	CloudWatch	Azure Monitor	Cloud Monitoring / Logging
Managed Kubernetes	EKS	AKS	GKE

Structural Differences Worth Knowing

Naming isn't the only difference — a few structural ones genuinely matter for finding your way around:

- **Regions & zones** — all three organize physical infrastructure into *regions* (geographic areas) containing multiple *availability zones* or *zones* (isolated data centers within that region). AWS has the largest region count historically; GCP has been expanding rapidly to close that gap.
- **Resource organization hierarchy** — AWS uses Organizations → Accounts; Azure uses Management Groups → Subscriptions → **Resource Groups**; GCP uses Organizations → Folders → Projects. Azure's Resource Group concept — a container for grouping related resources by lifecycle, deleted together — has no clean 1:1 equivalent on AWS or GCP, which matters practically: "where do I even look for this resource" genuinely differs by provider, not just what it's called once you find it.

> **"ROUGHLY EQUIVALENT," NOT "IDENTICAL"**
>
> The terminology map above is a genuinely useful starting point, not a guarantee of identical behavior. Two services with a clean name-to-name mapping can still differ in real, troubleshooting-relevant ways — default limits, permission models, or exactly how they interact with other services. Treat the mapping as "start here," not "this always works exactly the same."

Free Tiers & Getting Hands-On

All three providers offer a free tier or trial credit for learning and experimentation — the AWS Free Tier, an Azure free account, and GCP's free trial credit. Genuinely practical advice for a support role: having a sandbox account on all three, and actually poking around in each console directly, builds far more real intuition than reading terminology tables alone.

Certifications, Briefly

Each provider runs its own certification track — AWS Certified, Microsoft Azure certifications, and Google Cloud certifications — which some support roles value or explicitly require. This course isn't a certification prep track, but it's worth knowing these exist as a next step once the fundamentals here feel solid.

Hands-On Exercises

EXERCISE 1

A ticket describes an issue with "an S3 bucket policy blocking access, and an EC2 instance in a VPC that can't reach it." Translate this scenario into its Azure equivalents and its GCP equivalents, service by service.

View solution

EXERCISE 2

Explain why "AWS has the largest market share" doesn't mean a support engineer can safely learn only AWS and ignore Azure and GCP.

View solution

EXERCISE 3

Explain what a resource organization hierarchy (like AWS Organizations/Accounts or Azure's Management Groups/Subscriptions/Resource Groups) is actually for, and why Azure's Resource Group concept specifically is called out as not having a clean 1:1 equivalent elsewhere.

View solution

CHAPTER 2 QUICK REFERENCE

- **AWS** (2006, largest share) · **Azure** (2010, deep Microsoft-ecosystem ties) · **GCP** (2008/2011, strong in data/ML, birthplace of Kubernetes)

- The terminology map (VM, object/block storage, VPC, IAM, managed DBs, serverless, load balancer, DNS, monitoring, managed Kubernetes) is this course's core reusable artifact
- **Resource hierarchy** differs structurally, not just by name — Azure's Resource Group has no clean AWS/GCP equivalent
- Treat cross-provider mappings as "roughly equivalent," not identical — real behavioral differences exist underneath matching names
- All three offer a free tier/trial — genuinely worth a hands-on sandbox account on each
- **Next chapter:** Compute Fundamentals — VMs/instances in depth, the IaaS layer this course spends the most hands-on time on

CHAPTER 3 OF 12

Compute Fundamentals

Cloud Platform Fundamentals

Chapter 3 · Compute Fundamentals

Chapter 1 named IaaS as the layer this course spends the most hands-on time on — this chapter is why: virtual machines are the most common thing a support engineer actually gets tickets about, and the concepts here (sizing, lifecycle states, pricing model, auto-scaling) recur constantly in real troubleshooting work.

What a Virtual Machine Actually Is

A virtual machine is a software-emulated computer running on top of a **hypervisor** — a layer that lets one physical server safely host many isolated VMs at once, each unaware of the others. Cloud providers use bare-metal (type 1) hypervisors running directly on the physical hardware, rather than on top of a host operating system.

Worth distinguishing early: VMs virtualize the *hardware* — each one runs its own full operating system. Containers (this site's own `docker1` / `docker2` courses) virtualize the *operating system* instead — much lighter weight, but a genuinely separate topic this course doesn't go deep on.

Launching a VM — What You Actually Choose

Spinning up a VM on any of the three providers involves the same core decisions, just under different menu names:

1. **Machine image** — a preconfigured OS-plus-software snapshot (an AMI on AWS, a VM image on Azure, an image on GCP).
2. **Instance type/size** — the CPU/RAM/network combination (Chapter 3's own next section).
3. **Region and zone** — where physically the VM runs.
4. **Attached storage** — covered fully in Chapter 4.
5. **Networking and security group assignment** — covered fully in Chapter 5.
6. **Access credentials** — an SSH key pair (Linux) or equivalent login credential, set at launch time.

Instance Families & Sizing

Every provider groups instance types into similar families, just with different names:

FAMILY	AWS EXAMPLE	AZURE EXAMPLE	GCP EXAMPLE
General purpose	t3.medium	Standard_B2s	e2-medium
Compute-optimized	c6i.large	Fsv2-series	c2-standard-4
Memory-optimized	r6i.large	Esv5-series	m1-megamem
GPU	p4d	NC-series	a2-highgpu

Sizing is fundamentally a vCPU + RAM + network throughput trade-off. Undersizing causes real performance problems and timeouts under load; oversizing simply wastes money — Chapter 9 covers cost management as its own topic, but the sizing decision made here is where that cost is actually determined.

Instance Lifecycle & States

Every provider's VMs move through the same core states: **running**, **stopped**, and **terminated** (or deleted). The distinction between stopped and terminated is genuinely important, not just semantic:

- **Stopped** — the VM isn't running, and you stop paying for compute — but its attached storage still exists and is still billed.
- **Terminated/deleted** — the instance and (usually) its storage are gone permanently, unless a snapshot was taken beforehand.

> **ONE OF THE MOST COMMON BILLING-SUPPORT TICKETS IN PRACTICE**
>
> "My VM is stopped but I'm still being billed" is a genuinely frequent support scenario, and the answer is almost always this exact distinction: stopping a VM halts compute charges, but the attached storage volume keeps existing — and keeps costing money — until it's explicitly deleted. This single fact resolves a large share of "why am I still being charged" tickets.

Pricing Models

- **On-demand / pay-as-you-go** — full flexibility, no commitment, the highest per-hour cost.
- **Reserved / committed use** — a meaningful discount in exchange for a 1-3 year usage commitment, appropriate for predictable, always-on workloads.
- **Spot / preemptible instances** — a deep discount, but the provider can reclaim the instance with very little notice.

> **SPOT INSTANCES ARE NOT A FREE DISCOUNT**

> Spot (AWS)/preemptible (GCP) instances can be reclaimed with as little as a couple of minutes' notice, sometimes less. They're an excellent fit for fault-tolerant, interruptible workloads — batch processing, rendering, CI jobs — and a genuinely bad fit for stateful, always-on production services with no fallback plan. Choosing spot pricing for the wrong workload trades a cost saving for a real, recurring availability risk.

Auto-Scaling — Elasticity in Action

This is Chapter 1's "elasticity" made concrete. Auto-scaling groups (AWS Auto Scaling Groups, Azure VM Scale Sets, GCP Managed Instance Groups) automatically add or remove VM instances based on defined metrics — CPU utilization, request count, or similar. The core mechanism is the same everywhere: a minimum, maximum, and desired instance count, a scaling policy defining what triggers a change, and automatic health checks that replace unhealthy instances without manual intervention. Auto-scaling groups typically sit behind a load balancer (Chapter 5) that distributes incoming traffic across whatever the current instance count happens to be.

