CLOUD & DEVOPS

Azure Fundamentals

Azure's real ecosystem, resource groups & RBAC, Entra ID, virtual machines, Blob Storage, virtual networks, Azure SQL Database & Cosmos DB, Azure Functions, load balancing & scale sets, Azure Monitor & cost management, and a closing capstone rebuilding AWS Fundamentals' own capstone application on Azure's genuinely different architecture.

10 Chapters | Philip Osztromok | Generated with Claude

CHAPTER 1: WHY AZURE? ECOSYSTEM, RESOURCE MANAGER & RESOURCE GROUPS

Azure Fundamentals

Chapter 1 · Why Azure? Ecosystem, Resource Manager & Resource Groups

This is the second course in the site's own three-course Cloud Platform Specifics split, deliberately kept structurally parallel to AWS Fundamentals chapter-for-chapter — but Azure genuinely isn't just "AWS with different names." This chapter starts with its real, separate origin, and flags a genuine structural difference from AWS worth understanding from the very first chapter.

Azure's Real Origin

Azure was first introduced at Microsoft's own Professional Developers Conference in October 2008, under the real internal codename "Project Red Dog." It officially launched as **Windows Azure** in February 2010, and was renamed **Microsoft Azure** on 25 March 2014 — the same real year Microsoft introduced Azure Resource Manager (below), reflecting the platform's own broader shift beyond just Windows workloads.

Azure's Real Global Infrastructure

By fiscal year-end 2025, Microsoft reported Azure operating over 400 real datacenters across 70 regions — a genuinely large real footprint, grown from 54 regions as recently as 2018. Azure also runs a separate real CDN network of 118 point-of-presence Edge locations across 100 cities (as of January 2023), a similar concept to AWS's own Edge Locations from Chapter 1 of AWS Fundamentals.

> **A REAL, GENUINE STRUCTURAL DIFFERENCE FROM AWS**
>
> AWS Fundamentals Chapter 1 established that every AWS Region always contains multiple Availability Zones by design. Azure genuinely works differently: **not every Azure region supports Availability Zones at all**. Only a real, specific, documented subset of Azure's own 70+ regions offer AZ support — many others are single-datacenter regions with no zone redundancy available inside them. Before assuming a multi-AZ deployment pattern is possible in a given Azure region, you have to actually check whether that specific region supports it.

Availability Zones — Where They Exist

Where a region does support them, an Azure **Availability Zone** is a real, logical grouping of one or more physically separate datacenters, each with independent power, cooling, and networking — the same underlying concept as an AWS AZ. Zones within a region are typically separated by several real kilometers (usually within 100 km of each other), and Microsoft's own real target for inter-zone network latency is under roughly 2 milliseconds. A genuinely useful real detail: Azure does not charge for data transfer between Availability Zones within the same region.

> **A REAL, EASY-TO-MISS QUIRK**
>
> Each physical datacenter is mapped to a "logical" zone number (Zone 1, Zone 2, Zone 3) — but that mapping is assigned per Azure subscription. Your own subscription's "Zone 1" may correspond to a genuinely different physical datacenter than another subscription's own "Zone 1" in the exact same region. There's no universal, shared meaning to a zone number across subscriptions — a real detail worth knowing before assuming two teams' "Zone 1" resources are physically co-located.

Zone-Redundant vs. Zonal Resources

Azure services that support Availability Zones typically offer two real deployment modes: **zone-redundant** (Microsoft automatically replicates or spreads the resource across multiple zones, and handles failover itself), and **zonal** (you deploy into one specific zone yourself, gaining fault isolation from other zones but taking on responsibility for your own multi-zone failover architecture, if you want one).

Azure Resource Manager & Resource Groups

Introduced in 2014, **Azure Resource Manager (ARM)** lets you organize related resources — a virtual machine, its storage, its network interface — into a real, named **Resource Group**, deployed, managed, monitored, and (if needed) deleted together as one logical unit. This is a genuinely different default organizational model from AWS, which has no single mandatory grouping construct of this kind — AWS resources are more commonly organized loosely via tags and account/VPC boundaries instead. In Azure, every single resource belongs to exactly one Resource Group from the moment it's created.

The Shared Responsibility Model Still Applies

Exactly the same real principle AWS Fundamentals Chapter 1 introduced holds here too: Microsoft secures the underlying physical infrastructure, while you remain responsible for your own data, access configuration, and (depending on the specific service) operating system and application layer. Nothing about that division changes just because the provider does.

Azure's Real Market Position

Per the same real, documented Synergy Research Group data already cited in AWS Fundamentals Chapter 1 (Q1 2023): Azure holds roughly 25% of the global cloud infrastructure market — second place, behind AWS's real ~31% and ahead of Google Cloud's real ~11%.

AWS vs. Azure — The Region/Zone Model, Compared

PROPERTY	AWS	AZURE
Real launch	March 2006	February 2010 (as Windows Azure)
Real current scale	38 Regions, 120 AZs (Oct 2025)	70+ Regions, 400+ datacenters (FY2025)
Do all regions have AZs?	Yes, always	No — only a specific documented subset
Mandatory resource grouping	None (tags/VPC/account boundaries)	Resource Groups — every resource belongs to one
Real Q1 2023 market share	~31%	~25%

Hands-On Exercises

EXERCISE 1

A team assumes, based on their own prior AWS experience, that every Azure region they might deploy to will automatically support multiple Availability Zones. Explain, in your own words, why this assumption is genuinely risky, and what real step they should take before designing a multi-zone architecture in a specific Azure region.

View solution

EXERCISE 2

Explain, in your own words, why "Team A's Zone 1" and "Team B's Zone 1" in the same Azure region cannot be safely assumed to be the same physical datacenter, and what real, practical implication this has for two teams trying to deliberately co-locate resources for low latency.

View solution

EXERCISE 3

Explain, in your own words, how Azure Resource Groups genuinely change the default experience of organizing related cloud resources compared to AWS's own looser, tag-based approach, and name one real, practical benefit this mandatory grouping provides.

View solution

CHAPTER 1 QUICK REFERENCE

- **Azure's real origin** — announced Oct 2008 ("Project Red Dog"), launched Feb 2010 as Windows Azure, renamed Microsoft Azure 25 March 2014
- **Real current scale** — 70+ regions, 400+ datacenters (FY2025), 118 CDN Edge locations across 100 cities
- Real structural difference from AWS: **not every Azure region supports Availability Zones** — check before assuming multi-zone deployment is possible
- **Zone-redundant** (Microsoft manages failover) vs. **zonal** (you manage it yourself) resource deployment
- **Azure Resource Manager & Resource Groups** (2014) — every resource belongs to exactly one named group, unlike AWS's own looser tag-based model
- **Real market position** — Azure ~25%, behind AWS's ~31%, ahead of GCP's ~11% (Synergy Research, Q1 2023)

CHAPTER 2: MICROSOFT ENTRA ID & ROLE-BASED ACCESS CONTROL

Azure Fundamentals

Chapter 2 · Microsoft Entra ID & Role-Based Access Control

AWS Fundamentals Chapter 2 covered IAM — users, groups, roles, and policies, scoped per AWS account. Azure's own identity system is genuinely shaped differently from the very foundation up, and this chapter covers exactly how.

Microsoft Entra ID: Real Origin & Rename

Microsoft's own identity service launched 27 October 2008 as **Azure Active Directory (Azure AD)** — the same real month Azure itself was first announced (Chapter 1). Microsoft renamed it **Microsoft Entra ID**, announced 11 July 2023 and effective 15 July 2023, as part of a real, broader push to bring naming consistency across Microsoft's own cloud product line.

A Real Structural Difference From AWS IAM

> **IDENTITIES LIVE ABOVE THE SUBSCRIPTION, NOT INSIDE IT**
>
> AWS IAM identities are scoped per account. Entra ID identities are scoped per **tenant** — and one tenant can genuinely span multiple Azure subscriptions at once. A single Entra ID user or group can be granted access across several subscriptions without needing a separate identity created in each one — a real, structurally different starting point from AWS, where an IAM user only ever exists inside the one account it was created in.

