CLOUD & DEVOPS

AWS Fundamentals

AWS's real ecosystem and shared responsibility model, IAM, EC2, S3, VPC networking, RDS & DynamoDB, Lambda and serverless computing, load balancing and auto scaling, CloudWatch and cost management, and a closing capstone deploying a real, working web application.

10 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: WHY AWS? ECOSYSTEM, REGIONS & THE SHARED RESPONSIBILITY MODEL

AWS Fundamentals

Chapter 1 · Why AWS? Ecosystem, Regions & the Shared Responsibility Model

Cloud Platform Fundamentals (`cloud1`) covered cloud computing generically — IaaS/PaaS/SaaS, elasticity, the shared-responsibility idea in the abstract, without committing to any one provider's own real services or console. This course goes deep on one specific provider instead: Amazon Web Services, in genuine, hands-on depth — real services, real console/CLI workflows, real architecture decisions.

AWS's Real Origin

Amazon began offering web services as early as July 2002, but the real cloud-computing platform known today as AWS launched in March 2006. Its first real, generally available service was **S3** (Simple Storage Service, launched 14 March 2006) — object storage, not compute. **EC2**, the virtual-machine service most people associate with "the cloud," followed as a limited public beta on 25 August 2006, and didn't leave beta until October 2008. AWS's own real infrastructure, in other words, grew out of Amazon's own internal need to run its retail business at scale — Amazon itself started running its own retail platform on EC2 in November 2010, once the service had matured enough for that internal use.

Regions, Availability Zones & Edge Locations

AWS's own global infrastructure is organized in three real, distinct layers:

- **Region** — a real, distinct geographical area (e.g. `us-east-1` , `eu-west-2`), wholly contained within a single country, where your data stays unless you explicitly configure otherwise.
- **Availability Zone (AZ)** — each Region contains multiple AZs, and each AZ is one or more real, physically discrete data centers with their own independent power, networking, and connectivity — deliberately isolated from each other so a failure in one AZ doesn't take down another.
- **Edge Location** — a much larger number of smaller, real caching/content-delivery points (used by CloudFront and other services) positioned close to end users, distinct from a full Region or AZ.

As of a real, documented count from October 2025, AWS operates 38 geographical Regions containing 120 Availability Zones between them, plus over 700 real Edge Locations worldwide — a genuinely global footprint most single applications never come close to needing in full.

WHY THIS STRUCTURE MATTERS IMMEDIATELY

Choosing a Region isn't just about latency to your users — it also determines which specific AWS services are available (not every service ships to every Region on day one) and, for regulated data, which country's own legal jurisdiction your data sits under. Deploying across multiple AZs within one Region, not multiple Regions, is the real, standard way to survive a single data-center failure without meaningfully increasing latency.

The Shared Responsibility Model

AWS draws a real, explicit line between what it secures and what you, the customer, are responsible for securing — commonly summarized as security **of** the cloud versus security **in** the cloud.

AWS'S RESPONSIBILITY ("OF" THE CLOUD)	YOUR RESPONSIBILITY ("IN" THE CLOUD)
Physical data-center security	Your own IAM users, roles & permissions (Chapter 2)
The underlying hardware, networking, and virtualization layer	Your own operating-system patches, on any EC2 instance you run
The managed-service software itself (e.g. RDS's own database engine patching)	Your own data — encryption, access control, backups
Global infrastructure availability (Regions, AZs)	Your own network configuration — security groups, VPC design (Chapter 5)

This split shifts depending on the specific service: for EC2 (an unmanaged VM), you own the entire guest operating system and everything on it; for a fully managed service like Lambda, AWS takes on far more, and you're mostly responsible only for your own code and its permissions.

A REAL, COMMON MISUNDERSTANDING

"AWS is secure" does not mean "my application on AWS is automatically secure." An S3 bucket accidentally left publicly readable, or an EC2 instance with an overly permissive security group, is a

real, well-documented, and entirely real-world common cause of data breaches — and squarely the customer's own side of this model, not AWS's.

AWS's Real Market Position

Per real, documented Synergy Research Group data (Q1 2023), AWS held roughly 31% of the global cloud infrastructure market, ahead of Microsoft Azure's real 25% and Google Cloud's real 11% — the same three providers this bucket of courses covers, in the same real relative order they hold in the market today.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why deploying an application across multiple Availability Zones within a single Region is usually a better first step for high availability than deploying across two entirely separate Regions.

View solution

EXERCISE 2

A company deploys an EC2 instance, leaves its operating system unpatched for a year, and is later compromised through a known OS vulnerability. Using the shared responsibility model, explain whose real responsibility this failure was, and why.

View solution

EXERCISE 3

Explain, in your own words, why the shared responsibility split for a fully managed service like Lambda is genuinely different from the split for an unmanaged EC2 instance, and give one real, concrete example of a responsibility that shifts from you to AWS between the two.

View solution

CHAPTER 1 QUICK REFERENCE

- **AWS launched** — real cloud platform from March 2006, S3 first, EC2 beta later that year
- **Region** — a distinct geographical area; **Availability Zone** — an isolated data center (or group of them) within a Region; **Edge Location** — a smaller, latency-focused caching point
- **Real current scale** — 38 Regions, 120 Availability Zones, 700+ Edge Locations (as of October 2025)
- **Shared responsibility model** — AWS secures "of" the cloud (infrastructure); you secure "in" the cloud (your own data, access, configuration)
- **Real market position** — AWS ~31%, Azure ~25%, GCP ~11% (Synergy Research, Q1 2023)

CHAPTER 2: IAM: IDENTITY & ACCESS MANAGEMENT

AWS Fundamentals

Chapter 2 · IAM: Identity & Access Management

Chapter 1 named "your own IAM users, roles & permissions" as squarely your side of the shared responsibility model. This chapter covers exactly that — **IAM** (Identity and Access Management), the real AWS service controlling who, and what, can do what, to which resources, across your entire account.

The Root Account — and Why You Don't Use It Day to Day

Every AWS account has exactly one real **root user**, created when the account itself was created, with unlimited, unrestricted access to everything — including the ability to close the account or change its own billing. AWS's own real, standard guidance is to lock this account down immediately (a strong password, MFA enabled) and then **never use it for everyday work**. Every real task after that first setup should run through a separate, individually named IAM identity instead — so that if any single credential is ever compromised, the damage is bounded to whatever that one identity was actually permitted to do.

Users, Groups, Roles & Policies

- **IAM User** — a real, individual identity (a person or an application) with its own long-term credentials (a password for console access, and/or access keys for programmatic access).
- **IAM Group** — a real, named collection of users. You attach permissions to the group once, and every user in it inherits them — the standard way to manage permissions for a whole team rather than one user at a time.
- **IAM Role** — genuinely different from a user: a role has *no* long-term credentials of its own. It's assumed temporarily — by a real person, by an AWS service (an EC2 instance, a Lambda function), or by another AWS account — and grants only short-lived, automatically expiring credentials for the duration it's assumed.
- **IAM Policy** — a real JSON document defining exactly what's allowed or denied. Policies attach to users, groups, or roles, and are what actually grant (or restrict) permission — a

user or role with no policies attached can do essentially nothing.