A Support-Relevant Gotcha — Auto-Scaling Masking Real Problems

Auto-scaling is genuinely useful, but it has a real operational trap: it can quietly compensate for an underlying problem — a memory leak, inefficient code, an unoptimized query — by simply adding more instances, rather than the problem ever actually getting fixed. Cost climbs steadily in the background while the root cause goes unaddressed, until the auto-scaling group eventually hits its configured maximum instance count — at which point the original symptom reappears, now at a larger and more urgent scale than if it had been investigated in the first place.

Hands-On Exercises

EXERCISE 1

Explain the difference between a "stopped" and a "terminated" VM, and explain specifically why a customer might see continued charges on their bill despite insisting their VM has been "stopped, not running" for weeks.

View solution

EXERCISE 2

Recommend on-demand, reserved, or spot pricing for each: (a) a batch video-transcoding job that checkpoints its progress and can safely restart if interrupted, (b) a database server that needs to run continuously for the next two years, (c) an unpredictable dev/test environment used only sporadically.

📄 View solution

EXERCISE 3

In your own words, explain how auto-scaling can mask a real underlying problem rather than fix it, and describe what a support engineer should investigate before simply accepting "the auto-scaler handled it" as a resolution.

📄 View solution

CHAPTER 3 QUICK REFERENCE

- **VMs** virtualize hardware (full OS each); **containers** virtualize the OS (see `docker1` / `docker2`) — a separate topic
- Launching a VM = image + instance size + region/zone + storage (Ch.4) + networking (Ch.5) + access credentials
- Instance families (general/compute/memory/GPU-optimized) exist under different names across all three providers
- **Stopped ≠ terminated** — stopped still bills for attached storage; this explains a huge share of real billing tickets
- **On-demand** (flexible, priciest) vs. **reserved** (discounted, committed) vs. **spot/preemptible** (cheapest, reclaimable — wrong fit for stateful production)
- **Auto-scaling** (ASG/Scale Sets/Managed Instance Groups) — min/max/desired count, triggers, health checks; can mask a root cause instead of fixing it
- **Next chapter:** Storage Fundamentals — object vs. block storage, and storage tiers/lifecycle policies

CHAPTER 4 OF 12

Storage Fundamentals

Cloud Platform Fundamentals

Chapter 4 · Storage Fundamentals

Chapter 3 mentioned "attached storage" as one of the choices made when launching a VM without explaining it — this chapter covers storage properly, both the storage attached directly to a VM and the separate, independent storage services that exist alongside it.

Object Storage vs. Block Storage — Two Different Models

Cloud storage splits into fundamentally different models, not just different products:

	BLOCK STORAGE	OBJECT STORAGE
What it looks like	A raw disk volume, mounted as a filesystem	A flat namespace of "objects," accessed via an HTTP API
Access pattern	Low-latency, random-access reads/writes in fixed blocks	Whole-object reads/writes, higher per-request latency
Scale	Sized per volume, attached to one VM at a time	Virtually unlimited, massively parallel
Typical use	OS boot volumes, databases	Backups, static assets, logs, data lakes

Object Storage — S3, Blob Storage & Cloud Storage

Applying Chapter 2's terminology map: S3 (AWS), Blob Storage (Azure), and Cloud Storage (GCP) all organize data into **buckets** (or containers) holding **objects** — each object being the actual data plus a key/name and metadata. Access happens through a REST API or SDK, **not** a traditional filesystem mount — a common misconception worth clarifying directly, though tools exist (FUSE-based mounts, gateway appliances) that can simulate filesystem-style access on top of the underlying API.

Object storage is typically engineered for extremely high **durability** — commonly advertised around "11 nines" (99.999999999%) — via automatic replication across multiple facilities within a region. Durability and **availability** are genuinely different claims, worth keeping separate: durability means the data itself won't be lost; availability means the data is currently reachable when requested. A brief regional service disruption can affect availability without the underlying data ever being at risk of loss.

Block Storage — EBS, Managed Disks & Persistent Disk

Block storage (EBS on AWS, Managed Disks on Azure, Persistent Disk on GCP) attaches directly to a single VM as a virtual disk, used for OS boot volumes and anything needing low-latency, random-access I/O — databases especially. It persists independently of the VM's own lifecycle: a volume can be detached from one VM and reattached to another, and — directly echoing Chapter 3's stopped-VM billing gotcha — it continues to exist, and continues billing, even while its VM is stopped.

Snapshots capture a point-in-time backup of a block volume — incremental after the first full snapshot, and typically stored in the object storage layer underneath.

A Third Option, Briefly — File Storage

Neither block nor object storage directly solves one common need: **shared, simultaneous access from multiple VMs** at once. That's what network file storage (EFS on AWS, Azure Files, Filestore on GCP) is specifically for — a mountable, shared filesystem multiple VMs can read and write to concurrently, used for shared application data, home directories, or content management systems that genuinely need simultaneous multi-VM access. It's a real, distinct third category worth knowing exists, even though this course doesn't go deep on it.

Choosing Between Them — A Practical Decision Table

NEED	RIGHT CHOICE
OS boot disk / low-latency database storage	Block storage
Large numbers of files, backups, static assets, cost-effective at scale	Object storage
Shared access from multiple VMs simultaneously	File storage

Storage Tiers & Lifecycle Policies

Object storage typically offers multiple tiers trading cost against retrieval speed: standard/frequent-access, infrequent-access, and archive/cold storage (S3 Standard/IA/Glacier; Azure Hot/Cool/Archive; GCP Standard/Nearline/Coldline/Archive). **Lifecycle policies** automatically move objects between tiers — or delete them entirely — after a defined age, a genuinely practical cost-management tool that Chapter 9 builds on further.

ARCHIVE TIER RETRIEVAL ISN'T INSTANT, OR FREE

Retrieving data from an archive/cold storage tier can take anywhere from minutes to several hours, and often carries a real retrieval cost on top of the storage savings that made archiving attractive in the first place. A customer expecting the same instant access they get from standard-tier storage is a genuinely common source of confusion — and a support conversation worth having proactively, before a lifecycle policy quietly moves data somewhere slower to retrieve.

Data Durability, Redundancy & Replication

Both block and object storage typically offer a choice of redundancy — single-zone, multi-zone, or cross-region replication — trading additional cost for protection against a zone- or region-level failure, directly building on Chapter 1's regions/availability-zones concept.

DURABILITY VS. AVAILABILITY, RESTATED FOR SUPPORT CONVERSATIONS

"Your data is safe" (durability) and "your data is currently reachable" (availability) are different claims, and worth keeping separate when talking to a worried customer. A temporary regional service disruption affecting availability doesn't necessarily mean any data was lost — the replication underneath object storage's durability guarantee is often still intact even during a visible outage.

Hands-On Exercises

EXERCISE 1

Classify each storage need as block, object, or file storage, and justify each choice: (a) an OS boot disk for a VM, (b) millions of small log files that must be retained for compliance for 7 years, (c) a shared directory accessed simultaneously by 10 VMs.

View solution

EXERCISE 2

Explain the difference between durability and availability, and give one concrete example incident for each — one that affects durability, and one that affects availability without touching durability at all.

View solution

EXERCISE 3

A customer is surprised that restoring an old backup took several hours and incurred an unexpected extra charge. Explain why this happened, and what a support engineer should proactively communicate about lifecycle policies before this becomes a surprise.