The Real RBAC Scope Hierarchy

Azure's role-based access control (RBAC) applies permissions at a real, nested scope, with each level inheriting downward:

- **Management Group** — the broadest real scope, grouping multiple subscriptions together.
- **Subscription** — a billing and access boundary, roughly analogous to an AWS account.
- **Resource Group** — Chapter 1's own real, mandatory grouping construct.

- **Resource** — an individual real resource (a VM, a storage account).

A role granted at a higher level (say, a Resource Group) automatically applies to everything inside it — every resource in that group — unless overridden more narrowly at a lower level. This is a genuinely different default than AWS IAM's own flatter model, where a policy attaches directly to a specific identity rather than inheriting down a real, nested scope tree.

Real Built-in Roles

BUILT-IN ROLE	REAL PERMISSIONS
Owner	Full access to all resources, including the ability to grant access to others
Contributor	Full access to manage resources, but cannot grant access to others
Reader	Can view existing resources, but cannot make any changes

The real, meaningful distinction between Owner and Contributor — the ability to grant access to other identities — is exactly the kind of narrow, deliberate permission split least privilege (introduced in AWS Fundamentals Chapter 2) calls for: most real day-to-day work needs Contributor, not Owner.

Role Assignments

A real Azure role assignment combines exactly three things: a **role definition** (what's allowed — e.g. Reader), a **security principal** (who — a user, group, or Managed Identity), and a **scope** (where — a Management Group, Subscription, Resource Group, or single Resource). This maps conceptually onto AWS's own Policy + Identity + Resource model, but structured through Azure's own real, hierarchical scope system instead of AWS's flatter policy-attachment approach.

Managed Identities

Azure's real, direct equivalent of AWS's own IAM roles for services (Chapter 3 of AWS Fundamentals) is the **Managed Identity** — an identity Azure automatically creates and manages for a resource (a VM, a Function App), with no credentials for you to store, rotate, or

leak at all. A **system-assigned** Managed Identity is tied to one specific resource's own lifecycle (deleted when the resource is); a **user-assigned** Managed Identity exists independently and can be attached to multiple resources at once.

> **THE SAME REAL LESSON, A DIFFERENT NAME**
>
> Exactly as AWS Fundamentals Chapter 2 recommended IAM roles over embedded access keys, Azure's own real, standard guidance is the same: attach a Managed Identity to a resource rather than embedding a service principal's credentials directly in application code.

A Real, Recent Case Study: The 2025 Entra ID Vulnerability

> **WHAT ACTUALLY HAPPENED**
>
> On 14 July 2025, a real, documented security vulnerability was discovered that could allow a user to gain administrator privileges across every Entra ID directory — a genuinely severe real scope, since Entra ID underlies identity for the whole of Azure, Microsoft 365, and connected third-party applications. Microsoft issued a real fix three days later, on 17 July 2025.

This is a real, current illustration of why identity — not any individual resource — sits at the real center of cloud security: a flaw in the identity layer itself has the potential to undermine every other real access control covered in this chapter, regardless of how carefully RBAC scopes and role assignments are otherwise configured.

AWS IAM vs. Azure RBAC

PROPERTY	AWS IAM	AZURE RBAC (ENTRA ID)
Identity scope	Per AWS account	Per tenant — can span multiple subscriptions
Permission model	Policies attached directly to identities	Role assignments at a nested scope (inherits downward)
Service identity	IAM role	Managed Identity (system- or user-assigned)
Broadest built-in role	N/A (no single default "full access" role)	Owner

Hands-On Exercises

EXERCISE 1

A team wants a new employee to have Contributor access to every resource in three specific subscriptions, without creating three separate identities. Explain, in your own words, how Entra ID's tenant-wide identity model makes this genuinely simpler than it would be under AWS's own per-account IAM model.

View solution

EXERCISE 2

A Contributor role is assigned at the Resource Group level. Explain, in your own words, what real access this grants to a new resource created inside that Resource Group afterward, without any additional role assignment being made.

View solution

EXERCISE 3

Explain, in your own words, why the real 2025 Entra ID vulnerability was potentially so severe compared to a flaw in a single Azure service, using the chapter's own point about identity sitting at the center of cloud security.

View solution

CHAPTER 2 QUICK REFERENCE

- **Entra ID** — launched Oct 2008 as Azure AD, renamed Microsoft Entra ID July 2023
- Real structural difference: identities are scoped per **tenant**, spanning multiple subscriptions — unlike AWS IAM's per-account scope
- **RBAC scope hierarchy** — Management Group → Subscription → Resource Group → Resource, inheriting downward
- **Built-in roles** — Owner (full + can grant), Contributor (full, can't grant), Reader (view only)

- **Managed Identity** — Azure's real equivalent of an IAM role for services; system-assigned or user-assigned
- Real case study: the 2025 Entra ID privilege-escalation vulnerability (discovered 14 July, fixed 17 July) — a stark, current reminder that identity sits at the center of cloud security

CHAPTER 3: AZURE VIRTUAL MACHINES

Azure Fundamentals

Chapter 3 · Azure Virtual Machines

AWS Fundamentals Chapter 3 covered EC2 — AWS's own foundational, day-one compute service. Azure's own virtual machine service has a real, genuinely different history: Azure itself launched in February 2010 as a platform-as-a-service offering first, and general-purpose Windows and Linux virtual machines weren't announced until June 2012, over two years later — VMs were a real, deliberate later addition to Azure, not its original foundation the way EC2 was for AWS.

Real VM Sizes & Series

Azure organizes VM sizes into real families, each tuned for a different workload shape — genuinely analogous in spirit to AWS's own instance families (Chapter 3 of AWS Fundamentals), though named and structured differently:

FAMILY	TYPE	TUNED FOR
A-family	General purpose	Entry-level, economical workloads — dev/test, small databases
B-family	General purpose	Burstable — a real, distinctive CPU credit model (below)
D-family	General purpose	Balanced CPU/memory — enterprise applications, the real default choice
F-family	Compute optimized	High CPU-to-memory ratio — web servers, batch processing, gaming
E-family	Memory optimized	Relational databases, in-memory analytics
M-family	Ultra memory optimized	Extremely large in-memory databases, real massive RAM needs
L-family	Storage optimized	High disk throughput — big data, NoSQL databases
N-family (NC/ND/NG/NV)	GPU accelerated	AI/ML training, rendering, cloud gaming

FAMILY	TYPE	TUNED FOR
H-family (HB/HC/HX)	High performance compute	Scientific simulations, genomics, computational fluid dynamics

A Real, Distinctive Feature: B-Series CPU Credits

Unlike every other Azure VM family, B-series instances run on a real **CPU credit** model: while a B-series VM's actual usage stays below its own baseline CPU performance level, it accumulates real credits; when usage rises above that baseline, it spends those accumulated credits to burst to higher performance. Once all credits are exhausted, the VM is throttled back down to its own base performance level until it accumulates enough credits to burst again. This makes B-series VMs a genuinely good real fit for workloads with unpredictable, spiky-but-mostly-idle demand — web servers, dev environments, proof-of-concept projects.

Real Pricing Models

- **Pay-As-You-Go** — billed per second/hour, no commitment, the most flexible and most expensive real rate.
- **Reserved VM Instances** — commit to 1 or 3 years for a real, significant discount, broadly comparable in spirit to AWS's own Reserved Instances.
- **Spot VMs** — real discounts of up to 90% off Pay-As-You-Go pricing, using Azure's own spare capacity.

A REAL, GENUINE DIFFERENCE FROM AWS SPOT

Azure Spot VMs can be evicted for two real, distinct reasons — not just one: **capacity-based eviction** (Azure needs the compute capacity back, the same real reason AWS Spot instances get reclaimed), and a second, genuinely Azure-specific mechanism, **price-based eviction** — you set a maximum price you're willing to pay, and if the real current Spot price rises above it, your VM is evicted even if capacity is otherwise available. A Spot VM workload needs to tolerate both real interruption reasons, not just one.