ROLES ARE THE REAL DEFAULT FOR AWS SERVICES

Never embed long-term access keys directly inside application code running on an EC2 instance or in a Lambda function — the real, standard practice is to attach an IAM role to that resource instead. AWS automatically issues, and automatically rotates, temporary credentials to anything assuming that role, with nothing to leak in a code repository.

A Real Policy Document

```
// Allows read-only access to objects in one specific S3 bucket only
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": ["s3:GetObject", "s3:ListBucket"],
      "Resource": [
        "arn:aws:s3:::example-reports-bucket",
        "arn:aws:s3:::example-reports-bucket/*"
      ]
    }
  ]
}
```

Every real IAM policy statement has the same four real building blocks: an **Effect** (`Allow` or `Deny`), an **Action** (the specific API calls it applies to, like `s3:GetObject`), a **Resource** (which specific thing it applies to, by ARN), and optionally a **Condition** (narrowing it further — e.g. only from a specific IP range, or only with MFA present).

The Principle of Least Privilege

A policy should grant exactly the permissions a real task needs, and nothing more. This is not just a security nicety — it's the direct practical guard against exactly the kind of failure Chapter

1's own shared-responsibility warning described: a resource or credential with far broader permissions than it actually needs becomes a far bigger real liability if it's ever compromised.

Multi-Factor Authentication (MFA)

MFA requires a second, real proof of identity beyond a password — a time-based one-time code from an authenticator app, or a physical hardware key — before a sign-in (or, via a policy Condition, before a specific sensitive action) is allowed to proceed. AWS's own real, standard recommendation is MFA on the root account without exception, and strongly encouraged for every IAM user with console access.

A Real Case Study: The 2019 Capital One Breach

WHAT ACTUALLY HAPPENED

In 2019, a former AWS employee, Paige Thompson, exploited a misconfiguration in a web application firewall Capital One had deployed on AWS, gaining access to data for roughly 106 million people across the US and Canada — including around 140,000 Social Security numbers and 80,000 bank account numbers. Thompson was later convicted and sentenced. Amazon's own real, public statement on the incident was direct: the breach came from "a misconfiguration of the (Capital One-designed) web application and not the underlying (Amazon-designed) cloud-based infrastructure."

This is a real, documented, large-scale illustration of Chapter 1's own shared responsibility model in practice: AWS's own infrastructure was not the point of failure — a customer-side misconfiguration was. Getting IAM roles, policies, and least-privilege access right isn't an abstract best practice; it's the exact category of control that determines how much real damage a single misconfiguration like this one can actually do.

Users, Groups, Roles & Policies at a Glance

CONCEPT	HAS LONG-TERM CREDENTIALS?	TYPICAL USE
User	Yes	A specific real person or application needing standing access

CONCEPT	HAS LONG-TERM CREDENTIALS?	TYPICAL USE
Group	No (users inside it have their own)	Managing permissions for a whole team at once
Role	No — temporary credentials only	AWS services, cross-account access, or a person needing short-lived elevated access
Policy	N/A — not an identity	Defines what any of the above is actually allowed to do

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why AWS recommends never using the root account for day-to-day work, even though it has full permissions and would technically be able to complete any task.

View solution

EXERCISE 2

A developer hardcodes a long-term IAM access key directly into application code running on an EC2 instance, and later accidentally commits that code to a public GitHub repository. Explain what the real, standard alternative would have been, and why it avoids this specific risk entirely.

View solution

EXERCISE 3

Using the real 2019 Capital One breach as your example, explain, in your own words, how the principle of least privilege — if it had been fully applied to the compromised web application's own permissions — could have limited the real scale of the damage, even if the underlying web application firewall misconfiguration still occurred.

View solution

CHAPTER 2 QUICK REFERENCE

- **Root account** — unlimited access; secure it with MFA, then stop using it day to day
- **User** — a real identity with long-term credentials; **Group** — a named collection of users sharing permissions
- **Role** — assumed temporarily, no long-term credentials; the real default for AWS services and applications
- **Policy** — a JSON document (Effect/Action/Resource/Condition) that actually grants or restricts permission
- **Least privilege** — grant only what's needed, nothing more; directly limits the real damage of a single compromise
- **Real case study** — the 2019 Capital One breach (106 million people affected), a customer-side misconfiguration, not an AWS infrastructure failure

CHAPTER 3: EC2: VIRTUAL MACHINES IN THE CLOUD

AWS Fundamentals

Chapter 3 · EC2: Virtual Machines in the Cloud

Chapter 1 named EC2 as AWS's own real, original compute service — public beta in August 2006, full production by October 2008. Chapter 2 covered who's allowed to do what. This chapter covers the thing itself: launching, configuring, and securing a real virtual machine on AWS.

Instances & AMIs

An **instance** is a running virtual machine. Every instance boots from an **AMI** (Amazon Machine Image) — a real template bundling an operating system and, optionally, pre-installed software. AWS provides official AMIs (Amazon Linux, Ubuntu, Windows Server), the AWS Marketplace offers third-party ones, and you can create your own custom AMI from a configured instance to launch identical copies of it later.

Instance Types & Families

Real AWS instance types are named by family and generation (e.g. `t3.micro`, `m5.large`), each family tuned for a different real workload shape:

FAMILY	REAL EXAMPLES	TUNED FOR
General Purpose	T3, T2, M5, M4	Balanced CPU/memory — web servers, small databases, dev environments
Compute Optimized	C5, C4	CPU-heavy workloads — batch processing, media transcoding, high-traffic web servers
Memory Optimized	R5, R4, X1	Large in-memory datasets — big relational databases, in-memory caches
Storage Optimized	I3, D2, H1	High-throughput local storage — data warehousing, distributed file systems

FAMILY	REAL EXAMPLES	TUNED FOR
Accelerated Computing	P3, G3, F1	GPU/FPGA workloads — machine learning training, rendering

Real Pricing Models

- **On-Demand** — pay by the hour (or second, for many instance types), with no commitment; the most expensive per-hour rate, the most flexible.
- **Reserved** — commit to a specific instance type for 1 or 3 years, in exchange for a real, documented discount of roughly 35-75% off the On-Demand rate.
- **Spot** — bid on AWS's own real spare compute capacity, at discounts up to roughly 90% — but AWS can reclaim a Spot instance with only a short warning if it needs that capacity back, so there's no uptime guarantee.
- **Savings Plans** (introduced November 2019) — commit to a real dollar-per-hour spend rather than a specific instance type, trading some of Reserved pricing's own rigidity for flexibility across instance families.

> **MATCHING PRICING MODEL TO WORKLOAD**
>
> Spot is a genuinely good real fit for interruptible, fault-tolerant work (batch jobs, CI pipelines, stateless workers behind a load balancer) — never for a database's own primary instance or anything that can't tolerate an abrupt shutdown. Reserved/Savings Plans fit steady, predictable baseline load; On-Demand fits genuinely unpredictable or short-lived workloads.

Security Groups

A **security group** is a real virtual firewall attached to an instance's own network interface, controlling inbound and outbound traffic by port, protocol, and source/destination. Security groups are **stateful** — if inbound traffic on a given connection is allowed in, the matching outbound reply is automatically allowed back out, without needing a separate matching rule. (Chapter 5 covers the network-level, *stateless* alternative — Network ACLs — and exactly how the two differ.)

