

AWS A Visual Guide

Essential Cloud Foundations for New Systems Engineers

Reiner Braun

Sample Book — Chapters 1 to 2

Introduction

This book is designed for readers who are taking their first steps with AWS, including systems engineers who want to build the right foundation from the start.

Rather than asking you to memorize more than you need, the book focuses on the essential concepts that will support your future study and day-to-day work. More advanced topics are included only when they are likely to become useful to many readers, and even then they are introduced only far enough to explain their purpose, context, and overall role.

Processes, relationships, and structures that can be difficult to picture from text alone are supported by diagrams. By moving between explanations, visuals, and examples, you can build a clearer and more practical understanding of the subject.

Product names, service names, logos, icons, and other identifying marks mentioned in this book are trademarks or registered trademarks of their respective owners. This book is not an official publication, certified training resource, or endorsed learning material of any company or organization.

How to Use This Book

Read the chapters in order when possible. Each chapter builds on concepts introduced earlier, so following the sequence will help you develop the necessary background knowledge step by step instead of learning isolated service names.

Before reading a chapter, review its **What You'll Be Able to Do** section. The outcomes identify the practical knowledge you should expect to gain and help you distinguish the chapter's essential foundation from supporting detail.

Use the diagrams to clarify relationships, processes, boundaries, and structures that are difficult to hold in mind from prose alone. The diagrams support the explanations; they are not substitutes for the surrounding text.

Try the commands, code, and hands-on examples yourself whenever you have an appropriate test environment and permission to use it. Confirm the prerequisites, predict the expected result, run the smallest safe example, and then compare the observed result with your prediction.

At the end of each chapter, use the **Chapter Summary** and **Review Questions** to confirm that you understand and can recall the fundamentals. Return to the relevant section when you cannot explain an answer in your own words.

Advanced-topic overviews do not need to be memorized at this stage. Treat them as signposts: they show what exists, why it matters, and which subjects to study more deeply when a project or role requires them.

Table of Contents

[Introduction](#)

[How to Use This Book](#)

[Main Text](#)

[Scope of this guide](#)

[Part I — The AWS Mental Model and Safe Access](#)

[Chapter 1 — AWS as Programmable Infrastructure](#)

[What You'll Be Able to Do](#)

[From purchased hardware to requested capacity](#)

[The service responsibility spectrum](#)

[Six qualities that must not be confused](#)

[Consumption-based cost changes engineering behavior](#)

[The Well-Architected questions](#)

[The workload lifecycle](#)

[Practical comparison: one web application, three execution models](#)

[One EC2 instance](#)

[An ECS service](#)

[A Lambda function](#)

[Misconceptions that cause significant errors](#)

[Chapter Summary](#)

[Review Questions](#)

[Chapter 2 — Accounts, Regions, Availability Zones, and Resource Scope](#)

[What You'll Be Able to Do](#)

[The AWS account is an isolation and ownership boundary](#)

[Regions and Availability Zones](#)

[AZ names and AZ IDs](#)

[Resource scope: global, Regional, and zonal](#)

[Edge infrastructure is not a second application Region](#)

[Resource identifiers and ARNs](#)

[Choosing a Region](#)

[Quotas and capacity constraints](#)

[Worked scope trace](#)

[Frequent scope mistakes](#)

[Chapter Summary](#)

[Review Questions](#)

Main Text

Scope of this guide

AWS is best understood as programmable infrastructure with explicit boundaries, permissions, failure domains, and operating responsibilities. A dependable AWS system combines sound resource design with repeatable change, layered security, useful evidence, tested recovery, and clear ownership. This guide develops that complete foundation for systems engineers.

The chapters begin with the AWS mental model, resource scope, identity, safe access, networking, compute, storage, databases, containers, and serverless platforms. They then extend those foundations into asynchronous integration, defense in depth, observability, fleet operations, infrastructure as code, reliability, performance and cost control, multi-account governance, hybrid connectivity, incident response, and architecture decisions. The appendices provide concise systems and service-selection references.

A small web system with a public HTTPS entry path, an application tier, persistent data, background processing, and centralized operations appears throughout the guide. It is a concrete learning example, not a universal reference architecture. Each design choice must ultimately follow the workload's requirements, risks, operating model, and constraints.

Part I — The AWS Mental Model and Safe Access

Chapter 1 — AWS as Programmable Infrastructure

The central idea is that AWS provides on-demand building blocks through APIs, while the customer remains responsible for using those building blocks correctly. Learning AWS is therefore less about memorizing product names and more about understanding responsibility, scope, failure, automation, and trade-offs.