📄 View solution

CHAPTER 4 QUICK REFERENCE

- **Block storage** — mounted, low-latency, one VM at a time (OS disks, databases); **object storage** — HTTP API, massively scalable (backups, assets, logs)
- **File storage** — the third option, mountable and shared across multiple VMs simultaneously (EFS/Azure Files/Filestore)
- Block volumes persist and keep billing independently of the VM, even while stopped — a direct callback to Chapter 3's billing gotcha
- **Durability** (will the data survive) ≠ **availability** (can it be reached right now) — a real distinction worth using in support conversations
- **Lifecycle policies** automatically move objects to cheaper, slower tiers or delete them — archive-tier retrieval is neither instant nor free
- Redundancy (single-zone/multi-zone/cross-region) trades cost for protection against zone/region failure
- **Next chapter:** Networking Fundamentals — VPCs/VNets, subnets, load balancers, and DNS

CHAPTER 5 OF 12

Networking Fundamentals

Cloud Platform Fundamentals

Chapter 5 · Networking Fundamentals

Compute (Chapter 3) and storage (Chapter 4) are now covered — this chapter is about how everything actually talks to everything else, and to the outside world. It's arguably the single most support-relevant chapter in this course: "I can't connect to X" is probably the most common category of real cloud support ticket there is.

Regions & Availability Zones, Revisited

A quick recap from Chapters 1 and 4: regions are geographic areas, and availability zones (AZs) are isolated data centers within a region. For networking specifically, this matters because resources in the same region but different AZs still need connectivity between them — the provider's own backbone network handles that transparently. Latency naturally increases with geographic distance between regions, a real factor when placing resources relative to their users.

Virtual Networks — VPCs & VNets

A VPC (AWS/GCP) or VNet (Azure) is a private, logically isolated network you define within a region, with an IP address range you choose (a CIDR block). By default, it's isolated from every other customer's network — this is the foundational network security boundary everything else in this chapter builds on top of.

Subnets

A VPC/VNet is divided into smaller segments called subnets, typically one per availability zone. The most important distinction: a **public subnet** has a route to an internet gateway; a **private subnet** doesn't (or only routes outbound through a NAT gateway, covered below).

This split exists for a genuine architectural reason: web servers commonly sit in a public subnet, directly reachable from the internet, while databases sit in a private subnet, *not* directly reachable from the internet at all. A support ticket asking "why can't I reach my database directly from outside" is very often this exact pattern working correctly, by design — not a bug to fix.

Security Groups & Network ACLs — The Traffic Gatekeepers

	SECURITY GROUPS	NETWORK ACLS
Applies to	Individual instances/resources	An entire subnet
Stateful?	Yes — return traffic automatically allowed	No — return traffic must be explicitly allowed too
Rule types	Allow rules only (typically)	Both allow and deny rules

A misconfigured security group blocking the wrong port is very likely the single most common real-world networking support issue there is. The practical debugging order: check the resource's security group inbound rules first, then the subnet's network ACL, then routing.

> **CHECKING ONLY ONE OF THE TWO IS A COMMON MISTAKE**
>
> Because security groups are stateful and NACLs aren't, it's easy to fix a security group rule, confirm traffic is now allowed at that level, and stop looking — while a subnet-level NACL is *also* silently blocking the exact same traffic. Both layers need to explicitly permit a connection; either one alone can block it.

Routing & Gateways

- **Route tables** — direct traffic leaving a subnet to its correct destination.
- **Internet gateway** — provides a route to/from the public internet for a public subnet.
- **NAT gateway** — lets a private subnet reach the internet *outbound only*, without being directly reachable from it.
- **VPC peering** — connects two VPCs together directly.
- **Transit gateway / hub-and-spoke** — connects many VPCs through a central hub, rather than peering each pair individually.

This vocabulary is picked up again properly in Course 2's connectivity-troubleshooting chapter (`cloud2-2`).

Load Balancers

Revisiting Chapter 3's brief mention: a load balancer distributes incoming traffic across multiple backend instances, using **health checks** to route traffic only to instances currently reporting healthy — the exact same health-check mechanism Chapter 3 described for auto-scaling groups, working together with it.

LAYER	WHAT IT SEES	AWS	AZURE	GCP
Layer 4 (network)	Raw TCP/UDP connections	NLB	Load Balancer	Network LB

LAYER	WHAT IT SEES	AWS	AZURE	GCP
Layer 7 (application)	HTTP-aware — can route by URL path, host header, etc.	ALB	Application Gateway	HTTP(S) LB

DNS in the Cloud

Route 53 (AWS), Azure DNS, and Cloud DNS (GCP) provide hosted DNS zones for your own domains, and are also commonly used for internal service discovery within a VPC. A genuinely frequent support scenario: a customer makes a DNS change and asks why it hasn't taken effect everywhere yet. The answer is almost always **TTL** (time to live) — a value on each DNS record controlling how long resolvers around the internet are allowed to cache it before checking again. A record with a one-hour TTL can take up to an hour to fully propagate to every resolver that had it cached, purely by design.

A PRACTICAL "CAN'T CONNECT" CHECKLIST

Security group → network ACL → route table → DNS. Working through this order systematically catches the overwhelming majority of real connectivity tickets, and Course 2's own troubleshooting chapter (`cloud2-2`) builds directly on this exact sequence with a full worked flowchart.

Hands-On Exercises

EXERCISE 1

Explain the difference between a security group and a network ACL — what each applies to, whether each is stateful, and specifically why "return traffic is automatically allowed" is true for one but not the other.

View solution

EXERCISE 2

A web server sitting in a public subnet cannot be reached on port 443 from the internet. List, in order, the checks you'd perform to diagnose this, and explain what each check rules out.

View solution

EXERCISE 3

Explain what TTL is, and why a DNS record change might not be visible to all users immediately after it's made.

View solution

CHAPTER 5 QUICK REFERENCE

- **VPC/VNet** — a private, isolated network within a region; **subnets** divide it, usually per AZ
- **Public subnet** (internet gateway route) vs. **private subnet** (no direct inbound route, often NAT for outbound) — databases-in-private/webservers-in-public is a deliberate pattern, not a bug
- **Security groups** (stateful, per-resource) vs. **NACLs** (stateless, per-subnet) — both must allow traffic; checking only one is a common mistake
- **Route tables/internet gateway/NAT gateway/VPC peering/transit gateway** — the routing vocabulary picked up again in `cloud2-2`
- **Load balancers** use the same health checks as auto-scaling (Ch.3); Layer 4 (TCP) vs. Layer 7 (HTTP-aware)
- **DNS TTL** controls propagation delay — the standard explanation for "my DNS change hasn't taken effect everywhere yet"
- Practical troubleshooting order: security group → NACL → route table → DNS
- **Next chapter:** Identity & Access Management — IAM concepts across providers, least privilege, and MFA

CHAPTER 6 OF 12

Identity & Access Management

Cloud Platform Fundamentals

Chapter 6 · Identity & Access Management

Chapter 5 covered *where* traffic can go. This chapter covers *who* can do *what*, once they're already in — a genuinely different security layer, and per Chapter 1's shared responsibility model, IAM configuration sits squarely on the **customer's** side of the line, regardless of which service model (IaaS/PaaS/SaaS) is in use.

IAM's Core Building Blocks

- **Users** — individual identities, typically representing a specific person.
- **Groups** — collections of users that share the same set of permissions.
- **Roles** — an identity *assumed temporarily*, often by a service or application rather than a person (its own dedicated section below).
- **Policies** — documents (typically JSON) defining what actions are allowed or denied, on which resources.

Terminology diverges more here than in most of Chapter 2's mapping table: AWS uses IAM users/groups/roles/policies fairly directly; Azure uses Entra ID (formerly Azure AD) for users/groups, with separate Azure RBAC role assignments layered on top; GCP uses "members" (its umbrella term for any identity) combined with "roles," which in GCP specifically means a bundle of permissions assigned to a member — a subtly different use of the word "role" than AWS's assumable-identity meaning, worth keeping straight.

Roles — Why They're a Cloud-Native Concept

A role has no permanent credentials of its own — it's **assumed** temporarily, generating short-lived credentials that expire automatically. This solves a real, extremely common problem: an application running on a VM that needs to read from a storage bucket (Chapter 4) doesn't need a permanent access key embedded anywhere in its code or configuration at all — it can simply assume a role with exactly the permissions it needs, for exactly as long as it needs them.