A REAL, COMMON MISTAKE

Opening SSH (port 22) or a database port to `0.0.0.0/0` — every IP address on the internet — "just to make it work" is a genuinely common, real misconfiguration, and exactly the kind of overly permissive setup Chapter 1's own shared-responsibility warning and Chapter 2's own least-privilege principle both point directly at. Scope security group rules to specific, known IP ranges wherever possible.

SSH Key Pairs

Rather than a password, Linux instances are normally accessed via an SSH key pair: AWS holds the public key, you keep the private key, and only someone holding the matching private key can authenticate. The private key is generated (or provided) once, at instance launch — AWS never stores a copy of it, so losing it means losing that specific access method entirely, with no real recovery path other than provisioning a new key.

A Real Security Deep-Dive: The Instance Metadata Service

Every running EC2 instance can query a real, special local address — `169.254.169.254` — for its own metadata, including, when an IAM role is attached (Chapter 2), that role's own temporary credentials. This service is called **IMDS**, and its own real history is a genuine security case study in its own right.

IMDSV1'S REAL WEAKNESS

The original version, IMDSv1, answered a simple, unauthenticated request with no session or token involved. AWS's own real, documented security guidance identifies several real attack vectors this made possible — including exactly the same category of vulnerability behind Chapter 2's own Capital One case study: a misconfigured web application firewall or reverse proxy, exploited via server-side request forgery (SSRF), can be tricked into making a request to that local metadata address on the application's own behalf — handing an attacker the instance's own IAM role credentials without ever needing to compromise the instance directly.

AWS's real fix, **IMDSv2**, requires a session to be explicitly established first via a PUT request, returning a secret token (valid up to 6 hours) that every subsequent metadata request must include as a header. Because a typical SSRF exploit can only manipulate an existing request's

headers or URL — not initiate this specific, separate PUT request from the application itself — this two-step design is real, structurally more effective than requiring only a static header, and is now AWS's own recommended default.

Hands-On Exercises

EXERCISE 1

A team runs a short-lived, fault-tolerant batch-processing job that can be safely restarted from where it left off if interrupted. Recommend a real pricing model for this workload, and explain why it fits better than the alternatives.

View solution

EXERCISE 2

Explain, in your own words, why security groups being stateful means you don't need a separate outbound rule to allow a reply to traffic your own inbound rule already permitted in.

View solution

EXERCISE 3

Explain, in your own words, why IMDSv2's requirement of a separate PUT request before any metadata can be read makes it genuinely harder for a typical SSRF vulnerability to steal an instance's own IAM role credentials, compared to IMDSv1.

View solution

CHAPTER 3 QUICK REFERENCE

- **Instance** — a running VM, booted from an **AMI** (its own OS + software template)
- **Instance families** — General Purpose (T/M), Compute Optimized (C), Memory Optimized (R/X), Storage Optimized (I/D), Accelerated Computing (P/G)

- **Pricing** — On-Demand (flexible), Reserved (35-75% off, committed), Spot (up to 90% off, interruptible), Savings Plans (flexible commitment)
- **Security groups** — stateful virtual firewalls; never open a port to 0.0.0.0/0 without a real reason
- **SSH key pairs** — public key held by AWS, private key held by you; no real recovery if lost
- **IMDSv2** — a real, token-based fix for IMDSv1's own SSRF-exploitable weakness, now AWS's recommended default

CHAPTER 4: S3: OBJECT STORAGE

AWS Fundamentals

Chapter 4 · S3: Object Storage

Chapter 1 named S3 as AWS's own real first service — launched 14 March 2006 in the US, expanding to Europe in November 2007, ahead even of EC2. Where EC2 (Chapter 3) gives you a virtual disk attached to a running machine, S3 is genuinely different: **object storage**, with no server to manage at all — you store and retrieve whole files ("objects") directly over HTTP, addressed by a key, not a filesystem path.

Buckets & Objects

An S3 **bucket** is a real, top-level container with a name that must be globally unique across the whole of AWS — not just your own account. Inside a bucket, every **object** (a file, plus metadata) is addressed by a real, flat key — S3 has no true folder hierarchy, even though its own console displays keys containing `/` as if they were nested folders.

Real Storage Classes

S3 offers several real, distinct storage classes, each trading cost against retrieval speed and access frequency:

STORAGE CLASS	REAL USE CASE
S3 Standard	Frequently accessed data — the real default
S3 Standard-IA	Infrequently accessed data still needing millisecond retrieval
S3 One Zone-IA	Infrequently accessed, easily recreated data — stored in one AZ only, at lower cost
S3 Intelligent-Tiering	Unpredictable access patterns — AWS automatically moves objects between tiers
S3 Glacier (Instant/Flexible Retrieval)	Archival data, retrieved rarely, at minutes-to-hours notice

STORAGE CLASS	REAL USE CASE
S3 Glacier Deep Archive	The cheapest real tier — long-term archival retrieved perhaps once a year, hours of retrieval time

Durability vs. Availability — a Real, Important Distinction

S3 Standard is designed for a real, documented **99.999999999%** durability figure — informally, "11 nines" — meaning the odds of losing an object due to real infrastructure failure are vanishingly small, since S3 automatically stores redundant copies across multiple facilities. **Availability** is a genuinely separate real number (typically 99.9% for S3 Standard) — the odds S3 is reachable and responsive at any given moment. An object can be perfectly durable (never lost) while briefly unavailable (temporarily unreachable) — the two numbers answer different real questions.

Versioning

S3 versioning is real, and **disabled by default**. Once enabled on a bucket, every overwrite or delete of an object keeps the prior version recoverable instead of discarding it outright — a real, direct defense against the single most common real S3 accident: an application (or a person) overwriting or deleting the wrong object.

Lifecycle Policies

A lifecycle policy automatically transitions objects between storage classes, or expires (deletes) them entirely, based on real, defined rules — for example, "move to S3 Glacier after 90 days, delete after 7 years." This turns the cost/retrieval-speed tradeoff from Chapter's own storage-class table into something managed automatically over an object's own real lifetime, rather than requiring manual intervention.

Static Website Hosting

A bucket can be configured to serve its own objects directly as a website — an `index.html`, CSS, JavaScript, and images, all served straight from S3 with no real web server to patch or manage. This works well for genuinely static content; anything needing server-side logic still needs EC2, Lambda (Chapter 7), or an equivalent compute service behind it.

A Real Case Study: The 2017 Deep Root Analytics Leak

WHAT ACTUALLY HAPPENED

In June 2017, a security researcher discovered political data on more than 198 million American citizens sitting exposed on an unsecured, publicly accessible Amazon cloud storage server belonging to Deep Root Analytics, a political data-analytics firm — widely reported at the time as a misconfigured, publicly readable S3 bucket. No breach of AWS's own real infrastructure was involved; the bucket's own access settings had simply been left open.

This is a real, direct parallel to Chapter 2's own Capital One case study, applied to storage instead of compute: the failure sat squarely on the customer's side of the shared responsibility model (Chapter 1). AWS later introduced **S3 Block Public Access**, a real, account- and bucket-level setting that can override individual bucket policies to prevent exactly this class of accidental public exposure — now recommended as the default, on-by-default safeguard for any bucket that isn't deliberately, intentionally public (like a static website).