Network Security Groups (NSGs)

A REAL, GENUINE STRUCTURAL DIFFERENCE FROM AWS

AWS Fundamentals Chapter 5 taught two genuinely separate constructs — stateful security groups at the instance level, and stateless NACLs at the subnet level. Azure uses just **one** real construct instead: the **Network Security Group (NSG)**, which is stateful (a flow record tracks each connection, so an allowed inbound request automatically permits its own reply back out) — and can be applied at *either* the subnet level, the individual network interface level, or both at once. There's no separate stateless equivalent to AWS's own NACLs in Azure's model at all.

An NSG's own rules are evaluated in real priority order — a number between 100 and 4096, with *lower* numbers evaluated first. Once traffic matches a rule, evaluation stops; no lower-priority rule (a higher number) is ever checked after a match. Every new NSG automatically receives three real default inbound rules and three real default outbound rules (allowing traffic within the virtual network and from the Azure Load Balancer, denying everything else) at the lowest possible priority — you can't remove these defaults, but you can override them with your own, genuinely higher-priority custom rules.

A Real, Striking Coincidence: Azure's Own Instance Metadata Service

THE SAME REAL ADDRESS AS AWS

Azure exposes real per-VM instance metadata at `169.254.169.254` — the exact same real special address AWS's own IMDS (Chapter 3 of AWS Fundamentals) uses. Azure also uses a second real address, `168.63.129.16`, for platform services like DNS, licensing, and health monitoring. Both are real, Microsoft-owned virtualized addresses, and by default, traffic to them isn't subject to your own NSG rules at all — unless you deliberately target them with specific service tags (`AzurePlatformDNS`, `AzurePlatformIMDS`, `AzurePlatformLKM`) to override that default.

EC2 vs. Azure VM

PROPERTY	AWS EC2	AZURE VM
Real launch	2006, AWS's foundational service	June 2012, added over 2 years after Azure itself launched
Burstable family	T-series	B-series — real CPU credit model
Instance/network firewall	Two constructs — security groups (instance) + NACLs (subnet)	One construct — NSGs (subnet, NIC, or both)

PROPERTY	AWS EC2	AZURE VM
Spot eviction reasons	Capacity only	Capacity AND price-based
Metadata service address	169.254.169.254	169.254.169.254 (same address)

Hands-On Exercises

EXERCISE 1

A dev/test VM sits idle most of the day, with occasional short bursts of real CPU-intensive work. Recommend a real VM family for this workload, and explain how its own specific real mechanism fits the described usage pattern.

View solution

EXERCISE 2

A team sets a maximum Spot VM price well below the current real market rate for that VM size, expecting to only lose the VM if Azure runs out of capacity. Explain, in your own words, what real, second eviction reason they may have overlooked, and why.

View solution

EXERCISE 3

A team with prior AWS experience assumes they'll need two separate real constructs in Azure - one for instance-level filtering, one for subnet-level filtering - the way AWS uses security groups and NACLs. Explain, in your own words, why this assumption doesn't carry over correctly to Azure's own real NSG model.

View solution

CHAPTER 3 QUICK REFERENCE

- **Azure VMs** — launched June 2012, a real later addition to Azure, unlike AWS's own EC2-from-day-one foundation
- **VM families** — A (economical), B (burstable, real CPU credits), D (balanced default), F (compute), E/M (memory), L (storage), N (GPU), H (HPC)
- **Real pricing** — Pay-As-You-Go, Reserved (1/3-year), Spot (up to 90% off, with TWO real eviction reasons: capacity and price)
- **NSGs** — one real, stateful construct, applicable at subnet, NIC, or both levels — genuinely unlike AWS's split security-group/NACL model
- Real priority-based rule evaluation — lower number = higher priority, first match wins, defaults can't be removed but can be overridden
- Azure's own real IMDS uses the same real address as AWS's, 169.254.169.254, plus a second address (168.63.129.16) for platform services

CHAPTER 4: AZURE BLOB STORAGE

Azure Fundamentals

Chapter 4 · Azure Blob Storage

AWS Fundamentals Chapter 4 covered S3's own single, fixed 11-nines durability figure. Azure Blob Storage genuinely offers something S3 doesn't: a real, explicit choice of durability level, traded directly against cost and real geographic protection.

Storage Accounts & Containers

Blob data lives inside a real **storage account** — a top-level Azure resource (belonging to one Resource Group, Chapter 1) — which holds one or more real **containers**, Azure's own direct equivalent of an S3 bucket. Individual files (**blobs**) live inside a container, addressed by a real key, just as an S3 object is.

Real Access Tiers

TIER	REAL AVAILABILITY	MIN. RETENTION	USE CASE
Hot	99.9%	None	Frequently accessed data — the real default
Cool	99%	30 days	Infrequently accessed, still needs millisecond retrieval
Cold	99%	90 days	Rarely accessed, still needs fast (millisecond) retrieval
Archive	Offline	180 days	Long-term retention; retrieval takes up to 15 real hours
Smart Tier	Varies	N/A	Automatically moves data between Hot/Cool/Cold based on real usage — Azure's own version of S3 Intelligent-Tiering

Moving a blob out of the Cool, Cold, or Archive tier before its own real minimum retention period elapses triggers a genuine, prorated early-deletion charge — a real, easy-to-miss cost trap when experimenting with tier changes.

A Real Choice of Durability

GENUINELY DIFFERENT FROM S3'S SINGLE FIGURE

S3 Standard offers one fixed real durability figure, 11 nines. Azure Blob Storage instead lets you choose your own real redundancy configuration, each with its own documented durability:

REDUNDANCY	REAL DURABILITY	WHAT IT PROTECTS AGAINST
LRS	11 nines	Drive/server/rack failure — one single datacenter only
ZRS	12 nines	An entire datacenter/zone outage — requires a region that supports Availability Zones (Chapter 1's own real gotcha)
GRS / RA-GRS	16 nines	A full regional outage — asynchronously replicated to a paired secondary region
GZRS / RA-GZRS	16 nines	Both a zone outage AND a regional outage together

A genuinely direct, real consequence of Chapter 1's own AZ-support gotcha: ZRS and GZRS can only be used in Azure regions that actually support Availability Zones. In a region without AZ support, LRS or GRS/RA-GRS are the only real options available.

SAS Tokens

A **Shared Access Signature (SAS)** is a real, genuinely distinctive Azure Storage concept: a signed URL granting scoped, time-limited access to specific storage resources, without the requester needing an Entra ID identity (Chapter 2) at all. A SAS token can be scoped down to one specific blob, a whole container, or an entire storage account, with real, independently configurable permissions (read, write, delete) and a real expiration time.

A Real, Recent Case Study: The 2023 Microsoft AI Research Leak

WHAT ACTUALLY HAPPENED

> In June 2023, Wiz Research discovered a real, publicly accessible Microsoft Azure Storage account exposing roughly 38 terabytes of private internal data — disk backups from two employees' own workstations, secrets, private keys, passwords, and over 30,000 internal Microsoft Teams messages from 359 employees. The real cause: a SAS token, originally created to share an open-source AI training dataset, had been misconfigured to grant "full control" over the entire storage account rather than the specific intended files — and the token itself was valid until 2051, effectively never expiring, with no built-in Azure monitoring in place to flag its creation or ongoing use. Microsoft invalidated the token within two days of Wiz's real disclosure (24 June 2023) and publicly disclosed the incident on 18 September 2023.

This is a real, genuinely distinctive case study compared to Chapter 2 (AWS's Capital One breach) and AWS Fundamentals Chapter 4 (Deep Root Analytics) — the root cause here wasn't a public bucket left open, but a real, overscoped SAS token: correctly using SAS tokens means scoping each one narrowly to exactly the resource it needs, setting a genuinely real expiration date, and monitoring for their creation and use.