HARDCODED CREDENTIALS REMAIN A SERIOUS, COMMON REAL MISTAKE

Embedding a permanent access key directly in application code or a config file — worse, committing it to a source repository — is a genuinely frequent real-world security failure, directly echoing this site's own `pipelines1-5` ("a committed credential is compromised forever") and `crypto1-11`'s key management chapter. Using a role instead of

a static credential removes this risk entirely: there's no long-lived secret sitting in code for anyone to accidentally expose in the first place.

Roles are also used for cross-account access — letting a trusted identity in one account temporarily assume a role in another, without needing separate permanent credentials for every account involved.

The Principle of Least Privilege

This is exactly this site's own `dbsec1-3` lesson, applied directly to cloud IAM: one identity per purpose, granular grants scoped to specific resources, and avoiding broad "allow everything" policies. The "just grant AdministratorAccess, it's easier" anti-pattern is genuinely common in practice, and genuinely risky — it turns any single compromised credential into a compromise of the entire account, rather than one narrow slice of it.

Real policy scoping goes further than just which actions are allowed — **resource-level restrictions** (access to this specific storage bucket, not every bucket in the account) and **condition keys** (restricting a grant by source IP address, time of day, or similar context) both narrow a policy's real-world blast radius considerably.

Authentication vs. Authorization in Cloud IAM

This site's own `bc1-1` distinction applies directly: **authentication** is proving who you are (username and password, MFA, federated login); **authorization** is what that already-verified identity is actually allowed to do, determined by the policies attached to it.

This is a genuinely useful support-triage distinction — a "permission denied" or "access denied" error is an **authorization** problem, not a login problem, even though confused end users very commonly describe the two identically ("I'm logged in but I can't do X"). Recognizing that "logged in fine, action denied" almost always points at a missing or misconfigured policy — not a broken login — saves real troubleshooting time.

Multi-Factor Authentication (MFA)

Revisiting `bc1-6`'s MFA coverage specifically for cloud console access: TOTP apps, hardware keys, and push notifications all apply here exactly as described there. One practice worth calling out as close to non-negotiable across every provider: **enable MFA on the root/owner account specifically**, since that account typically has unrestricted privileges that can't be scoped down by ordinary IAM policies the way a regular user's access can.

ROOT ACCOUNT MFA, RESTATED PLAINLY

Every ordinary IAM user's damage potential can be limited through least-privilege policies. The root/owner account is the one identity that typically can't be restricted that way — which is exactly why MFA on that specific account is treated as an absolute baseline, not an optional hardening step, across AWS, Azure, and GCP alike.

Federated Identity & Single Sign-On (SSO)

Rather than creating a separate cloud-native user account for every employee, organizations commonly federate identity from an existing corporate identity provider (Active Directory, Okta, or similar) via SAML or OIDC — so employees log in with their existing corporate credentials, and access is centrally managed from one place. Genuinely common in real enterprise environments; this course doesn't go deep on the protocol mechanics, but it's worth knowing the concept exists and why organizations reach for it.

A Support-Relevant IAM Troubleshooting Pattern

Facing an "access denied" error, the first question is exactly the AuthN-vs-AuthZ split above: can the user log in at all (authentication), or are they logged in fine but blocked from a specific action (authorization)? If it's authorization, the next question is: what policy is actually attached to this identity, and does it explicitly allow this specific action on this specific resource?

AN EXPLICIT DENY ALWAYS WINS

Across all three providers, policies aren't purely additive — if *any* attached policy contains an explicit deny for an action, that deny overrides an allow granted anywhere else, even a broad allow from a separate policy. A genuinely common, real troubleshooting trap is granting broad access via one policy, then being confused why a specific resource is still blocked — the answer is very often a separate, explicit deny sitting in another attached policy that nobody remembered was there.

Hands-On Exercises

EXERCISE 1

A user reports: "I'm logged in fine, but I get 'access denied' trying to do X." Is this an authentication or authorization problem, and what should be checked next?

View solution

EXERCISE 2

Explain why using an IAM role (rather than embedding a permanent access key directly in application code) is the correct pattern for an application that needs to read from a storage bucket, and connect this to the real-world risk of

hardcoded credentials.

View solution

EXERCISE 3

A user has a policy granting them broad access to a service, but a separate, attached policy explicitly denies access to one specific resource within that service. What happens when they try to access that resource, and why?

View solution

CHAPTER 6 QUICK REFERENCE

- **Users/groups/roles/policies** — terminology diverges more here than most of Ch.2's map, especially GCP's "role" meaning
- **Roles** are temporary, assumable identities with no permanent credentials — the correct pattern instead of hardcoded access keys (`pipelines1-5`, `crypto1-11`)
- **Least privilege** — `dbsec1-3`'s lesson applied to cloud IAM; resource-level scoping and condition keys narrow blast radius
- **AuthN vs. AuthZ** (`bc1-1`) — "logged in but denied" is almost always an authorization problem, not a login problem
- **MFA on the root/owner account** is close to non-negotiable — that account typically can't be scoped down by IAM policies at all
- **Federated identity/SSO** — centralizing access via an existing corporate identity provider (SAML/OIDC)
- **An explicit deny always wins** — policies aren't purely additive; one deny anywhere overrides an allow elsewhere
- **Next chapter:** Databases in the Cloud — managed relational and NoSQL options across providers

CHAPTER 7 OF 12

Databases in the Cloud

Cloud Platform Fundamentals

Chapter 7 · Databases in the Cloud

Deliberately brief, by design: this chapter doesn't re-teach SQL or NoSQL query writing — this site's own `mysql2`/`mysql3` and `mongodb1`/`mongodb2` courses already cover that in real depth. This chapter is specifically about what changes when a database becomes a **managed cloud service** rather than something installed and operated by hand.

What "Managed" Actually Means for a Database

A managed database service sits at the PaaS layer of Chapter 1's IaaS/PaaS/SaaS spectrum, even when it's used alongside plain IaaS compute elsewhere in the same architecture. The provider handles patching, backup scheduling, replication setup, and failover — but per Chapter 1's shared responsibility model, schema design, query writing, and the data itself remain entirely the customer's responsibility.

Managed Relational Databases

RDS (AWS), Azure SQL Database, and Cloud SQL (GCP) run the exact same database engines already covered elsewhere on this site — MySQL, PostgreSQL, SQL Server — just operated by the provider. This has a genuinely direct practical consequence: every bit of `mysql2`/`mysql3`'s SQL knowledge transfers completely unchanged. Only the *operational* layer changes — who handles backups, patching, and scaling — not the query language or schema design underneath.

Two genuinely useful managed features: **automated backups with point-in-time recovery**, and **Multi-AZ/high-availability deployments** — a standby replica maintained in a different availability zone (Chapter 5), with automatic failover if the primary instance fails.

Managed NoSQL Databases

DynamoDB (AWS), Cosmos DB (Azure), and Firestore (GCP) are **not** simply "MongoDB, but managed." DynamoDB and Cosmos DB are each proprietary services with their own distinct data models and APIs — though Cosmos DB does offer a genuine MongoDB-compatible API mode worth knowing about. Firestore is

Google's own document-database service, conceptually close to `mongodb1-1`'s document model (documents, collections, embedding vs. referencing) but not the same product or API as MongoDB itself.

The document-model concepts from `mongodb1-1` — schema flexibility, embedding vs. referencing trade-offs — remain broadly useful thinking tools across all of these services, even though the specific query API differs from provider to provider.