Hands-On Exercises

EXERCISE 1

A company stores compliance records it must legally retain for 7 years, but almost never actually needs to read after the first 90 days. Design a real lifecycle policy for this data, naming which storage class(es) it should move through and when.

View solution

EXERCISE 2

Explain, in your own words, why "S3 has 11 nines of durability" does not mean "S3 is always available," and give one real, concrete scenario where these two properties could genuinely diverge.

View solution

EXERCISE 3

Using the real 2017 Deep Root Analytics leak as your example, explain why S3 Block Public Access existing as an account-level override (rather than only a per-bucket setting) is a meaningfully stronger real safeguard than simply telling every engineer to configure each bucket correctly.

View solution

CHAPTER 4 QUICK REFERENCE

- **S3** — object storage, addressed by key, no filesystem or server to manage; AWS's own first real service (2006)
- **Storage classes** — Standard, Standard-IA, One Zone-IA, Intelligent-Tiering, Glacier tiers, Glacier Deep Archive — trading cost against access speed/frequency
- **Durability (11 nines)** — the odds data is never lost; **Availability** — the odds it's reachable right now; genuinely different real numbers
- **Versioning** — off by default; keeps prior versions recoverable after an overwrite or delete
- **Lifecycle policies** — automatically transition or expire objects over time
- **Real case study** — the 2017 Deep Root Analytics leak (198 million records), a customer-side misconfiguration; S3 Block Public Access is AWS's own real, account-level fix

CHAPTER 5: VPC: VIRTUAL PRIVATE CLOUD & NETWORKING

AWS Fundamentals

Chapter 5 · VPC: Virtual Private Cloud & Networking

Amazon VPC launched 25 September 2009 — three years after EC2 (Chapter 3), once "a virtual machine anywhere on AWS's own shared network" stopped being enough, and real customers needed a genuinely isolated, self-controlled network of their own. This chapter also delivers on a real promise from Chapter 3: the full contrast between security groups and their subnet-level counterpart, Network ACLs.

What Is a VPC?

A **VPC** (Virtual Private Cloud) is a real, logically isolated section of AWS's own network, scoped to one Region, with its own private IP address range defined as a CIDR block (e.g. `10.0.0.0/16`). Everything inside it — EC2 instances, RDS databases (Chapter 6), and more — lives on that private network by default, invisible to anything outside the VPC unless you deliberately open a path.

Subnets: Public vs. Private

A VPC is carved into **subnets**, each a smaller CIDR range within the VPC's own block, and each tied to exactly one Availability Zone (Chapter 1). Whether a subnet is real "public" or "private" isn't an inherent property of the subnet itself — it's entirely determined by its **route table** (below): a subnet is public if its route table sends internet-bound traffic to an Internet Gateway, and private otherwise.

Worked Example: Carving a VPC Into Subnets

A VPC with CIDR block `10.0.0.0/16` (65,536 addresses) is split into two `/24` subnets — `10.0.1.0/24` (public) and `10.0.2.0/24` (private), each with 256 addresses on paper. A real, easy-to-miss detail: AWS reserves the first 4 and the last 1 address in every subnet for its own internal use (the network address, the VPC router, DNS, future use, and the broadcast address) — so each real `/24` subnet actually offers 251 usable addresses, not 256.

Route Tables

A **route table** is a real, ordered set of rules mapping destination CIDR ranges to a target — "traffic within the VPC's own `10.0.0.0/16` stays local," "everything else (`0.0.0.0/0`) goes to the Internet Gateway." Every subnet is associated with exactly one route table (though one route table can serve several subnets), and it's this association — not anything about the subnet's own configuration — that ultimately decides where a subnet's own traffic can go.

Internet Gateway

An **Internet Gateway** attaches directly to a VPC and provides a real, two-way path to the public internet for any subnet whose route table points to it. Instances in that subnet also need a real public IP address of their own (assigned automatically, or via Elastic IP) for this path to actually be usable — the gateway alone doesn't grant one.

NAT Gateway

A private subnet's own instances often still need real *outbound* internet access (downloading OS updates, calling an external API) without ever being reachable from the internet *inbound*. A **NAT Gateway**, deployed in a public subnet, solves exactly this: private-subnet instances route their outbound traffic to it, and it forwards that traffic to the Internet Gateway using its own public IP — but nothing on the internet can initiate a new connection back in through it. This asymmetry — outbound yes, inbound no — is the real, defining difference between a NAT Gateway and an Internet Gateway.

> **A USEFUL MENTAL MODEL**
>
> Internet Gateway = a real, two-way door. NAT Gateway = a one-way door that only opens outward, letting private resources start conversations with the internet without ever letting the internet start one with them.

Security Groups vs. Network ACLs — the Real Contrast

Chapter 3 covered security groups as stateful, instance-level firewalls. **Network ACLs (NACLs)** are the VPC's own subnet-level counterpart, and they work genuinely differently.

PROPERTY	SECURITY GROUPS	NETWORK ACLS
Applies to	Individual instances (network interfaces)	An entire subnet
Statefulness	Stateful — reply traffic automatically allowed	Stateless — inbound and outbound rules both needed explicitly
Rule types	Allow rules only	Allow AND explicit Deny rules
Evaluation	All rules evaluated; if any allows it, it's allowed	Rules evaluated in numbered order; first match wins, then stops

Because NACLs are stateless, a rule allowing inbound traffic on a port does *not* automatically allow the matching outbound reply — that needs its own separate, explicit rule, in the opposite direction. And because NACL rules are evaluated in real numbered order with the first match winning, rule ordering itself carries meaning that security groups simply don't have.

> **TWO LAYERS, NOT A CHOICE BETWEEN THEM**
>
> Security groups and NACLs aren't alternatives — real, well-designed AWS architectures use both together, as two independent layers (defense in depth): NACLs providing a coarse, subnet-wide baseline, security groups providing fine-grained, per-instance control on top of it.

Hands-On Exercises

EXERCISE 1

A subnet's route table sends 0.0.0.0/0 traffic to a NAT Gateway rather than an Internet Gateway. Explain, in your own words, whether this makes the subnet public or private, and what real capability its instances have (and lack) as a result.

View solution

EXERCISE 2

A VPC uses the CIDR block 10.0.0.0/24. Explain how many usable IP addresses this real subnet actually provides once AWS's own reserved addresses are accounted for, and why.

View solution

EXERCISE 3

A NACL has an inbound rule allowing HTTPS traffic on port 443, but no matching outbound rule exists. Explain, in your own words, whether traffic will actually flow correctly, and why this scenario couldn't happen with a security group instead.