Lifecycle Management

A real, rule-based lifecycle management policy automatically moves blobs between access tiers, or deletes them entirely, based on real, defined conditions — for example, "move to Cool after 30 days, Archive after 180, delete after 3 years" — the same real automation concept as an S3 lifecycle policy (AWS Fundamentals Chapter 4).

Static Website Hosting

A storage account can serve its own blobs directly as a real static website — `index.html`, CSS, and JavaScript served straight from Blob Storage, with no web server to manage — the same real capability S3 offers.

S3 vs. Azure Blob Storage

PROPERTY	AWS S3	AZURE BLOB STORAGE
Container equivalent	Bucket	Container (inside a Storage Account)

PROPERTY	AWS S3	AZURE BLOB STORAGE
Durability	Fixed at 11 nines (Standard)	A real choice — 11, 12, or 16 nines, by redundancy configuration
Scoped, time-limited access	Pre-signed URLs / IAM policies	SAS tokens
Coldest real tier	Glacier Deep Archive	Archive (up to 15-hour retrieval)

Hands-On Exercises

EXERCISE 1

A company needs to protect a dataset against both a single zone outage AND a full regional disaster, and the target region does support Availability Zones. Recommend a real redundancy configuration, and explain why the alternatives fall short.

View solution

EXERCISE 2

Explain, in your own words, the specific real mistake that caused the 2023 Microsoft AI research leak - not simply "a token leaked," but what was actually misconfigured about the token itself.

View solution

EXERCISE 3

A blob is moved from the Cool tier to the Hot tier after only 12 days. Explain, in your own words, whether this triggers a real, documented cost consequence, and why.

View solution

CHAPTER 4 QUICK REFERENCE

- **Storage Account → Container → Blob** — Azure's real equivalent of S3's account/bucket/object structure
- **Access tiers** — Hot, Cool (30-day min), Cold (90-day min), Archive (180-day min, up to 15hr retrieval), Smart Tier
- **Real redundancy choice** — LRS (11 nines), ZRS (12 nines, needs AZ-supporting region), GRS/RA-GRS & GZRS/RA-GZRS (16 nines)
- **SAS tokens** — scoped, time-limited access without a full Entra ID identity; scope and expiration matter enormously
- Real case study: the 2023 Microsoft AI research leak (38TB exposed) — an overscoped, effectively-never-expiring SAS token, not a public bucket
- **Lifecycle management** and **static website hosting** — the same real capabilities as S3

CHAPTER 5: AZURE VIRTUAL NETWORK

Azure Fundamentals

Chapter 5 · Azure Virtual Network

AWS Fundamentals Chapter 5 built a VPC around subnets each tied to exactly one Availability Zone. Azure's own Virtual Network (VNet) works on a genuinely different, more flexible real principle — and this chapter's own biggest structural difference from AWS shows up in the very first section.

VNets Are Free

Unlike almost everything else in this course, there is real, documented zero charge for the Virtual Network resource itself — standard charges still apply to the VMs, gateways, and other resources running inside it, but the VNet and its own subnets cost nothing on their own.

A Real, Major Structural Difference: Subnets Span Every AZ

> **GENUINELY UNLIKE AWS**
>
> In AWS, a subnet is bound to exactly one specific Availability Zone (Chapter 5 of AWS Fundamentals) — placing an instance in an AZ means choosing the matching subnet. Azure works fundamentally differently: **an Azure Virtual Network, and every subnet inside it, spans all Availability Zones in the region at once**. You don't need separate subnets per zone to place a VM into a specific zone — zone placement and subnet choice are two entirely independent decisions in Azure, genuinely unlike AWS where they're the same decision.

This directly changes how a real multi-zone architecture is designed: rather than creating one subnet per AZ the way an AWS VPC requires, an Azure deployment typically uses a single subnet and specifies each individual VM's own zone separately, at creation time.

Real Reserved Addresses

Azure reserves the same real five addresses per subnet as AWS does (Chapter 5 of AWS Fundamentals) — the first four and the last one: the network address, the default gateway, two addresses reserved to map Azure's own DNS into the virtual network space, and the broadcast address. A genuine similarity, not a difference, between the two providers here.

Route Tables

Azure routes traffic between subnets, peered VNets, on-premises networks, and the internet automatically by default, using real system routes. You can override this default behavior with a **user-defined route (UDR)** — a real, custom route table entry directing specific traffic (e.g. `0.0.0.0/0`) to a different next hop, such as a NAT Gateway or a network virtual appliance.

NAT Gateway

Azure's own real NAT Gateway attaches directly to a subnet and immediately provides outbound-only internet connectivity to every private resource in it — genuinely simpler to set up than AWS's own equivalent, since it requires no manual route table configuration at all once attached. Exactly like AWS's own NAT Gateway, Azure's version allows only response traffic back in; nothing outside the virtual network can initiate a new inbound connection through it.

> **A REAL, RECENT PLATFORM CHANGE**
>
> As of 31 March 2026, newly created Azure virtual networks default to **private subnets with no default outbound internet access at all** — a real, deliberate security-hardening change from Azure's own earlier, more permissive default. An explicit outbound method (a NAT Gateway, a load balancer's own outbound rules, or an instance-level public IP) is now required for any outbound internet connectivity at all, matching AWS's own long-standing more restrictive, opt-in approach far more closely than Azure's older default ever did.

VNet Peering

VNet peering connects two virtual networks directly, letting resources in either one communicate as though they shared a single network — the peered VNets can sit in the same region or in genuinely different regions.

Application Security Groups

An **Application Security Group (ASG)** lets you group VMs logically — by application role rather than by IP address — and then reference that whole group as the source or destination in an NSG rule (Chapter 3). Instead of hardcoding a specific set of IP addresses into a security rule and updating it every time a VM is added or removed, you add or remove VMs from the ASG itself, and every NSG rule referencing that ASG updates automatically.

VPC vs. VNet

PROPERTY	AWS VPC	AZURE VNET
Subnet-to-AZ relationship	One subnet = one specific AZ	A subnet spans every AZ in the region
Reserved addresses per subnet	5	5 (same real reservation)
NAT setup	Requires a manual route table entry	No route table config needed once attached
Grouping resources for firewall rules	Security group membership directly	Application Security Groups (a separate, real logical layer)
Base resource cost	No charge for the VPC itself	No charge for the VNet itself

Hands-On Exercises

EXERCISE 1

A team with prior AWS experience creates three separate subnets in a new Azure VNet, expecting to place one VM into each of three different Availability Zones - one subnet per zone. Explain, in your own words, why this approach reflects an AWS assumption that doesn't apply to Azure's own real model.

View solution

EXERCISE 2

A team creates a new Azure VNet after 31 March 2026 and deploys a VM into it, expecting it to reach the internet for OS updates without any additional configuration. Explain, in your own words, why this expectation is likely wrong under Azure's own current real default, and what they'd need to add.

View solution

EXERCISE 3

A team frequently adds and removes VMs from a specific application tier, and wants their NSG rules to automatically apply to whichever VMs currently belong to that tier, without manually editing IP addresses each time. Explain, in your own words, how Application Security Groups solve this specific problem.

View solution

CHAPTER 5 QUICK REFERENCE

- **VNets are free** — no charge for the resource itself, only for what runs inside it
- Real major structural difference: **a subnet spans every AZ in the region** — unlike AWS's one-subnet-per-AZ binding; zone placement and subnet choice are independent decisions
- Real similarity: Azure reserves the same 5 addresses per subnet as AWS does
- **NAT Gateway** — attaches directly to a subnet, no route table config needed; real default outbound access was removed for new VNets as of 31 March 2026
- **VNet peering** — connects two VNets, same or different regions
- **Application Security Groups** — group VMs logically for NSG rules, avoiding hardcoded IP address management

CHAPTER 6: AZURE SQL DATABASE & COSMOS DB

Azure Fundamentals

Chapter 6 · Azure SQL Database & Cosmos DB

AWS Fundamentals Chapter 6 covered RDS and DynamoDB — a relational option and a NoSQL option, each fairly focused in scope. Azure's own two flagship database services genuinely diverge from that pattern in real, specific ways: Azure SQL Database offers a real, explicit choice of billing philosophy, and Cosmos DB offers real capabilities DynamoDB simply doesn't.