Read Replicas & Scaling Reads

A **read replica** is a read-only copy of a database used to offload read traffic away from the primary instance handling writes — a scaling tool. This is genuinely easy to confuse with the Multi-AZ/HA standby replica described above, which exists specifically for *failover*, not scaling, even though the underlying replication mechanism is often similar. A standby typically isn't meant to serve regular application read traffic; a read replica typically isn't automatically promoted on primary failure the way an HA standby is. Some providers do offer configurations blending both roles — but the two purposes are distinct by default and worth keeping separate.

Connection Management — A Common Support Issue

"Connection refused" and "too many connections" are genuinely frequent real tickets, usually from one of two causes:

- **Connection pool exhaustion** — a managed database instance has a maximum connection limit based on its size. An application opening connections without properly pooling or closing them can exhaust that limit, causing failures for *other* legitimate connections too, not just its own.
- **A blocking security group** — directly echoing Chapter 5's networking chapter: a database sitting in a private subnet, unreachable from outside it, is very often working exactly as designed, not broken.

> **NEVER MAKE A DATABASE PUBLICLY REACHABLE TO "FIX" A CONNECTIVITY TEST**
>
> A database that's unreachable because it's correctly placed in a private subnet (Chapter 5) should stay that way. The correct fix for needing occasional external access is a bastion host, a VPN connection, or an application-tier proxy — never opening the database directly to the public internet, which trades a minor testing inconvenience for a serious, ongoing security regression.

Choosing Managed vs. Self-Managed on a VM

Managed is the right default recommendation most of the time — less operational burden, at the cost of somewhat less control and sometimes fewer configuration options or supported engine versions. Self-managing a database on a plain VM (Chapter 3) makes sense specifically when a required configuration,

extension, or engine version genuinely isn't supported by the managed offering — a real, occasionally legitimate reason, not simply a preference for more control.

WHERE THE ACTUAL QUERY-WRITING KNOWLEDGE LIVES

This chapter deliberately stays at the operational/service layer. For the SQL and schema-design knowledge itself, this site's own `mysql2` / `mysql3` courses cover relational databases in depth, and `mongodb1` / `mongodb2` cover the document model — all of it applies directly on top of whatever this chapter's managed service is running underneath.

Hands-On Exercises

EXERCISE 1

An organization migrates a self-hosted MySQL database to a managed RDS/Cloud SQL/Azure SQL instance. Explain what changes and what stays the same, and specifically which existing SQL knowledge continues to apply directly, unchanged.

View solution

EXERCISE 2

Distinguish a read replica from a Multi-AZ/HA standby replica — what each is actually for, and whether one can also serve the other's purpose by default.

View solution

EXERCISE 3

A customer says: "I can't connect to my database from my laptop, so let's just make it publicly accessible temporarily to test." Explain why this is a bad idea, and what the correct troubleshooting/access approach should be instead.

View solution

CHAPTER 7 QUICK REFERENCE

- **Managed** = PaaS layer for the database — provider handles patching/backups/failover, customer still owns schema/queries/data
- **RDS/Azure SQL/Cloud SQL** run the same MySQL/PostgreSQL/SQL Server engines — all `mysql2` / `mysql3` SQL knowledge transfers unchanged
- **DynamoDB/Cosmos DB/Firestore** are not "managed MongoDB" — distinct proprietary APIs, though Cosmos DB offers a MongoDB-compatible mode

- **Read replicas** (scaling reads) ≠ **Multi-AZ/HA standby** (failover) — easy to confuse, distinct default purposes
- "Connection refused"/"too many connections" — usually pool exhaustion or a security group blocking the port, not a broken database
- Never expose a database publicly to work around a private-subnet connectivity issue — use a bastion host, VPN, or app-tier proxy instead
- **Next chapter:** Monitoring & Logging — CloudWatch/Azure Monitor/Cloud Monitoring, log aggregation, and alerting basics

CHAPTER 8 OF 12

Monitoring & Logging

Cloud Platform Fundamentals

Chapter 8 · Monitoring & Logging

With compute, storage, networking, IAM, and databases now covered, this chapter is about how you actually *see* what's happening across all of them. "Check the logs and metrics" is usually step one of any real investigation — this is genuinely foundational support-work material.

Metrics vs. Logs vs. Traces — Three Different Signals

Sometimes called the "three pillars of observability" — each answers a genuinely different question:

SIGNAL	WHAT IT IS	BEST SUITED TO ANSWER
Metrics	Numeric time-series data (CPU%, request count, latency)	"Is something wrong?" — trends and alerting
Logs	Discrete, timestamped event records	"What exactly happened?"
Traces	A single request's path across multiple services	"Where in a multi-service chain did it break?"

This chapter focuses mainly on metrics and logs — tracing is more specifically a distributed-systems/microservices topic, worth knowing exists but not covered in depth here.

The Terminology Map

- **AWS** — CloudWatch, with Metrics, Logs, and Alarms as its components.
- **Azure** — Azure Monitor, with Metrics, Log Analytics, and Alerts.
- **GCP** — Cloud Monitoring and Cloud Logging (formerly both branded "Stackdriver" — a rename still worth knowing, since older documentation and community content commonly still references the old name), plus Alerting policies.

What Gets Monitored Automatically vs. What You Have to Configure

Baseline infrastructure metrics — CPU, disk, network — are typically collected automatically for compute resources at a basic level. **Application-level or custom metrics** need to be explicitly instrumented and

pushed by your own code — they don't appear on their own.

> **"WHY ISN'T X SHOWING UP IN MONITORING?"**
>
> A genuinely common support scenario, and the answer is almost always "because nobody configured it to be collected," not a monitoring system failure. One specific, practical gotcha worth naming directly: memory metrics, on several providers' basic compute monitoring, often require an additional agent installed on the instance — they're not automatically included at the same baseline level as CPU.

Log Aggregation

Rather than SSHing into individual instances to read local log files one by one, cloud logging services centralize logs from VMs, containers, and managed services into one searchable place. Organizing structures differ by name — log groups/streams (AWS), workspaces (Azure), logging buckets (GCP) — but the underlying idea is identical. Logs aren't kept forever by default; **retention settings** are configurable and directly affect storage cost, echoing Chapter 4's lifecycle-policy material and feeding directly into Chapter 9's cost chapter.

Querying Logs

Each provider has its own log query language — CloudWatch Logs Insights, Azure's KQL (Kusto Query Language), and GCP's Cloud Logging query language. Unlike SQL's broad portability across relational databases (Chapter 7), these are genuinely different syntaxes — another place where cross-provider knowledge doesn't transfer directly, worth flagging honestly rather than glossing over. The single most useful, universally applicable query pattern regardless of syntax: filtering by time range plus a specific error pattern or status code — the backbone of most real support investigations.

Alerting Basics

An alert defines a threshold or condition on a metric or log pattern that triggers a notification — email, SMS, chat integration, or paging. The right things to alert on are **symptoms that actually matter to users** — error rate, latency — rather than every possible internal metric a system happens to expose. Dashboards provide the "at a glance" view, built from the same underlying metrics as alerts.

> **ALERT FATIGUE IS A REAL, SERIOUS OPERATIONAL RISK**
>
> Too many low-value, noisy alerts train people to start ignoring notifications generally — including the genuinely important ones buried among them. A monitoring setup that pages someone for every minor fluctuation isn't more thorough than one with carefully chosen thresholds; it's often measurably less safe, because the signal that actually matters gets lost in the noise.

A Support Workflow — Where to Look First

A practical two-step pattern: check the relevant **metrics** first for an obvious spike or drop around the reported time — a fast, high-level signal for narrowing down *when* and roughly *where* something went wrong. Then drill into **logs** for the specific detail once a rough time window and affected component are identified — logs explain *what* actually happened. Course 2's own `cloud2-4` ("Reading Logs & Metrics Under Pressure") builds directly on this exact two-step pattern.

Hands-On Exercises

EXERCISE 1

Explain the difference between metrics, logs, and traces, giving one example of a question each is specifically best suited to answer.