View solution

CHAPTER 5 QUICK REFERENCE

- **VPC** — a logically isolated network within a Region, defined by a CIDR block (launched 2009)
- **Subnet** — a smaller CIDR range within a VPC, tied to one AZ; public/private is determined by its route table, not the subnet itself
- **Route table** — maps destination CIDR ranges to a target; decides where a subnet's traffic actually goes
- **Internet Gateway** — two-way internet access; **NAT Gateway** — outbound-only, for private subnets
- **Security groups** — stateful, instance-level, allow-only; **NACLs** — stateless, subnet-level, allow+deny, evaluated in numbered order
- Real subnet capacity — a /24 provides 251 usable addresses, not 256 (AWS reserves 5)

CHAPTER 6: RDS & DYNAMODB: MANAGED DATABASES

AWS Fundamentals

Chapter 6 · RDS & DynamoDB: Managed Databases

Chapter 5's own VPC now has somewhere real to put a database. AWS offers two genuinely different real managed database services covering two genuinely different data models — RDS for relational data, DynamoDB for a schema-flexible key-value/document model — and this chapter covers both, plus how to actually choose between them.

RDS: A Real Managed Relational Database

Amazon RDS launched 26 October 2009, supporting MySQL first. Real support for additional engines followed over the next several years: Oracle Database (June 2011), Microsoft SQL Server (May 2012), PostgreSQL (November 2013), and MariaDB (October 2015) — plus Amazon's own real, purpose-built engine, **Aurora** (MySQL-compatible, announced November 2014; PostgreSQL-compatible, October 2017). RDS handles real, routine operational work automatically — provisioning, OS/engine patching, and backups — that you'd otherwise manage yourself on a self-hosted database.

Multi-AZ Deployments

Announced May 2010, a real Multi-AZ deployment automatically provisions and maintains a standby replica of your database in a separate, physically distinct Availability Zone. If the primary fails (or during planned maintenance), RDS automatically fails over to that up-to-date standby — real database operations resume with no administrative intervention needed. A genuinely useful side benefit: RDS performs backups against the standby instance specifically, so the primary's own I/O is never interrupted to take one.

Read Replicas

Distinct from Multi-AZ's own failover purpose, RDS supports up to five real read replicas (for MySQL, MariaDB, and PostgreSQL) — asynchronous copies used specifically for scaling read traffic, not for failover. A real, useful engine-specific detail: MySQL and MariaDB replicas became independently writable starting October 2012, while PostgreSQL replicas remain read-only.

Automated Backups

RDS creates real automated backups with a maximum real retention period of 35 days — an initial full snapshot, then incremental snapshots afterward, letting you restore to any point within that retention window.

> **MULTI-AZ ≠ READ REPLICAS**
>
> Don't conflate the two: Multi-AZ exists for real high availability (an automatic failover target if the primary dies), while read replicas exist for real read-scaling (spreading query load across copies). A production database commonly uses both, for genuinely different reasons.

DynamoDB: A Real Departure From Relational

DynamoDB launched in January 2012, but its own real origin traces back further — to Amazon's own internal 2004 holiday season, when several of Amazon.com's own technologies genuinely failed under real Christmas-shopping traffic. That real crisis drove Amazon's own internal push toward a database purpose-built for fast, massively scalable primary-key lookups — a need traditional relational databases, tuned for complex joins and flexible queries, weren't well suited to at that scale. DynamoDB is the real, published result of that lineage.

Partition Keys & Sort Keys

A DynamoDB table's real primary key is either a single **partition key** (which physically determines where an item is stored, and must be unique across the table on its own), or a composite **partition key + sort key** — allowing many items to share one partition key, as long as each has its own unique sort key value within that partition. A specific partition-key/sort-key combination is guaranteed to identify at most one real item.

The Real Consistency Model

DynamoDB offers a genuine, explicit choice on every read:

- **Strongly consistent read** — routed to the leader node; always reflects the most recent successful write.
- **Eventually consistent read** (the real default) — routed to a random replica node instead, trading a real, small chance of staleness for lower cost and higher throughput. AWS's own documented figure: roughly a 1-in-3 real chance of reading slightly stale data within the

write's own propagation window — though in the vast majority of real cases, that window closes within milliseconds.

A REAL, DELIBERATE TRADE-OFF

Eventually consistent reads aren't a bug to work around — they're a genuine, deliberate design choice trading a small, real, usually-momentary staleness risk for meaningfully better throughput and lower cost. Reach for strongly consistent reads only where the specific real correctness need (e.g. reading your own just-written data back immediately) actually justifies the cost.

RDS vs. DynamoDB: When to Use Which

PROPERTY	RDS	DYNAMODB
Data model	Relational — tables, joins, foreign keys	Key-value/document — no joins, schema-flexible
Query flexibility	Real, complex ad-hoc SQL queries	Fast lookups by (real) known key; complex queries need careful design up front
Scaling	Vertical, plus read replicas	Near-limitless horizontal scaling, built in
Best real fit	Existing relational schemas, complex reporting, transactional integrity across related tables	High-throughput, predictable-access-pattern workloads — sessions, real-time leaderboards, IoT event data

Hands-On Exercises

EXERCISE 1

A production RDS database needs both automatic failover if the primary instance fails, and the ability to spread heavy read traffic across multiple copies. Explain, in your own words, why this genuinely requires both Multi-AZ AND read replicas, rather than either one alone.

View solution

EXERCISE 2

An application immediately reads back a record it just wrote, and the correctness of the next step in its own workflow genuinely depends on seeing that exact write. Explain which DynamoDB read type this scenario calls for, and why the default option could cause a real problem here.

View solution

EXERCISE 3

A team is building a system that needs complex, ad-hoc SQL reports joining data across many related tables, with access patterns that will likely change over time. Explain, in your own words, why DynamoDB would be a genuinely poor fit here, even though it can technically store the same underlying data.

View solution

CHAPTER 6 QUICK REFERENCE

- **RDS** — real managed relational databases (MySQL, PostgreSQL, MariaDB, Oracle, SQL Server, Aurora); launched 2009
- **Multi-AZ** — automatic failover to a standby replica; **read replicas** — scale read traffic (up to 5, engine-dependent)
- **Automated backups** — up to 35 days' retention, incremental after an initial full snapshot
- **DynamoDB** — key-value/document NoSQL, launched 2012, born from Amazon's own real 2004 holiday-traffic crisis
- **Partition key / sort key** — DynamoDB's real primary-key structure; determines physical placement and item uniqueness
- **Consistency** — eventually consistent (default, faster/cheaper) vs. strongly consistent (always current) reads

CHAPTER 7: LAMBDA & SERVERLESS COMPUTING

AWS Fundamentals

Chapter 7 · Lambda & Serverless Computing

Every service so far — EC2 (Chapter 3), RDS (Chapter 6) — runs on something you provision and keep running. Lambda, launched 13 November 2014, is genuinely different: real **serverless** compute, where you supply only code, and AWS provisions, runs, scales, and tears down the actual execution environment automatically, entirely behind the scenes.

Function as a Service

Lambda is real, event-driven "Function as a Service" (FaaS): you write a function, AWS runs it in response to a real trigger, and you're never responsible for an underlying server at all — no OS patching (Chapter 2's own shared-responsibility line shifts even further toward AWS here than it did for RDS), no idle capacity to pay for while nothing's happening.

Real Triggers & Event Sources

A Lambda function does nothing on its own — it runs only in response to a real, configured event source:

- **S3 events** — a function fires automatically when an object is uploaded to a bucket (Chapter 4), e.g. resizing an image the moment it lands.
- **API Gateway** — a function fires in response to a real, incoming HTTP request, turning it into a genuine REST/HTTP API endpoint with no server behind it.
- **DynamoDB Streams** — a function fires when items change in a DynamoDB table (Chapter 6).
- **EventBridge (scheduled events)** — a function fires on a real, defined schedule (cron-style), a serverless alternative to a scheduled task on a persistent server.