What You'll Be Able to Do

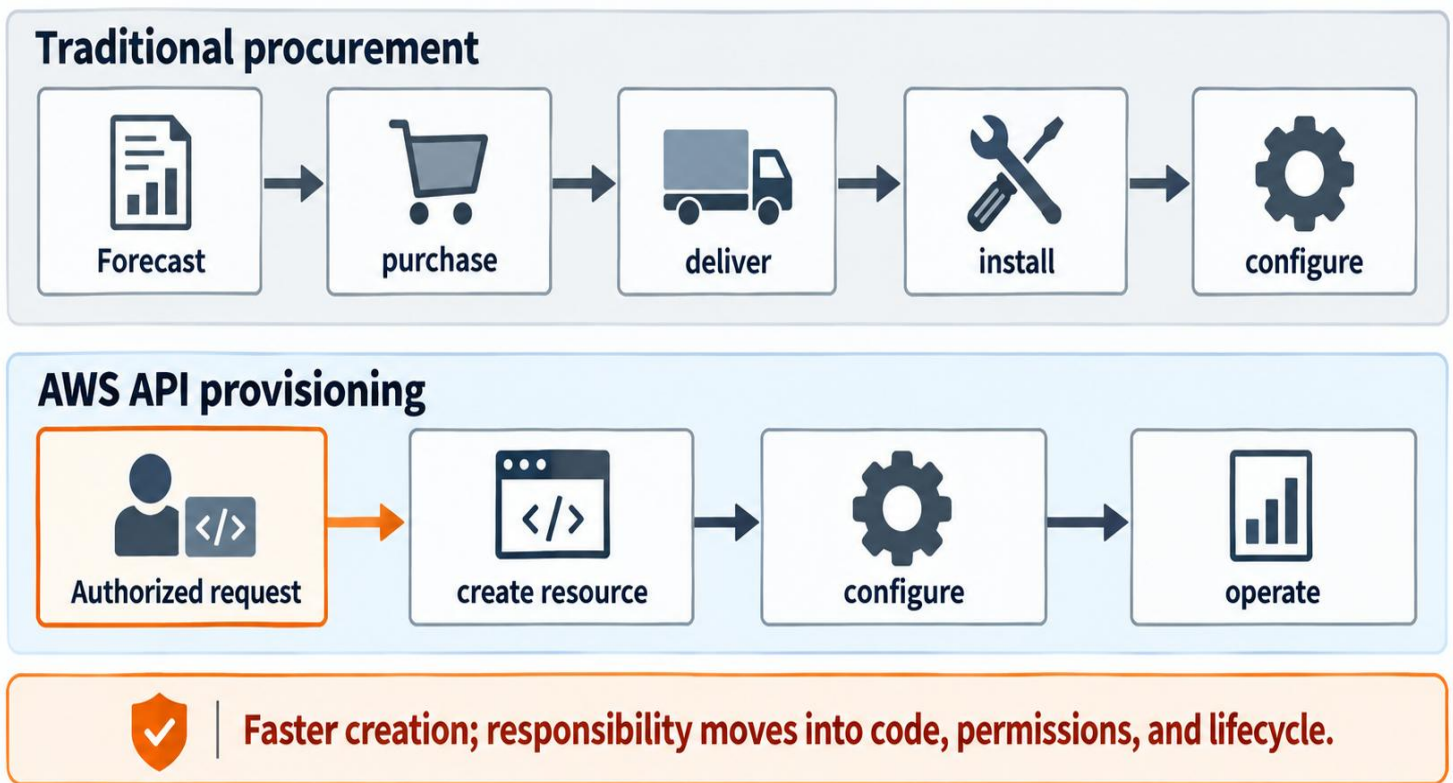
- Explain why AWS is best understood as API-driven, programmable infrastructure rather than merely remote hardware.
- Compare the customer and AWS operating responsibilities for EC2, ECS on Fargate, and Lambda.
- Distinguish scalability, elasticity, availability, durability, resilience, and performance.
- Relate workload lifecycle, cost, and the Well-Architected pillars to everyday engineering decisions.

From purchased hardware to requested capacity

In a traditional data center, acquiring capacity often begins with forecasting, purchasing, delivery, installation, cabling, and operating-system setup. In AWS, an authorized request can create a virtual server, a database, a network, or storage in minutes. The request may come from the AWS Management Console, a command-line tool, application code, or an infrastructure template, but it ultimately becomes an **API request**, which is a structured request to an AWS service to read or change a resource.

Chapter 1: AWS as Programmable Infrastructure

1-1 From physical procurement to an API-created resource



001

Figure 1-001: From physical procurement to an API-created resource

This changes the engineer's work in several important ways.

- **Provisioning becomes repeatable.** The same declared configuration can create similar environments many times.