View solution

EXERCISE 2

A customer says: "Our CPU usage looks completely fine in monitoring, but the app was definitely slow." Give a plausible monitoring-gap explanation for this.

View solution

EXERCISE 3

Explain what alert fatigue is, and why setting too many low-value alerts can actually make a system less safe rather than more.

View solution

CHAPTER 8 QUICK REFERENCE

- **Metrics** (is something wrong?) · **Logs** (what happened?) · **Traces** (where in a multi-service chain?)
- CloudWatch (AWS) · Azure Monitor · Cloud Monitoring/Logging (GCP, formerly Stackdriver)
- Infrastructure metrics are usually automatic; application/custom metrics (and often memory) must be explicitly instrumented — "why isn't X showing up" is almost always a configuration gap, not a monitoring failure
- Log query languages (Logs Insights/KQL/Cloud Logging query language) genuinely differ by provider — unlike SQL, this knowledge doesn't transfer directly

- Alert on symptoms that matter to users (error rate, latency), not every internal metric — **alert fatigue** is a real risk that buries important signals in noise
- Practical workflow: metrics narrow down when/where, logs explain what — built on directly by `cloud2-4`
- **Next chapter:** Cost Management & Billing — pricing models, reserved/spot instances, cost anomalies, budgets and alerts

CHAPTER 9 OF 12

Cost Management & Billing

Cloud Platform Fundamentals

Chapter 9 · Cost Management & Billing

This chapter turns Chapter 8's "observe what's happening" lens specifically onto cost — a topic this course has been foreshadowing since Chapter 1's honest note about CapEx-to-OpEx trade-offs, and touching repeatedly since (Chapter 3's pricing models, Chapter 4's lifecycle policies).

Why Cloud Cost Management Is Its Own Discipline

On-prem hardware cost is mostly fixed and predictable once purchased. Cloud cost is variable, driven directly by **usage** — meaning it can grow (or shrink) dynamically, sometimes unexpectedly. Chapter 1's CapEx-to-OpEx trade brings real flexibility, but it also removes the natural spending cap a big upfront hardware purchase used to provide — which is exactly why cost needs active, ongoing management rather than a one-time budgeting exercise.

How Cloud Billing Actually Works

Billing is usage-based, at a granularity that varies by resource type — compute is often billed per-second or per-hour, storage per GB-month, data transfer per GB. Monthly invoices aggregate usage across every service used. Organizations with multiple sub-accounts or projects (echoing Chapter 2's resource hierarchy — AWS Organizations, Azure Management Groups, GCP Billing Accounts + Projects) typically use consolidated billing, giving one centralized invoice across many accounts rather than a separate bill per account.

Pricing Models, Revisited From Chapter 3

Chapter 3 covered on-demand, reserved, and spot pricing specifically for compute. Broadening slightly: AWS also offers **Savings Plans** — committing to a dollar-per-hour spend level rather than a specific instance type, trading some of reserved pricing's discount depth for meaningfully more flexibility. Azure and GCP offer similar concepts (Azure Reservations/Savings Plans, GCP Committed Use Discounts). The general principle holds across all of them: the more you commit in advance — in time, and in specificity — the deeper the discount, and the less flexibility you retain.

Where Costs Actually Come From — The Usual Suspects

- **Idle or oversized compute** — instances left running unnecessarily, or sized larger than the workload actually needs (Chapter 3).
- **Unattached/orphaned storage volumes** — a VM was terminated, but its disk wasn't deleted, and keeps billing indefinitely (Chapter 4).
- **Data transfer/egress costs** — genuinely often underestimated; moving data *out* of a provider, or between regions, is frequently the surprising line item on a bill — much more so than inbound transfer, which is usually free or cheap.
- **Unused load balancers or NAT gateways** left provisioned after they're no longer needed.
- **Forgotten dev/test environments** — never torn down after a project ended, quietly accumulating cost for months.

Cost Anomalies & How to Investigate Them

A cost anomaly is an unexpected, unexplained spend spike. Each provider offers native anomaly-detection tooling (AWS Cost Anomaly Detection, Azure Cost Management anomaly alerts, GCP's budget-based alerting). Investigation almost always starts by breaking spend down — by service, by resource, by **tag/label**.

> **TAGGING IS THE SINGLE MOST VALUABLE HABIT FOR MAKING COST INVESTIGATION POSSIBLE**
>
> Properly tagging resources by team, project, or purpose (echoing Chapter 6's resource-organization ideas) is what makes it possible to attribute a cost spike to a specific owner at all. An untagged resource showing up in a cost spike is genuinely hard to investigate — nobody can easily tell whose workload it belongs to, or whether it's still needed.

Budgets & Cost Alerts

Rather than discovering a runaway cost issue on next month's invoice, a budget threshold can trigger a notification — or, in stricter configurations, actually restrict further provisioning — as spend approaches or exceeds a defined limit. This is the cost equivalent of Chapter 8's alerting material: the same underlying pattern (a threshold plus a notification), applied to a different signal.

A Support-Relevant Billing Scenario

"Why is my bill higher this month?" — a practical investigation order: check the usual suspects list above first, then look at the cost breakdown by service/tag for the actual delta, then check for any pricing changes or newly provisioned resources around the time the increase began. Two specific gotchas this course has already

covered are common concrete answers here: Chapter 3's stopped-VM-still-billing-for-storage scenario, and Chapter 4's archive-retrieval-cost surprise.

DATA EGRESS COSTS DESERVE SPECIAL ATTENTION IN MULTI-CLOUD/HYBRID DESIGNS

Chapter 1 discussed hybrid and multi-cloud architectures as legitimate choices for real reasons. Worth flagging explicitly here: moving large amounts of data between providers, or out to the internet, can be far more expensive than expected — a real, easily underestimated factor in any architecture spanning more than one cloud environment.

Hands-On Exercises

EXERCISE 1

A customer's bill is unexpectedly high this month, and they report no obvious change in their own usage. List the "usual suspects" to check first, and explain the reasoning for checking them in that order.

View solution

EXERCISE 2

Explain why untagged resources make cost anomaly investigation significantly harder, and what proactive step prevents this problem.

View solution

EXERCISE 3

Explain the general principle behind cloud pricing discounts (on-demand vs. reserved/committed vs. spot) in terms of what you're actually trading away in exchange for a lower price.

View solution

CHAPTER 9 QUICK REFERENCE

- Cloud cost is usage-driven and variable, not a fixed upfront capital expense — needs active, ongoing management
- Billing granularity varies by resource; consolidated billing centralizes invoices across sub-accounts/projects
- The general discount trade-off: more commitment (time/specificity) = deeper discount, less flexibility
- Usual suspects: idle/oversized compute, orphaned storage volumes, data egress, unused load balancers/NAT gateways, forgotten dev/test environments
- **Tagging** is the single most valuable practice for making cost anomaly investigation tractable at all

- Budgets/cost alerts are Chapter 8's alerting pattern (threshold + notification) applied to spend
- Data egress costs are an easily underestimated, real factor in multi-cloud/hybrid architectures
- **Next chapter:** Security & Compliance Basics — encryption at rest/in transit, security groups/NSGs revisited, and a compliance-framework overview

CHAPTER 10 OF 12

Security & Compliance Basics

Cloud Platform Fundamentals

Chapter 10 · Security & Compliance Basics

Security Is a Thread, Not a Single Chapter

Real security material has already run through this course from the start — Chapter 1's shared responsibility model, Chapter 5's security groups and NACLs, Chapter 6's IAM/least privilege/MFA. This chapter pulls together what's left: encryption specifically, and a brief compliance overview — connecting directly into this site's own dedicated `crypto1`, `dbsec1`, and `owasp1` security courses, rather than re-covering ground already handled.