Building a Real Serverless API

API Gateway plus Lambda is a real, common pattern for a fully serverless backend: API Gateway receives an HTTP request, invokes the corresponding Lambda function with that request's own data, and returns the function's response back to the caller — with no EC2 instance, load balancer, or persistent process running between requests at all.

The Real Pricing Model

Lambda charges for exactly two real things: the **number of requests**, and the **duration** a function actually runs, measured in GB-seconds (memory allocated × execution time). AWS's own real, documented free tier covers 1 million requests and 400,000 GB-seconds of compute every month, at no cost. Genuinely different from EC2 (Chapter 3), where an On-Demand instance bills by the hour whether it's doing useful work or sitting idle — a Lambda function costs nothing at all while it isn't running.

> **WHERE SERVERLESS GENUINELY WINS ON COST**
>
> A workload that runs briefly and infrequently (a few seconds, a handful of times an hour) is often dramatically cheaper on Lambda than on even the smallest always-on EC2 instance, since you're never paying for idle time between invocations at all.

Cold Starts

When a function hasn't run recently, Lambda must provision a fresh execution environment before running it — a real, measurable delay called a **cold start**. This delay genuinely varies by language: Rust and Go, which compile to native static binaries, start with minimal real overhead, while Java and C# tend to see longer real cold starts due to their own runtime/JVM initialization cost. AWS's own real mitigation for Java specifically, **SnapStart**, significantly reduces this overhead by resuming from a pre-initialized snapshot rather than starting cold every time.

A Real, Hard Boundary: The Execution Limit

> **LAMBDA ISN'T FOR EVERYTHING**

Every Lambda function has a real, hard maximum execution timeout of 15 minutes. Anything genuinely long-running — a multi-hour batch job, a persistent WebSocket connection, a process that must keep real state in memory across requests — doesn't fit Lambda's own execution model at all, and belongs on EC2 (Chapter 3) or a container-based service instead.

EC2 vs. Lambda

PROPERTY	EC2	LAMBDA
You manage	The entire guest OS and runtime	Only your own function code
Billing	Per hour/second the instance runs, idle or not	Per request + actual execution duration only
Max run time	Unlimited (as long as the instance runs)	15 minutes, hard limit
Best real fit	Long-running processes, persistent connections, full control needed	Short, event-driven, bursty, or infrequent workloads

A Lambda function still needs an IAM role (Chapter 2) — least privilege applies exactly as much here as it did to an EC2 instance's own attached role in Chapter 3; "serverless" changes what you manage, not whether permissions still need to be scoped correctly.

Hands-On Exercises

EXERCISE 1

A function processes a real image upload roughly 200 times per day, each run taking about 2 seconds. Using the chapter's own real free tier figures, explain whether this workload would likely stay within Lambda's real free tier for requests, and roughly why.

View solution

EXERCISE 2

A team wants to run a data-processing job that reliably takes about 45 minutes to complete. Explain, in your own words, why Lambda is a genuinely poor fit for this specific job, regardless of cost.

View solution

EXERCISE 3

Explain, in your own words, why a Java-based Lambda function is more likely to suffer from noticeable cold starts than a Go-based one, and how SnapStart specifically addresses this for Java.

View solution

CHAPTER 7 QUICK REFERENCE

- **Lambda** — real serverless FaaS, launched 2014; no server to provision, patch, or manage
- **Triggers** — S3 events, API Gateway (HTTP requests), DynamoDB Streams, EventBridge schedules
- **Pricing** — per request + duration (GB-seconds); real free tier: 1M requests + 400,000 GB-seconds/month
- **Cold starts** — real, language-dependent startup delay; SnapStart mitigates it for Java
- **Execution limit** — 15 minutes, hard maximum; long-running work belongs on EC2 instead
- IAM roles and least privilege (Chapter 2) still apply — serverless doesn't mean permission-less

CHAPTER 8: LOAD BALANCING & AUTO SCALING

AWS Fundamentals

Chapter 8 · Load Balancing & Auto Scaling

Elastic Load Balancing and Auto Scaling are real, deliberately paired AWS services — announced together on 23 October 2008 and launched together on 18 May 2009. This chapter covers both, and the real, standard high-availability pattern they form when combined with the multi-AZ architecture from Chapter 1.

Three Real Load Balancer Types

AWS's original **Classic Load Balancer** operates at Layer 4 (transport layer) — routing purely by IP and port, with no awareness of HTTP content. Two real, more capable successors followed:

- **Application Load Balancer (ALB)** — launched 11 August 2016, operating at Layer 7 (application layer). ALB can route by URL path or HTTP header — sending `/api` traffic to one backend and `/mobile` to another — and performs fine-grained, per-port health checks, making it the real standard choice for microservices and container-based applications.
- **Network Load Balancer (NLB)** — launched 7 September 2017, staying at Layer 4 but engineered for real, extreme scale and speed: AWS's own internal testing measured over 3 million requests per second at 30 Gbps before exhausting test resources. NLB provides one real static IP address per Availability Zone and preserves the original client source IP untouched — properties ALB doesn't offer — making it the real fit for long-running connections, gaming, IoT, and anything needing a fixed, hardcodable IP.

> **CHOOSING BETWEEN ALB AND NLB**
>
> Reach for ALB by default for real HTTP/HTTPS web traffic and microservices needing content-based routing. Reach for NLB specifically when you need a static IP, must preserve the real original client IP without extra headers, or need genuinely extreme throughput at the lowest possible latency.

Health Checks

A load balancer periodically sends a real, configured health-check request to each registered target (an EC2 instance, a Lambda function, an IP address) and routes new traffic only to targets currently reporting healthy. An instance that starts failing its own health checks is automatically removed from rotation — no manual intervention needed — and automatically added back once it starts passing again.

Auto Scaling Groups

An **Auto Scaling Group (ASG)** maintains a real, defined range of running instances — a minimum, a maximum, and a desired capacity — launching new instances from a template when capacity needs to grow, and terminating instances when it needs to shrink. Real, common scaling policies include:

- **Target tracking** — e.g. "keep average CPU utilization at 60%"; the ASG automatically adds or removes instances to hold that target.
- **Step scaling** — add or remove a specific number of instances once a real metric crosses a defined threshold.
- **Scheduled scaling** — proactively scale capacity up or down at known, predictable times (e.g. before a real daily traffic peak).

An ASG also directly replaces an unhealthy instance the moment a health check (whether the ASG's own, or the attached load balancer's) reports it as failed — the same self-healing principle Chapter 3's own EC2 coverage never had a mechanism for on its own.

The Real High-Availability Pattern

Combining everything so far into one real, standard architecture: an Auto Scaling Group spreads its own instances across multiple Availability Zones within a Region (Chapter 1), a load balancer sits in front distributing traffic only to the healthy ones (this chapter), and the ASG automatically replaces any instance that fails, or scales the whole group up or down as real demand changes. No single instance failure, and no single AZ failure, takes the application down.