- **Capacity becomes adjustable.** Resources can be added or removed as demand changes.
- **Configuration becomes part of the system.** Permissions, routes, tags, health checks, and service settings are operational assets, not incidental console choices.
- **Mistakes can spread quickly.** A script can create or modify many resources just as easily as it can create one.
- **Cost follows consumption.** Rapid creation must be paired with ownership, cleanup, and monitoring.

A useful analogy is a programmable building. In a conventional building, adding a room requires a construction project. In a programmable building, an authorized instruction can add a room immediately. That speed is valuable, but the instruction must still specify who can enter, how power and fire exits work, who pays for it, and when the room should be removed.

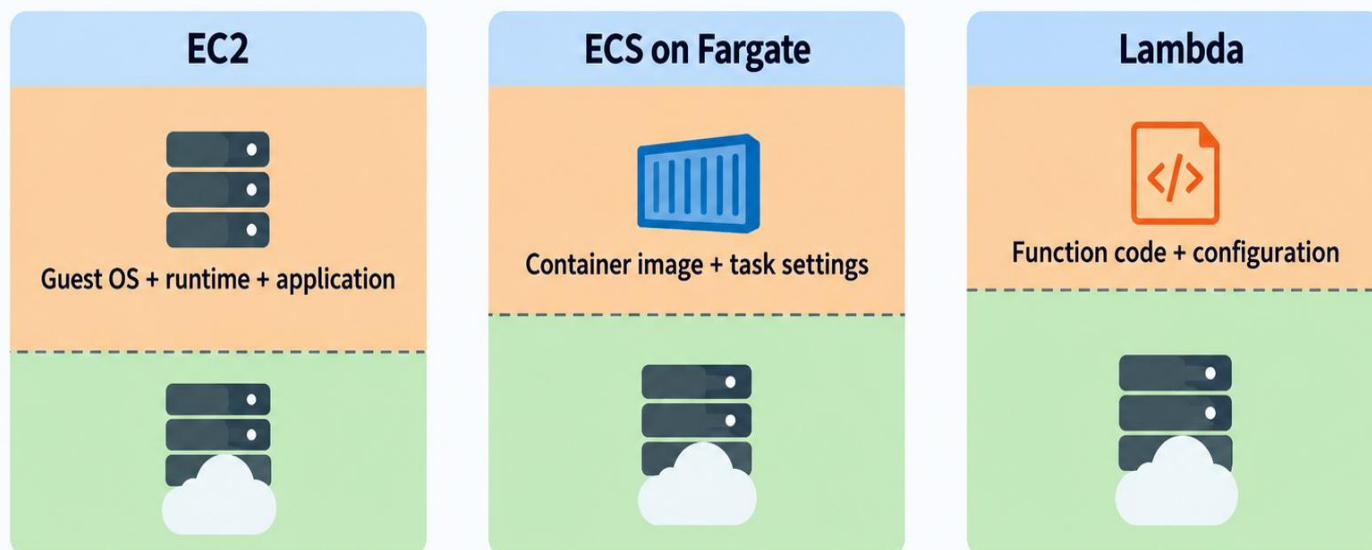
The service responsibility spectrum

AWS services differ mainly in how much of the technology stack AWS operates for you. **Managed does not mean responsibility-free; it means that the responsibility boundary moves.**

Chapter 1: AWS as Programmable Infrastructure

1-2 Responsibility spectrum from EC2 to Fargate to Lambda

AWS operates more layers →



Customer always owns application, data, access, and availability design.

002

Figure 1-002: Responsibility spectrum from EC2 to Fargate to Lambda

Consider three ways to run application code.

Execution model	AWS primarily operates	The customer still operates or decides
Amazon EC2 virtual machine	Physical facilities, hardware, network fabric, and virtualization layer	Guest operating system, patching, runtime, application, capacity, access, data, and most availability design
Amazon ECS on AWS Fargate	The preceding layers plus the host fleet and container runtime infrastructure	Container image, task sizing, application, IAM roles, networking, scaling policy, data, and deployment behavior
AWS Lambda function	The preceding layers plus execution-environment provisioning and automatic function scaling	Function code, dependencies, permissions, timeout, concurrency controls, event handling, data, and downstream protection

The difference is similar to choosing among an empty rented building, a serviced office, and a fully managed workspace.

- In the empty building, you control almost everything and must maintain almost everything.
- In the serviced office, the provider manages the building and common facilities, while you manage your team and its work area.
- In the managed workspace, even more operations are handled for you, but you still decide who may enter, what work is performed, and how information is protected.

No model is universally superior. EC2 is often appropriate when an application needs operating-system control, unusual software, or broad compatibility. Containers are useful when a team wants a portable, versioned application package. Lambda is useful for event-driven or request-driven work that fits its execution model. The correct question is not “Which service is most modern?” but “**Which responsibility model best fits the workload and the team?**”

Six qualities that must not be confused

Several common terms describe different system properties.