Azure SQL Database: Two Real Purchasing Models

Azure SQL Database offers a real, deliberate choice between two genuinely different ways of buying compute:

- **DTU-based** — a real, blended, single unit (a Database Transaction Unit) bundling CPU, memory, reads, and writes together into simple, preconfigured tiers (Basic, Standard, Premium). Genuinely simpler, with fixed monthly bundles, but no independent control over any one resource.
- **vCore-based** (Microsoft's own recommended default) — independently choose compute (vCores), memory, and storage, across three real service tiers: **General Purpose**, **Business Critical**, and **Hyperscale**. Also offers a real **serverless** compute option, which automatically pauses (billing only for storage) during genuine inactivity, and automatically resumes when activity returns.

> **A REAL, SPECIFIC COST DETAIL**
>
> The Business Critical tier automatically provisions three additional real replicas for higher availability and lower-latency local SSD storage — as a direct, real consequence, it costs roughly 2.7 times more than the equivalent General Purpose tier. This isn't an arbitrary price difference; it's the real, documented cost of the extra infrastructure Business Critical actually provisions on your behalf.

Cosmos DB: A Real 2017 Launch

Azure Cosmos DB released in 2017 as a real, globally distributed, multi-model database service — genuinely more ambitious in scope from the outset than DynamoDB's own single, focused key-value/document model (AWS Fundamentals Chapter 6).

A Real, Genuine Differentiator: Multi-Model APIs

Cosmos DB exposes the same underlying data through real, multiple wire-compatible APIs, letting an application use whichever query language it already knows:

API	REAL COMPATIBILITY
SQL (Core) API	Cosmos DB's own native, JSON-friendly query language
MongoDB API	Wire protocol version 6, server version 3.6
Cassandra API	Cassandra Query Language (CQL) wire protocol v4
Gremlin API	Graph queries, Gremlin specification v3.2
Table API	Azure Table Storage compatibility
etcd API	etcd wire protocol v3

DynamoDB has no real equivalent to this — it exposes exactly one, DynamoDB-specific API. An application already built against MongoDB or Cassandra can move to Cosmos DB with far less real rework than migrating to DynamoDB would require.

A Real, Genuine Differentiator: Five Consistency Levels

AWS Fundamentals Chapter 6 covered DynamoDB's own simple, two-option consistency choice — strongly consistent or eventually consistent. Cosmos DB offers a genuinely richer, real five-level spectrum instead:

- **Strong** — every read reflects the latest globally committed write.
- **Bounded Staleness** — reads never lag beyond a real, configured number of versions or a specified time window.
- **Session** (the real default) — guarantees a client always reads its own prior writes ("read-your-own-writes"), even if other clients might see slightly stale data.

- **Consistent Prefix** — reads never see out-of-order writes, though they may lag behind the latest.
- **Eventual** — no ordering guarantee at all; replicas converge eventually.

> **A REAL, MEANINGFUL MIDDLE GROUND**
>
> Cosmos DB's real default, Session consistency, solves a genuine, common problem neither of DynamoDB's own two options directly addresses as cleanly: guaranteeing a single client always sees its own writes immediately, without paying the real, full cost of Strong consistency for every reader globally.

RDS/DynamoDB vs. Azure SQL/Cosmos DB

PROPERTY	AWS (RDS / DYNAMODB)	AZURE (SQL DB / COSMOS DB)
Relational billing choice	Instance-size-based only	A real choice — DTU (bundled) or vCore (independent)
NoSQL API surface	One, DynamoDB-specific API	Six real wire-compatible APIs (SQL, MongoDB, Cassandra, Gremlin, Table, etcd)
Consistency options	Strong or Eventual (2 choices)	Strong, Bounded Staleness, Session, Consistent Prefix, or Eventual (5 choices)
Real default consistency	Eventual	Session

Hands-On Exercises

EXERCISE 1

A team wants the highest real availability and lowest latency for a critical Azure SQL Database workload, and is comparing Business Critical against General Purpose. Explain, in your own words, why Business Critical costs roughly 2.7 times more, rather than treating it as an arbitrary price markup.

View solution

EXERCISE 2

A team has an existing application built against the MongoDB API and wants to migrate to a managed cloud database with minimal code changes. Explain, in your own words, why Cosmos DB is a genuinely better real fit here than DynamoDB.

View solution

EXERCISE 3

A client needs to reliably read back its own just-written data, but the team doesn't want to pay the real cost of Strong consistency for every reader globally. Explain, in your own words, why Cosmos DB's real Session consistency level (the default) fits this need better than either of DynamoDB's own two consistency options would.

View solution

CHAPTER 6 QUICK REFERENCE

- **Azure SQL Database** — real DTU (bundled) vs. vCore (independent, with a real serverless auto-pause option) purchasing models
- Real cost detail: Business Critical costs ~2.7x General Purpose, due to three real automatically-provisioned extra replicas
- **Cosmos DB** — released 2017, real multi-model support (SQL, MongoDB, Cassandra, Gremlin, Table, etcd APIs)
- **Five real consistency levels** — Strong, Bounded Staleness, Session (default), Consistent Prefix, Eventual — genuinely richer than DynamoDB's two-option model
- Session consistency's real "read-your-own-writes" guarantee fills a genuine gap between DynamoDB's own Strong and Eventual options

CHAPTER 7: AZURE FUNCTIONS & SERVERLESS COMPUTING

Azure Fundamentals

Chapter 7 · Azure Functions & Serverless Computing

Azure Functions became generally available on 15 November 2016 — a real, striking near-exact two-year echo of AWS Lambda's own real launch on 13 November 2014 (AWS Fundamentals Chapter 7). The two services solve the same real problem, but genuinely diverge on execution limits and how a function connects to other data.

Real Hosting Plans

Azure Functions offers several genuinely different real hosting plans, each with its own execution model:

- **Flex Consumption** — the real current recommended default for serverless functions: pay-as-you-go, scales dynamically (up to 1,000 instances), with optional always-ready instances to reduce cold starts.
- **Consumption** — the original, now-legacy plan (Linux support retiring 30 September 2028); still pay-as-you-go, but with real, tighter limits than Flex Consumption offers.
- **Premium** — real prewarmed, always-warm instances, avoiding cold starts entirely, with more CPU/memory options and longer real allowed execution times.
- **Dedicated (App Service)** — functions run continuously on provisioned instances; cold start "isn't really an issue" at all, per Microsoft's own real documentation, since the host is always running.

A Real, Genuine Difference From Lambda: No Universal Hard Ceiling

GENUINELY DIFFERENT EXECUTION LIMITS

AWS Lambda enforces a real, fixed 15-minute maximum execution time, with no exceptions, regardless of configuration (AWS Fundamentals Chapter 7). Azure Functions works differently depending on the plan: the legacy Consumption plan has a real, tighter 5-minute default / 10-minute maximum timeout — actually *shorter* than Lambda's own ceiling — but Flex Consumption, Premium, and Dedicated all offer real, effectively **unbounded** execution time (with a 60-minute grace period

during scale-in events, and 10 minutes during platform updates). A genuinely long-running job that would need to be redesigned or split up for Lambda can, on the right Azure Functions plan, simply keep running.

A Real, Easy-to-Miss HTTP Gotcha

230 SECONDS, REGARDLESS OF YOUR OWN TIMEOUT SETTING

Even on a plan with an unbounded function timeout, an HTTP-triggered function has a real, separate, hard ceiling: 230 seconds to respond to any single request — a limit imposed by the default idle timeout of the underlying Azure Load Balancer itself, not by your own function's own configured timeout at all. A function's own `functionTimeout` setting genuinely cannot override this. For longer real HTTP-triggered processing, Microsoft's own real recommendation is the Durable Functions async pattern — return an immediate response and process the real work separately.