A LOAD BALANCER ALONE ISN'T HIGH AVAILABILITY

A load balancer distributing traffic across a fixed, unchanging set of instances still leaves you exposed if one of those instances fails and nothing ever replaces it. The real high-availability guarantee comes from combining a load balancer WITH an Auto Scaling Group — the two genuinely need each other for the pattern to hold up.

ALB vs. NLB at a Glance

PROPERTY	APPLICATION LOAD BALANCER	NETWORK LOAD BALANCER
OSI Layer	7 (application)	4 (transport)
Real launch date	11 August 2016	7 September 2017
Routing	By URL path / HTTP header	By IP/port only
Static IP	No	Yes — one per AZ
Best real fit	Web apps, microservices, containers	Extreme throughput, gaming, IoT, fixed-IP needs

Hands-On Exercises

EXERCISE 1

A company builds a real microservices application needing to route /orders and /inventory requests to two genuinely different backend services based on the URL path. Recommend a real load balancer type for this, and explain why the alternative would be a poor fit.

View solution

EXERCISE 2

An Auto Scaling Group has a target tracking policy set to keep average CPU utilization at 60%. Explain, in your own words, what the ASG will do if real average CPU utilization rises to 85%, and what it will do if it later drops to 30%.

View solution

EXERCISE 3

A team places a load balancer in front of three manually launched EC2 instances, but doesn't use an Auto Scaling Group. Explain, in your own words, why this setup is genuinely NOT the same as real high availability, even though traffic is being distributed across multiple instances.

View solution

CHAPTER 8 QUICK REFERENCE

- **Elastic Load Balancing & Auto Scaling** — announced together (Oct 2008), launched together (May 2009)
- **ALB** (2016) — Layer 7, content-based routing, the real default for web apps/microservices
- **NLB** (2017) — Layer 4, static IPs, source-IP preservation, extreme throughput
- **Health checks** — remove unhealthy targets from rotation automatically, add them back once healthy
- **Auto Scaling Group** — min/max/desired capacity, target tracking / step / scheduled scaling policies
- **Real HA pattern** — ASG spread across multiple AZs + load balancer + automatic unhealthy-instance replacement, together

CHAPTER 9: CLOUDWATCH & COST MANAGEMENT

AWS Fundamentals

Chapter 9 · CloudWatch & Cost Management

CloudWatch launched alongside Elastic Load Balancing and Auto Scaling, on the exact same real day — 18 May 2009 — as part of one coordinated announcement. That's not a coincidence: Chapter 8's own auto scaling policies (target tracking, step scaling) work specifically because CloudWatch supplies the real metrics they scale against.

CloudWatch Metrics — and a Real, Common Gotcha

CloudWatch automatically collects real infrastructure metrics — CPU utilization, disk I/O, network traffic, even RDS replica lag (Chapter 6) — with zero setup required.

> **MEMORY AND DISK SPACE AREN'T AUTOMATIC**
>
> A genuinely common, real surprise: CloudWatch does *not* report memory usage or disk space utilization for an EC2 instance out of the box — only metrics visible from outside the instance (CPU, network, disk I/O) are collected automatically. Getting real memory or disk-space metrics requires installing the CloudWatch Agent (available for Windows and Linux since December 2017) inside the instance itself. Teams that assume "CloudWatch monitors everything" by default are often missing exactly these two metrics without realizing it.

Custom Metrics

Since May 2011, CloudWatch has accepted real, custom application-level metrics submitted programmatically via its own API — anything your own code wants to track (queue depth, business-specific counters, cache hit rate) that AWS's own infrastructure-level metrics could never know about.

CloudWatch Logs

CloudWatch Logs, available since July 2014, centralizes real log data from EC2 instances, Lambda functions (Chapter 7), and other AWS services in one searchable place. Retention is real and configurable per log group — anywhere from 1 day up to 10 years, or set to never expire — and a genuinely easy real mistake is leaving it at its own default of indefinite retention, quietly accumulating real storage cost over months and years for logs nobody ever looks at again.

Alarms & Dashboards

A CloudWatch **alarm** watches a real metric against a defined threshold and triggers an action once it's crossed — directly feeding Chapter 8's own Auto Scaling target tracking policies, or sending a real notification (via SNS) to alert a human. A **dashboard** is a real, customizable visualization combining multiple metrics from across services into one view.

Real Cost Management Tools

Two real, distinct tools handle cost specifically:

- **Cost Explorer** — a real, visual tool for analyzing spend over time, broken down by service, account, or tag — the place to answer "where is our money actually going, and why did it change."
- **AWS Budgets** — set a real, specific dollar (or usage) threshold, and get notified as actual or forecasted spend approaches or crosses it.

> **A REAL, CRITICAL MISUNDERSTANDING**
>
> AWS Budgets *alerts* — it does not, on its own, *stop* spending. Crossing a budget threshold sends a real notification; it does not shut down resources, revoke access, or halt billing automatically. Genuinely preventing runaway spend (for example, from a leaked access key being used to launch unauthorized compute, echoing Chapter 2's own root-account and least-privilege warnings) requires separate, deliberate controls — service quotas, billing alarms wired to an automated Lambda response, or IAM restrictions — not a Budget alone.

CloudWatch vs. Cost Explorer vs. Budgets

TOOL	ANSWERS	TAKES ACTION?
CloudWatch	"Is this resource healthy right now?"	Yes — alarms can trigger scaling or notifications
Cost Explorer	"Where has money been spent, and why?"	No — analysis only
AWS Budgets	"Are we approaching a spending threshold?"	Notification only — does not stop spending on its own

Hands-On Exercises

EXERCISE 1

A team assumes CloudWatch is already alerting them if an EC2 instance's own disk fills up, but no alert ever fires before the disk actually runs out. Explain, in your own words, why this happened, and what real step was missing.

View solution

EXERCISE 2

A company sets an AWS Budget alert at $5,000/month, and a compromised access key is later used to launch a large amount of unauthorized compute. Explain, in your own words, why the Budget alert alone would not have prevented the unexpected charges from accumulating.

View solution

EXERCISE 3

A team wants to know exactly which specific service caused their AWS bill to increase by 20% last month. Recommend the real, correct tool for this from the chapter, and explain why CloudWatch alone wouldn't answer this question.

View solution

CHAPTER 9 QUICK REFERENCE

- **CloudWatch** — launched 18 May 2009, alongside ELB & Auto Scaling
- Real gotcha: memory & disk-space metrics need the CloudWatch Agent (since Dec 2017) — not automatic
- **Custom metrics** (since 2011), **CloudWatch Logs** (since 2014, configurable 1 day–10 years retention)
- **Alarms** — trigger Auto Scaling (Chapter 8) or SNS notifications when a threshold is crossed
- **Cost Explorer** — analyzes past spend; **AWS Budgets** — alerts on a threshold, but does NOT stop spending on its own

CHAPTER 10: CAPSTONE — DEPLOYING A SIMPLE WEB APPLICATION ON AWS

AWS Fundamentals

Chapter 10 · Capstone: Deploying a Simple Web Application on AWS

Nine chapters, one piece at a time. This closing chapter combines every one of them into a single, realistic architecture — a small web application handling user photo uploads, with a database, background image processing, and real monitoring — built the way a genuine, small production deployment actually would be.