Encryption at Rest

Directly building on `crypto1-5` / `crypto1-6` and `dbsec1-5`: data encrypted while stored on disk. Cloud providers typically offer this as a simple default-on option for storage services (Chapter 4) and managed databases (Chapter 7) — the provider handles key generation and management by default (a **provider-managed key**), or a **customer-managed key** can be used instead, via a Key Management Service, for more control at the cost of more operational responsibility.

ENCRYPTION AT REST DOESN'T REPLACE ACCESS CONTROL

Restating a point directly from `crypto1-6`: encryption at rest protects against a physically stolen disk — it does **not** protect against a compromised application or IAM credential that already has legitimate access to decrypt and read the data. Encryption is not a substitute for the access control covered in Chapter 6; they're two separate layers of defense, both needed.

Encryption in Transit

Directly building on `https1` / `crypto1` and `dbsec1-6`: TLS protecting data moving over the network, both between users and services and between services themselves. Most managed services (Chapter 7) support or enforce TLS connections by default today. A genuinely common misconfiguration worth flagging explicitly: an application connecting to a managed database *without* actually enforcing TLS, even though the option is available — "available" and "enforced" are different things, worth checking directly rather than assuming.

Key Management Services (KMS)

Expanding the customer-managed-key point above: AWS KMS, Azure Key Vault, and GCP Cloud KMS centralize encryption key generation, storage, and rotation — directly applying `crypto1-11`'s key management material to a specific product category. One genuinely important nuance: *who can use a key* to encrypt or decrypt is itself governed by IAM policies (Chapter 6) — key access and data access are two **separate** permission layers, and both need to be correctly configured for encryption to actually mean anything as a control.

Security Groups, NSGs & Firewalls, Revisited

A brief callback to Chapter 5 rather than re-teaching it: the network-layer security control. This chapter adds one thing Chapter 5 didn't cover — a **Web Application Firewall (WAF)**, a Layer 7 control sitting in front of a web application, filtering malicious HTTP requests. This connects directly to this site's `xss1` / `sqli1` courses' own attack categories: a WAF is a genuine defense-in-depth layer against exactly those attack types, **not** a replacement for actually fixing the underlying application vulnerability — echoing `owasp1`'s own recurring theme that no single defense replaces properly handling the root cause.

Compliance Frameworks — A Brief Overview

Not a deep dive — just orientation to frameworks a support engineer is likely to hear referenced:

FRAMEWORK	SCOPE
SOC 2	A common enterprise vendor-trust standard
PCI DSS	Payment card data
HIPAA	US health data
GDPR	EU data protection

The general pattern across all of them: specific technical and procedural requirements for how sensitive data is handled, stored, and audited. Providers offer compliance certifications for their *own* infrastructure — the "of the cloud" side of Chapter 1's shared responsibility model — but using a compliant provider does **not** automatically make a customer's own application or configuration compliant. The "in the cloud" half of that responsibility remains entirely the customer's.

A Support-Relevant Distinction — "Is This Compliant?" Isn't a Simple Yes/No

Tying shared responsibility and compliance together directly: a customer asking "is your platform HIPAA compliant" often really means "is *my* specific configuration compliant" — which depends on things only they control (IAM policies, encryption settings, data handling practices), not just which provider they're using. Being able to explain this clearly, rather than giving a false blanket "yes" or "no," is a genuinely useful support skill.

PROVIDER COMPLIANCE CERTIFICATION ≠ AUTOMATIC CUSTOMER COMPLIANCE

A provider's SOC 2/HIPAA/PCI DSS certification covers their own infrastructure and operational practices — it says nothing about whether a specific customer's own IAM policies, encryption settings, or application code meet that same standard. The "in the cloud" half of the shared responsibility model (Chapter 1) is still entirely the customer's to get right.

Hands-On Exercises

EXERCISE 1

Explain what encryption at rest protects against, and what it does *not* protect against — specifically contrasting a stolen physical disk with a compromised credential that already has legitimate decrypt access.

View solution

EXERCISE 2

Explain why "our cloud provider is SOC 2 / HIPAA / PCI DSS certified" doesn't automatically mean a customer's own application built on that provider is compliant.

View solution

EXERCISE 3

A team has already fixed a SQL injection vulnerability directly in their application code. Explain why deploying a WAF in front of the application is still worth doing, even after the "real" fix is already in place.

View solution

CHAPTER 10 QUICK REFERENCE

- This chapter fills the gaps left by earlier security-adjacent chapters (Ch.1, Ch.5, Ch.6), not a full security course — see `crypto1` / `dbsec1` / `owasp1` for depth
- **Encryption at rest** — provider-managed or customer-managed (KMS) keys; protects against stolen disks, not against a compromised credential with legitimate access

- **Encryption in transit** — TLS; "available" and "enforced" are different things, worth verifying explicitly
- **KMS** — key access (IAM) and data access are two separate permission layers, both must be correctly configured
- **WAF** — a Layer 7 defense-in-depth layer against attacks like XSS/SQLi, never a substitute for fixing the underlying vulnerability
- **Compliance frameworks** (SOC 2/PCI DSS/HIPAA/GDPR) — provider certification covers only the "of the cloud" half; the customer's own configuration is still entirely their responsibility
- **Next chapter:** Infrastructure as Code — A First Look — CloudFormation/ARM-Bicep/Terraform conceptually

CHAPTER 11 OF 12

Infrastructure as Code — A First Look

Cloud Platform Fundamentals

Chapter 11 · Infrastructure as Code — A First Look

Every chapter so far has covered individual services. This chapter steps back to look at *how* those resources actually get created and managed in practice, at real scale — deliberately a first look, not a deep dive: this site's own bucket list has a separate, dedicated Terraform course planned for genuine hands-on depth.

The Problem With Manual (Console-Driven) Provisioning

Clicking through a web console to create resources is genuinely fine for learning and experimentation — Chapter 2's own free-tier advice assumed exactly this. It doesn't scale to real production use, though: there's no record of exactly what was created or why, it's hard to reproduce identically (dev, staging, and production environments quietly drift apart over time — **configuration drift**), there's no history or audit trail of changes, it's error-prone for repetitive tasks, and it simply doesn't scale to hundreds or thousands of resources.

What Infrastructure as Code Actually Means

Rather than a sequence of manual click-steps, infrastructure is described in config files defining the **desired state** — the config file itself becomes the single source of truth for what should exist. Because it's just text, it can be version-controlled exactly like application code — this site's own `git1`-`git3` courses apply directly, and `git2-7`'s code review process now applies equally to infrastructure changes, not just application code.

Declarative vs. Imperative

Most IaC tools are **declarative** — you describe the desired end state ("I want 3 web servers and 1 load balancer"), and the tool figures out what changes are needed to get there. This contrasts with an **imperative** approach — a sequence of manual steps ("create server 1, create server 2, create server 3...") that doesn't inherently know how to safely reconcile drift or a partial failure the way a declarative tool's plan/apply cycle does. Terraform, CloudFormation, and ARM/Bicep are all declarative.

The Big Three's Native Tools

PROVIDER	NATIVE TOOL
AWS	CloudFormation (JSON/YAML templates)
Azure	ARM templates, or Bicep — a newer, cleaner syntax that compiles down to ARM JSON
GCP	Deployment Manager (older); increasingly config-driven approaches or Terraform rather than one dominant native tool

Terraform — The Cross-Provider Option

HashiCorp's Terraform works across all three major providers (and many others) using one consistent language (HCL) and workflow, rather than three separate provider-specific tools — directly echoing this course's own cross-provider framing since Chapters 1 and 2. This is exactly why Terraform is such a common real-world choice for organizations working across multiple clouds.

> **THIS CHAPTER IS INTENTIONALLY INTRODUCTORY**
> This chapter exists so IaC concepts and terminology are recognizable, not to teach writing Terraform configs — that's the scope of the site's own separate, dedicated Terraform course, planned for real hands-on depth.