Triggers & Bindings

Azure Functions has a real, distinctive, declarative concept AWS Lambda doesn't offer in the same form: every function has exactly one real **trigger** (itself a special kind of input binding, defining what invokes the function and optionally passing in data), plus any number of optional real **input bindings** (data flowing in) and **output bindings** (data flowing out) — configured declaratively, rather than by writing manual SDK client calls inside the function's own code.

```
// function.json — an HTTP trigger with a Queue Storage output binding
{
  "bindings": [
    {
      "type": "httpTrigger",
      "direction": "in",
      "name": "req",
      "methods": ["get", "post"]
    },
    {
      "type": "queue",
      "direction": "out",
      "name": "msg",
```

```
      "queueName": "outqueue"
    }
  ]
}
```

The function's own code never has to construct a Storage SDK client, authenticate it, or manage a connection — it simply receives the incoming request via the `req` parameter and returns a value that Azure automatically writes to the queue via the `msg` output binding. Functions genuinely support multiple bindings at once — a real function could, for example, read from a queue (input) and write to a database (output) in the same invocation.

> **BINDINGS ARE OPTIONAL, NOT MANDATORY**
>
> Azure Functions doesn't force this declarative model on you — you can still create an Azure SDK client directly in your own code and manage connections manually, exactly as a Lambda function typically does. Bindings are a real, genuine convenience layered on top, not a hard requirement.

Cold Starts

On the legacy Consumption plan (and Flex Consumption without always-ready instances configured), a function app can scale to zero when idle, and the next real invocation pays a genuine cold-start delay. Premium's own prewarmed instances, and Dedicated's own continuously-running host, both avoid this real cost entirely — the same underlying tradeoff Lambda's own Provisioned Concurrency feature addresses (AWS Fundamentals Chapter 7), just built into the choice of hosting plan itself in Azure's model.

API Management

Azure's own real equivalent of API Gateway (Chapter 7 of AWS Fundamentals) is **Azure API Management** — it sits in front of one or more Azure Functions (or any other backend), handling routing, authentication, rate limiting, and request/response transformation before traffic ever reaches your own function code.

Lambda vs. Azure Functions

PROPERTY	AWS LAMBDA	AZURE FUNCTIONS
Real launch	November 2014	November 2016
Max execution time	15 minutes, always, no exceptions	Unbounded on Flex Consumption/Premium/Dedicated; 10 min on legacy Consumption
HTTP response ceiling	Governed by the same 15-min limit	230 seconds, regardless of function timeout setting
Data connections	Manual SDK client calls, typically	Real declarative triggers/bindings, optionally
Avoiding cold starts	Provisioned Concurrency (a feature)	Premium/Dedicated plan choice, or always-ready instances

Hands-On Exercises

EXERCISE 1

A team needs a function that reliably runs for 25 minutes to process a large real batch job, and they're deciding between AWS Lambda and Azure Functions on the Flex Consumption plan. Explain, in your own words, why this job genuinely cannot run unmodified on Lambda, but could run on Azure Functions.

View solution

EXERCISE 2

A team sets their HTTP-triggered Azure Function's functionTimeout to 20 minutes, expecting a slow request to be allowed to run that long. Explain, in your own words, why a request might still fail well before 20 minutes, and what real, separate limit is actually responsible.

View solution

EXERCISE 3

Explain, in your own words, why a function using an input binding to read from a queue and an output binding to write to a database is considered more declarative than a Lambda function manually calling AWS SDK clients to do the same two things.

View solution

CHAPTER 7 QUICK REFERENCE

- **Azure Functions** — launched November 2016, real two-year echo of Lambda's own 2014 launch
- **Hosting plans** — Flex Consumption (current default), legacy Consumption (5/10 min), Premium (prewarmed, no cold starts), Dedicated (always running)
- Real major difference: Flex Consumption/Premium/Dedicated offer **unbounded** execution time — genuinely unlike Lambda's fixed 15-minute ceiling
- Real gotcha: HTTP-triggered functions face a separate, hard **230-second** response ceiling regardless of the function's own timeout setting
- **Triggers & bindings** — declarative, optional input/output connections, avoiding manual SDK client code
- **API Management** — Azure's real equivalent of API Gateway

CHAPTER 8: LOAD BALANCING & VIRTUAL MACHINE SCALE SETS

Azure Fundamentals

Chapter 8 · Load Balancing & Virtual Machine Scale Sets

AWS Fundamentals Chapter 8 covered three real load balancer types (Classic, ALB, NLB) plus Auto Scaling Groups. Azure genuinely splits this into more, more specialized real services — four distinct load balancing options, each solving a different real problem, plus Virtual Machine Scale Sets for the compute side.

Four Real Azure Load Balancing Services

SERVICE	LAYER	SCOPE	REAL USE CASE
Load Balancer	4 (TCP/UDP)	Regional (or cross-region)	High-performance, ultra-low-latency non-HTTP(S) traffic; zone redundant
Application Gateway	7 (primarily)	Regional	Path-based routing, TLS offload, WAF, for public traffic entering one region's own private network
Azure Front Door	7	**Global**	Global HTTP(S) load balancing, CDN, fast failover across regions
Traffic Manager	DNS-based	Global	Any traffic type via DNS routing, but real, genuinely slower failover (DNS caching/TTL delays)

A Real, Useful Technical Distinction: Passthrough vs. Terminating

TWO GENUINELY DIFFERENT CONNECTION MODELS

Azure's own documentation draws a real, useful distinction: a **passthrough load balancer** (like Load Balancer) lets a client establish a connection directly with the specific backend server the load balancer's own algorithm selected. A **terminating load balancer** (like Application Gateway) has the client connect to the load balancer itself, which then initiates a genuinely separate connection to the backend on the client's behalf. This maps conceptually onto AWS Fundamentals Chapter 8's own NLB (passthrough-like) vs. ALB (proxy-based, terminating) distinction — the same underlying real trade-off, just named more explicitly in Azure's own documentation.

Regional vs. Global — A Real, Explicit Decision

Every one of the four services above is real, explicitly categorized along two dimensions: **global vs. regional**, and **HTTP(S) vs. non-HTTP(S)**. A regional service (Load Balancer, Application Gateway) distributes traffic within one virtual network. A global service (Front Door, Traffic Manager) distributes traffic across multiple regions, clouds, or hybrid on-premises services through one real, unified control plane — genuinely closer to what a multi-Region AWS architecture would need to build manually on top of Route 53 and cross-region infrastructure.

Virtual Machine Scale Sets

Azure's own real equivalent of an Auto Scaling Group is the **Virtual Machine Scale Set (VMSS)** — a group of load-balanced, identically configured VM instances that automatically grows or shrinks in response to real demand or a defined schedule. Every instance is created from the same base OS image and configuration, exactly as an AWS ASG's own launch template ensures.

> **FREE, AGAIN**
>
> Continuing this course's own recurring real pattern (VNets in Chapters 1 and 5), there's genuinely no extra charge for the scale set service itself — you're billed only for the real compute, network, and storage resources it actually uses.

A real scale set supports up to 1,000 VM instances when using standard marketplace or custom images through Azure Compute Gallery — that real limit drops to 600 instances if you instead use a managed image directly.

> **A REAL, ONE-TIME DECISION**
>
> VMSS offers two real orchestration modes — **Uniform** and **Flexible** — and the mode is set when the scale set is created and genuinely cannot be changed afterward. Flexible mode offers higher availability guarantees by spreading instances across fault domains, and supports mixing VM types or Spot and on-demand instances together within the same scale set; Uniform mode is the older, simpler model. Choosing wrong at creation time means recreating the scale set entirely to switch.

> **A SCALE SET ALONE ISN'T ZONE PROTECTION**

> Microsoft's own real documentation states this plainly: "a scale set alone can't protect you against data center failures." Real Availability Zone distribution has to be explicitly configured on top of the scale set — and, per Chapter 1's own gotcha, this real protection is only available at all in regions that actually support Availability Zones in the first place.