The Target Architecture

Users upload a photo through a web app; the app stores metadata in a database, saves the original image to object storage, and a background job automatically generates a thumbnail. Every piece of this deliberately gives every prior chapter's own technique a real, concrete place to apply.

Step 1 — Region & Multi-AZ Planning

Chapter 1: the whole deployment lands in one Region, with the web and database tiers deliberately spread across two Availability Zones from the very start — not retrofitted later once something has already failed.

Step 2 — An IAM Role, Not Embedded Keys

Chapter 2: a dedicated IAM role is created for the web servers, scoped to exactly what they need — `s3:PutObject` / `s3:GetObject` on the one specific application bucket, and permission to write to CloudWatch Logs. Nothing broader, and no long-term access key ever touches the application's own code.

```
{
  "Effect": "Allow",
  "Action": ["s3:PutObject", "s3:GetObject"],
```

```
    "Resource": "arn:aws:s3:::app-photo-bucket/*"
  }
```

Step 3 — VPC, Subnets & the Two Gateways

Chapter 5: one VPC, public subnets (route to an Internet Gateway) for the web tier, and private subnets (route to a NAT Gateway) for the database — outbound-only internet access for the database tier, never inbound, exactly as Chapter 5's own asymmetry described.

Step 4 — EC2 Web Servers

Chapter 3: web servers launch from a custom AMI (the application pre-installed), sized as `t3.small` — general-purpose, matching the workload's own modest, steady traffic. The step 2 IAM role is attached at launch; a security group allows inbound HTTP only from the load balancer (Step 8), never directly from the internet.

Step 5 — S3 for the Photos

Chapter 4: the application bucket stores original uploads in S3 Standard, with versioning enabled (protecting against an accidental overwrite) and Block Public Access left firmly on — exactly the real, account-level safeguard the chapter's own Deep Root Analytics case study made the case for. Nothing in this bucket is ever meant to be directly public.

Step 6 — RDS for Application Data

Chapter 6: a Multi-AZ RDS PostgreSQL instance sits in the private subnets from Step 3, storing photo metadata (uploader, timestamp, S3 key). Its own security group allows inbound traffic only from the web tier's security group — never a public IP, never `0.0.0.0/0`.

Step 7 — A Lambda Function for Thumbnails

Chapter 7: an S3 event trigger fires a Lambda function the moment a new photo lands in the bucket, generating a real thumbnail and saving it back to S3 under a separate key prefix — a genuinely well-suited serverless job (short, event-driven, bursty) that would be wasteful to run on a permanently-provisioned EC2 instance instead. Its own IAM role is scoped identically narrowly to Step 2's — read the original, write the thumbnail, nothing more.

Step 8 — Load Balancer & Auto Scaling

Chapter 8: an Application Load Balancer (real content-based routing isn't needed here, but ALB's own container/microservice-friendly health checks are still the right default) sits in front of the web tier, distributing traffic across an Auto Scaling Group spanning both AZs from Step 1. A target tracking policy holds average CPU at 60% — the same worked example from Chapter 8 itself.

Step 9 — Monitoring & a Budget

Chapter 9: a CloudWatch alarm watches RDS free storage space (a metric collected automatically, unlike the memory/disk-space gotcha for EC2) and notifies the team before it runs low; the ASG's own target tracking policy from Step 8 is fed directly by CloudWatch's CPU metric. An AWS Budget is set at a real, planned monthly threshold — understood, per Chapter 9's own warning, as an alert mechanism, not an automatic spending cutoff.

The Full Picture

STEP	CHAPTER
1	Ch.1 — Region & multi-AZ planning
2	Ch.2 — Scoped IAM role, no embedded keys
3	Ch.5 — VPC, public/private subnets, Internet/NAT Gateways
4	Ch.3 — EC2 instances, security groups
5	Ch.4 — S3 storage, versioning, Block Public Access
6	Ch.6 — Multi-AZ RDS, scoped security group

STEP	CHAPTER
7	Ch.7 — Lambda triggered by an S3 event
8	Ch.8 — ALB + Auto Scaling Group, target tracking
9	Ch.9 — CloudWatch alarms + AWS Budget

What's Still Out of Scope, Honestly

This capstone deliberately stops short of several real, genuine next steps: no CI/CD deployment pipeline for the application code itself, no CloudFront CDN or WAF in front of the ALB, no Route 53 DNS configuration, no multi-Region disaster recovery, and no containerization (ECS/EKS) as an EC2 alternative. Each is a real, legitimate next step for a production system — named honestly as still outstanding rather than treated as solved here, the same discipline this site applies to every other course's own capstone.

THE THROUGH-LINE, RESTATED

Every step in this capstone is a direct, concrete application of a chapter's own material — nothing new was introduced here. Least privilege (Ch.2), private-by-default networking (Ch.5), scoped security groups (Ch.3/Ch.6), and honest monitoring (Ch.9) all show up together, repeatedly, because they're the same small set of real disciplines applied consistently across every AWS service, not a separate lesson for each one.

SECURITY RUNS THROUGH EVERY CHAPTER, NOT JUST CHAPTER 2

Chapter 1's shared responsibility model, Chapter 2's least privilege, Chapter 3's IMDSv2, Chapter 4's Block Public Access, and Chapter 5's NACLs/security groups are all real, separate mechanisms — but they're all answering the same underlying real question: what's the smallest possible blast radius if any single one of them fails. This capstone's own architecture never relies on just one of them holding up alone.

Closing the Course

From Chapter 1's real global infrastructure and shared responsibility model to this chapter's own working, multi-service architecture — every chapter in between covered one specific,

genuine piece of what it actually takes to run something real on AWS. Compute, storage, networking, databases, serverless, scaling, and monitoring together are what make a deployment resilient and secure — not any single service used in isolation.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why Step 7's thumbnail-generation job was built as a Lambda function rather than as a service running continuously on the Step 4 EC2 web servers, using the chapter's own real cost/execution-model reasoning from Chapter 7.

View solution

EXERCISE 2

Trace, step by step, every real point in this capstone's own architecture where least privilege (Chapter 2's principle) is specifically applied, and explain what would go wrong at each point if it weren't.

View solution

EXERCISE 3

A teammate suggests this capstone's own AWS Budget alone is sufficient to prevent unexpected overspending. Using Chapter 9's own real distinction, explain why this claim is incomplete, and name one concrete real scenario from earlier chapters where it would fail to hold.

View solution

COURSE COMPLETE — AWS FUNDAMENTALS

- 10/10 chapters — from AWS's real 2006 origin and global infrastructure through a full, working multi-service architecture
- Every chapter's own real service (IAM, EC2, S3, VPC, RDS/DynamoDB, Lambda, ALB/Auto Scaling, CloudWatch) applied together in this capstone, not in isolation

- A recurring real thread: least privilege, private-by-default networking, and honest monitoring, applied consistently rather than as one-off lessons
- Real, honest next steps named rather than glossed over: CI/CD, CDN/WAF, DNS, multi-Region DR, containerization
- Next in the Cloud Platform Specifics arc: Azure Fundamentals (`azure1`) and GCP Fundamentals (`gcp1`), deliberately structured chapter-for-chapter in parallel with this course