The Plan/Apply Workflow

The standard declarative IaC cycle:

```
1. WRITE/UPDATE CONFIG   -- describe the desired infrastructure state
2. PLAN                  -- shows exactly what would change, without changing anything yet
3. REVIEW THE PLAN        -- a genuinely valuable safety check before anything happens
4. APPLY                 -- actually makes the changes
```

To compute that difference between current and desired state, the tool needs to track what it currently believes exists — a **state file**, in Terraform's case.

> **A REAL GOTCHA: MANUAL CHANGES BREAK THE TOOL'S PICTURE OF REALITY**
> If infrastructure is changed manually, outside the IaC tool — a well-intentioned "quick console fix" during an incident — the tool's tracked state no longer matches what's actually deployed. This is exactly how configuration drift creeps back in even in an organization that genuinely uses IaC, and it's a common source of confusing, unexpected plan output later, when the tool tries to reconcile a reality it no longer accurately understands.

Why This Matters for Support Work

Even without writing IaC configs day to day, knowing that infrastructure *might* be managed this way genuinely changes the right troubleshooting approach: a manual emergency fix applied through the console can get

silently overwritten — or conflict outright — the next time the IaC tool runs. The right emergency practice is usually to make the fix *and* update the IaC config to match it, not just the console alone.

Hands-On Exercises

EXERCISE 1

Explain the difference between declarative and imperative approaches to provisioning infrastructure, giving one example of each.

View solution

EXERCISE 2

Explain what configuration drift is, and describe how it can still happen even in an organization that genuinely uses IaC.

View solution

EXERCISE 3

Explain why a support engineer should think twice before making a manual "quick fix" directly in the console on infrastructure that's managed by IaC, and what the better alternative practice is.

View solution

CHAPTER 11 QUICK REFERENCE

- Manual console provisioning doesn't scale — no record, no reproducibility, no audit trail, drift-prone
- **IaC** — config files defining desired state, version-controllable exactly like application code (`git1` - `git3`)
- **Declarative** (describe end state, tool figures out the diff) vs. **imperative** (manual step sequence)
- **CloudFormation** (AWS) · **ARM/Bicep** (Azure) · **Deployment Manager/Terraform** (GCP) · **Terraform** as the cross-provider option
- **Plan/apply workflow** — plan shows changes before they happen; a state file tracks what the tool believes currently exists
- A manual "quick fix" outside the IaC tool causes drift and can be silently overwritten on the next apply — fix the config too, not just the console
- **Next chapter:** Choosing & Comparing Providers, and Where to Go Next — a decision framework, bridging into Course 2

CHAPTER 12 OF 12

Choosing & Comparing Providers, and Where to Go Next

Cloud Platform Fundamentals

Chapter 12 · Choosing & Comparing Providers, and Where to Go Next

The closing chapter of Course 1 — bringing everything from Chapters 1-11 together into a practical provider decision framework, then bridging into Course 2's dedicated support-engineer focus.

There's Rarely a Single "Best" Provider

Revisiting Chapter 2's own honest framing directly: market share isn't the whole story. The right choice depends on a specific organization's actual context, not a universal ranking — which is exactly why this course has stayed cross-provider throughout, rather than picking a "winner" to specialize in.

A Practical Decision Framework

- **Existing ecosystem investment** — an organization heavily invested in Microsoft 365/Active Directory has a real, concrete integration advantage available on Azure (Chapter 2).
- **The team's existing expertise** — retraining cost is real; the technically "best" option sometimes loses fairly to "what the team already knows how to operate well."
- **Specific service needs** — does the workload need something one provider does distinctly better, like GCP's particular strength in data/ML tooling (Chapter 2)?
- **Compliance and data residency requirements** (Chapter 10) — specific regions or certifications the workload genuinely requires.
- **Existing multi-cloud/hybrid reality** (Chapter 1) — sometimes the "choice" is already made by what's already in place, and the real question is how to work well within that constraint rather than an idealized greenfield decision.

When Multi-Cloud Is the Answer Rather Than a Question

Revisiting Chapter 1's hybrid/multi-cloud material: organizations often end up on more than one provider not through a single deliberate strategic choice, but through acquisitions, different teams making different decisions over time, or deliberate risk-diversification. Worth stating honestly: multi-cloud adds real operational

complexity — Chapter 2's terminology-mapping friction, Chapter 8's per-provider log query language differences — a genuine trade-off, not a free win.

This Course's Own Throughline, Restated

A pattern has run through nearly every chapter of this course, worth naming explicitly now that it's complete:

CHAPTER	"LOOKS BROKEN" SCENARIO	ACTUALLY...
1	A leaked public storage bucket, "the cloud is insecure"	A customer-side shared-responsibility misconfiguration
3	"I'm still being billed, my VM is stopped"	Stopped ≠ terminated — storage keeps existing and billing
5	"I can't reach my database from outside"	The private-subnet pattern working exactly as designed
7	Same database-reachability complaint	The correct, deliberate fix is a bastion/VPN, not going public
11	A confusing IaC plan showing unexpected changes	Drift from an earlier manual console fix, not a bug

Five separate chapters, five separate services — the same underlying pattern every time: something that *looks* like a failure is, on closer investigation, a deliberate design choice or an expected consequence of how the system actually works. This is this course's own version of what other courses on this site have identified as their own recurring throughlines — recognizing it is arguably the single most transferable skill this course teaches.

> **THE INSTINCT WORTH CARRYING FORWARD**
> Before assuming something is broken, ask what it would mean for this behavior to be intentional. Across this entire course, that question turned out to be the right one to ask far more often than not.

Bridging Into Course 2

Course 1 built the conceptual foundation across every major service category. Course 2 (`cloud2`) takes the troubleshooting instincts developed along the way — Chapter 1's shared-responsibility triage, Chapter 5's connectivity checklist, Chapter 6's AuthN/AuthZ triage, Chapter 7's connection-management patterns, Chapter 8's metrics-then-logs workflow, Chapter 9's cost-investigation approach — and turns them into a dedicated, deep, support-engineer-focused course:

CHAPTER	TOPIC
1	The Support Engineer's Cloud Toolkit

CHAPTER	TOPIC
2	Diagnosing Connectivity Issues
3	IAM & Permission Troubleshooting
4	Reading Logs & Metrics Under Pressure
5	Common Failure Modes & Root Cause Analysis
6	Incident Response in the Cloud
7	Cost Anomalies & Billing Support
8	Working With Cloud Provider Support
9	Multi-Cloud & Hybrid Environments
10	Capstone: Diagnosing a Real Multi-Service Outage

Hands-On Exercises

EXERCISE 1

A small team, already deeply invested in Microsoft 365 and Active Directory, is building a new internal tool. Using this chapter's decision framework, recommend a provider and justify the recommendation.

View solution

EXERCISE 2

Explain why "multi-cloud" is sometimes not a deliberate strategic choice at all, and describe the real operational cost this course has already covered that comes with it regardless of how it happened.

View solution

EXERCISE 3

List at least three "looks like an outage/bug but is actually working as designed" examples from across this course, and explain what they have in common.

View solution

CHAPTER 12 QUICK REFERENCE

- No single "best" provider — decisions depend on ecosystem investment, team expertise, specific service strengths, compliance needs, and existing multi-cloud reality
- Multi-cloud is often not a deliberate choice — and it carries real, unavoidable operational complexity regardless of how it happened
- This course's throughline: something that looks broken is very often working exactly as designed — recognizing that pattern is the most transferable skill here
- **Course 1 complete** — Cloud Platform Fundamentals, 12 chapters, from "what is the cloud" to a provider decision framework
- **Course 2 next:** Cloud Troubleshooting & Support — turning this course's scattered troubleshooting instincts into a dedicated, deep support-engineer course