Autoscale Rules

VMSS autoscale rules work the same real way as an AWS Auto Scaling Group's own target tracking policy (AWS Fundamentals Chapter 8) — a real, defined metric (commonly CPU utilization) drives capacity up or down automatically to hold a target range, adding instances as demand rises and removing them as it falls, to avoid paying for unnecessary capacity during quiet periods.

ALB/NLB vs. Azure's Four Services

PROPERTY	AWS	AZURE
Regional Layer 4	NLB	Load Balancer
Regional Layer 7	ALB	Application Gateway
Global HTTP(S) + CDN	CloudFront + Route 53, combined manually	Azure Front Door, one real integrated service
DNS-based global routing	Route 53 routing policies	Traffic Manager
Auto-scaling compute group	Auto Scaling Group	Virtual Machine Scale Set (free of extra charge)

Hands-On Exercises

EXERCISE 1

A public web application needs path-based routing (/api vs /web), TLS offload, and a web application firewall - but is deployed in only a single Azure region. Recommend a real service from this chapter,

and explain why Azure Front Door alone would be a genuinely poor fit despite also supporting Layer 7 features.

View solution

EXERCISE 2

A team creates a VM Scale Set using Uniform orchestration mode, and six months later decides they need Flexible mode's own ability to mix Spot and on-demand instances. Explain, in your own words, what real, concrete step this actually requires.

View solution

EXERCISE 3

A team deploys a Virtual Machine Scale Set in a region and assumes this alone genuinely protects them against an entire datacenter outage. Explain, in your own words, why this assumption is incorrect, and what real, additional step from this chapter and Chapter 1 is needed.

View solution

CHAPTER 8 QUICK REFERENCE

- **Load Balancer** (L4, regional, passthrough), **Application Gateway** (L7, regional, terminating/WAF), **Front Door** (L7, global, CDN), **Traffic Manager** (DNS-based, global)
- Real distinction: **passthrough** (client connects directly to the chosen backend) vs. **terminating** (client connects to the load balancer itself)
- **VMSS** — free of extra charge, up to 1,000 instances (600 with a managed image), auto-scales via real metric-based rules
- Real gotcha: orchestration mode (**Uniform** vs. **Flexible**) is set at creation and cannot be changed afterward
- Real gotcha: a scale set alone doesn't protect against a datacenter failure — explicit AZ configuration is still required (and only available in AZ-supporting regions, per Chapter 1)

CHAPTER 9: AZURE MONITOR & COST MANAGEMENT

Azure Fundamentals

Chapter 9 · Azure Monitor & Cost Management

AWS Fundamentals Chapter 9 covered CloudWatch's own genuine memory/disk-space monitoring gap, and the real, critical distinction between AWS Budgets alerting versus actually stopping spending. Azure's own equivalents share both real lessons — but Azure Monitor frames the first one through a genuinely richer, more explicit real model.

Azure Monitor's Real Four-Layer Model

Azure's own real documentation defines exactly four monitoring layers for a virtual machine, each with genuinely different real requirements:

LAYER	COLLECTED AUTOMATICALLY?	REAL EXAMPLE DATA
Virtual machine host	Yes — no agent needed	CPU utilization, whether the machine is running, configuration changes
Guest operating system	No — requires the Azure Monitor Agent	Memory usage, disk space, OS-level events
Workloads	No — requires the Azure Monitor Agent	Performance data and events from software running inside the VM
Application	No — requires Application Insights	Business application performance and traces

THE SAME REAL GAP CLOUDWATCH HAS, MORE EXPLICITLY FRAMED

Exactly like CloudWatch's own genuine memory/disk-space gap (AWS Fundamentals Chapter 9), the host layer alone gives you only basic real metrics — CPU utilization and whether the machine is running — with zero setup. Real memory and disk-space data live one layer deeper, in the guest operating system, and require deploying the Azure Monitor Agent before they exist as monitorable data at all.

A Real, Easy-to-Confuse Pair: Two Different Workspace Types

Azure Monitor's own real data platform uses two genuinely separate workspace types, despite their similar names: a **Log Analytics workspace** collects logs and traces, queried with Kusto Query Language (KQL); an **Azure Monitor workspace** collects Prometheus and OpenTelemetry metrics, queried with PromQL instead. They're real, separate resource types with different data stores — not two names for the same thing.

Alerts & Autoscale

Azure Monitor alerts watch a real metric or log query against a threshold and trigger a response once crossed — including real, AI-assisted dynamic thresholds that adapt automatically to a metric's own normal pattern, reducing false alarms compared to a single fixed number. **Autoscale** — Chapter 8's own VMSS scaling mechanism — is itself literally a feature of Azure Monitor, not a separate service: the same real metric data driving your alerts also drives your scale set's own capacity decisions.

Real Cost Management Tools

- **Cost Management + Billing** — Azure's own real cost analysis tool, breaking down spend by service, resource group, or tag, directly analogous to AWS's own Cost Explorer.
- **Budgets** — set a real, specific dollar threshold (based on actual or forecasted cost) and get notified when it's approached or crossed.

> **A REAL, EXPLICIT CONFIRMATION: BUDGETS ALERT, THEY DON'T STOP SPENDING**
>
> Azure's own real, documented language is direct and unambiguous: "Notifications are triggered when the budget thresholds are exceeded. Resources aren't affected, and your consumption isn't stopped." This is the exact same real lesson AWS Fundamentals Chapter 9 taught about AWS Budgets — crossing a threshold notifies; it never automatically halts anything.

A Real, Distinctive Azure Feature: Action Groups on a Budget

Azure genuinely offers a more direct, built-in path to real automated response than simply pairing a budget with a separate notification service: when creating or editing a budget, you can attach an **Action Group** directly — triggering a mobile push notification, a webhook, an Azure Function, or a Logic App the moment a threshold is crossed. This doesn't change the core lesson (the budget itself still only observes and notifies) — but it makes wiring up a real, automated response noticeably more direct than in AWS's own equivalent workflow, where connecting a Budget alert to an automated action (like a Lambda function) has to be assembled from separate pieces.

CloudWatch vs. Azure Monitor

PROPERTY	AWS (CLOUDWATCH / COST EXPLORER / BUDGETS)	AZURE (MONITOR / COST MANAGEMENT)
Automatic host metrics	CPU, network, disk I/O	CPU, running status (a real, narrower default set)
Memory/disk-space gap	Needs the CloudWatch Agent	Needs the Azure Monitor Agent (guest OS layer)
Log/metric query language	CloudWatch Logs Insights (one system)	KQL (Log Analytics) and PromQL (Azure Monitor workspace) — two separate systems
Budget behavior	Alerts only, never stops spending	Alerts only, never stops spending — real, identical behavior
Built-in automated response	Assembled via SNS + Lambda separately	A real, direct Action Group attachment on the budget itself

Hands-On Exercises

EXERCISE 1

A team assumes Azure Monitor is already tracking disk space usage on their VMs, but an alert never fires before a disk actually fills up. Using the real four-layer model, explain, in your own words, why this happened and what real step was missing.

View solution

EXERCISE 2

A team wants to query both their application's own log traces and Prometheus-based container metrics using the same query language and the same workspace. Explain, in your own words, why this expectation is incorrect under Azure Monitor's own real data platform.

View solution

EXERCISE 3

A team sets an Azure Budget at $10,000/month with no Action Group attached, and a compromised credential is later used to launch unauthorized compute. Explain, in your own words, why the Budget alone would not have prevented the unexpected charges, and how attaching an Action Group could have changed the outcome.

View solution

CHAPTER 9 QUICK REFERENCE

- **Four real monitoring layers** — host (automatic), guest OS/workloads (need the Azure Monitor Agent), application (needs Application Insights)
- Real gap: memory and disk-space data live at the guest OS layer, not automatically collected — the same real lesson as CloudWatch's own gotcha
- **Log Analytics workspace** (KQL, logs/traces) vs. **Azure Monitor workspace** (PromQL, Prometheus/OpenTelemetry metrics) — two separate real systems
- **Autoscale** is literally a feature of Azure Monitor, directly powering Chapter 8's own VMSS scaling
- Real, confirmed: Azure Budgets alert only — "Resources aren't affected, and your consumption isn't stopped"
- Real distinctive feature: an **Action Group** can be attached directly to a budget for a genuinely automated response

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

Azure Fundamentals

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

AWS Fundamentals Chapter 10 built a small photo-upload web application — metadata in a database, originals in object storage, an automated thumbnail job. This capstone deliberately rebuilds the exact same real application on Azure, using every one of this course's own nine chapters — letting the two capstones be compared side by side, service for service, on genuinely different real architecture.

Step 1 — Region & Availability Zone Planning

Chapter 1: before anything else, confirm the target Region actually supports Availability Zones — a real, necessary check that AWS Fundamentals' own capstone never needed, since every AWS Region always has them.

Step 2 — A Scoped Managed Identity

Chapter 2: a Managed Identity is attached to the web VM, with an RBAC role assignment scoped narrowly to the Resource Group holding the application's own Storage Account — Contributor, not Owner, since the VM needs to manage resources but never needs to grant access to others.

Step 3 — VNet & Subnet

Chapter 5: one VNet, one subnet — genuinely simpler than AWS Fundamentals' own capstone, which needed separate public and private subnets per AZ. Because an Azure subnet spans every AZ in the region at once, both the web VM and the database sit in the same subnet, with each resource's own zone chosen independently at creation time.

Step 4 — The Web VM

Chapter 3: a D-series VM (the real, balanced general-purpose default) runs the application. Its own NSG allows inbound traffic only from Application Gateway (Step 8) — never directly from the internet.

Step 5 — Blob Storage

Chapter 4: the application's photo container uses **ZRS** redundancy (real 12-nines durability, since the target region supports Availability Zones per Step 1), with versioning enabled. Any SAS tokens issued to the application are scoped narrowly to this one specific container, with a real, genuine expiration date — a direct, deliberate lesson learned from Chapter 4's own 2023 Microsoft AI research leak case study.

Step 6 — Azure SQL Database

Chapter 6: a vCore-based, General Purpose-tier Azure SQL Database stores photo metadata (uploader, timestamp, blob key) — General Purpose, not Business Critical, since this small application doesn't need the real ~2.7x cost of Business Critical's own extra replicas.

Step 7 — An Azure Function for Thumbnails

Chapter 7: a Blob Storage trigger fires an Azure Function on the Flex Consumption plan the moment a new photo lands, generating a thumbnail and writing it back via a Blob output binding — the function's own code never constructs a Storage SDK client at all, a real, direct contrast with AWS Fundamentals' own capstone Lambda function, which reads and writes S3 via manual SDK calls.

Step 8 — Application Gateway & a Virtual Machine Scale Set

Chapter 8: Application Gateway (Layer 7, regional — genuinely the right real choice here, since this is a single-region deployment, not Front Door's global scope) sits in front of a VM Scale Set running the web tier, with autoscale rules holding CPU near a target. The scale set's own orchestration mode — Flexible — is chosen deliberately and explicitly at creation, since that real decision can never be changed afterward.

Step 9 — Azure Monitor & a Budget With a Real Action Group

Chapter 9: the Azure Monitor Agent is deployed to the web VM specifically so its own real guest-OS-layer memory and disk-space metrics actually exist to alert on — closing the exact gap the chapter's own four-layer model warns about. A Budget is set at a real, planned monthly threshold, with an **Action Group** attached directly to it — a real, more automated response path than AWS Fundamentals' own capstone Budget had, which was left deliberately unattached to any automated action.

The Full Picture

STEP	CHAPTER	AWS FUNDAMENTALS' OWN EQUIVALENT
1	Ch.1 — Region & AZ-support planning	Region & multi-AZ planning
2	Ch.2 — Scoped Managed Identity	Scoped IAM role
3	Ch.5 — One VNet, one subnet (spans all AZs)	VPC, public/private subnets per AZ
4	Ch.3 — Web VM, scoped NSG	EC2 instances, scoped security group
5	Ch.4 — Blob Storage, ZRS, scoped SAS tokens	S3, Block Public Access, versioning
6	Ch.6 — Azure SQL Database, General Purpose	Multi-AZ RDS
7	Ch.7 — Azure Function, Blob bindings (no manual SDK)	Lambda, manual S3 SDK calls
8	Ch.8 — Application Gateway + VMSS, Flexible mode	ALB + Auto Scaling Group
9	Ch.9 — Azure Monitor Agent + Budget with an Action Group	CloudWatch + unattached Budget

What's Still Out of Scope, Honestly

Matching AWS Fundamentals' own capstone's honesty, this deployment deliberately stops short of several real, genuine next steps: no CI/CD deployment pipeline, no Azure Front Door or WAF, no Azure DNS configuration, no cross-region disaster recovery, and no

containerization (AKS) as a VM alternative. Each is a real, legitimate next step for a production system, named honestly rather than treated as solved here.

SAME APPLICATION, GENUINELY DIFFERENT ARCHITECTURE

Every one of this capstone's own real decisions traces back to a specific, concrete lesson from this course — not to a habit carried over unexamined from AWS. The VNet spanning every AZ (Ch.5), the declarative Function bindings (Ch.7), the ZRS-vs-LRS durability choice (Ch.4), and the Action-Group-attached Budget (Ch.9) are all real, deliberate choices this course's own nine chapters specifically prepared you to make.

THE SAME UNDERLYING SECURITY DISCIPLINE, DIFFERENT MECHANISMS

Least privilege (Ch.2's Managed Identity scope), private-by-default networking (Ch.5's real March 2026 default change), scoped access tokens (Ch.4's real SAS lesson), and honest monitoring (Ch.9) all show up together here — the same recurring discipline AWS Fundamentals' own capstone closed with, expressed through Azure's own genuinely different real mechanisms.

Closing the Course

From Chapter 1's real 2008 origin story and global infrastructure through this chapter's own working, multi-service architecture — every chapter covered one genuine piece of what it actually takes to run something real on Azure, and, deliberately, how that differs from the equivalent AWS approach at almost every single step. Compute, storage, networking, databases, serverless, scaling, and monitoring together — not any single service in isolation — are what make a real Azure deployment resilient and secure.

Hands-On Exercises

EXERCISE 1

Explain, in your own words, why Step 3 of this capstone needed only one subnet, while AWS Fundamentals' own capstone needed separate public and private subnets across multiple AZs to achieve the same real multi-zone resilience.

View solution

EXERCISE 2

Trace, step by step, every real point in this capstone's own architecture where least privilege is specifically applied, and explain what would go wrong at each point if it weren't - the same exercise AWS Fundamentals' own capstone asked, applied here to Azure's own real mechanisms.

View solution

EXERCISE 3

A teammate argues that attaching an Action Group to Step 9's own Budget means the budget can now directly stop unexpected spending. Using Chapter 9's own real distinction, explain why this claim is still incorrect, even with the Action Group attached.

View solution

COURSE COMPLETE — AZURE FUNDAMENTALS

- 10/10 chapters — from Azure's real 2008 origin (Project Red Dog) through a full, working multi-service architecture
- Every chapter's own real service applied together in this capstone, deliberately rebuilding AWS Fundamentals' own capstone application to highlight genuine architectural differences at every step
- A recurring real thread: least privilege, private-by-default networking, and honest monitoring — Azure's own mechanisms, the same underlying discipline
- Real, honest next steps named rather than glossed over: CI/CD, Front Door/WAF, DNS, cross-region DR, containerization
- Last in the Cloud Platform Specifics arc: GCP Fundamentals (`gcp1`), completing the deliberate three-way, chapter-for-chapter parallel structure