Quality	Practical meaning	Example question
Scalability	The ability to handle more work by adding or enlarging resources	Can the application handle ten times as many requests?
Elasticity	The ability to add and remove capacity as demand changes	Can capacity grow for a peak and shrink afterward?
Availability	The proportion of time a service is usable	Can users reach the application during an instance failure?
Durability	The likelihood that stored data remains intact over time	Will an uploaded object still exist after a hardware failure?
Resilience	The ability to withstand or recover from disruption	Can the system continue or recover when a dependency fails?
Performance	The speed and throughput with which work is completed	Does the API meet its latency target under expected load?

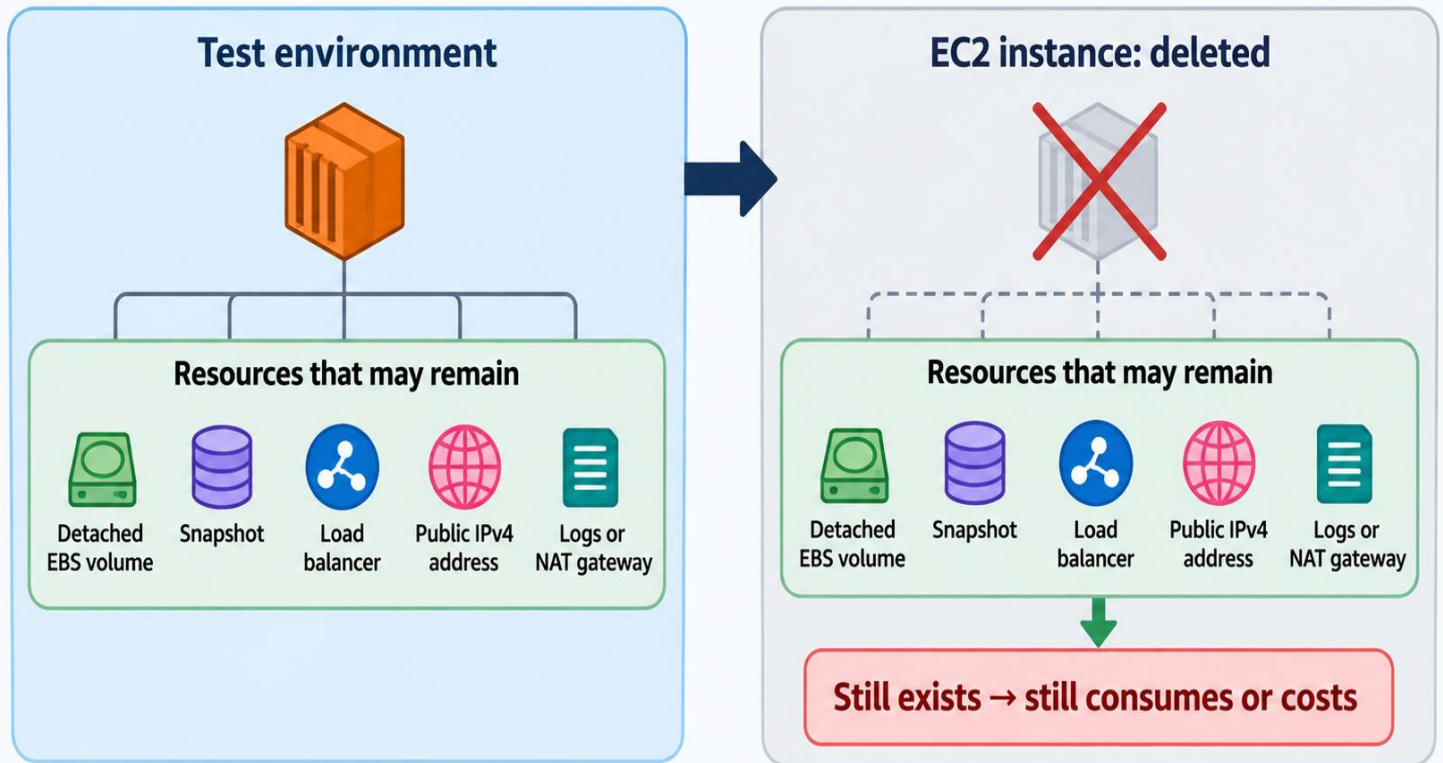
A system can score well in one quality and poorly in another. A single fast server may have strong performance but weak availability. Two servers may improve availability but do nothing to preserve accidentally deleted data. A backup can improve recoverability but does not keep an application online while restoration is in progress.

High availability is not the same as backup. High availability reduces interruption by keeping usable capacity available. Backup preserves a recoverable copy of data or configuration. Most important systems need both.

Consumption-based cost changes engineering behavior

AWS commonly charges according to dimensions such as resource runtime, provisioned capacity, storage amount, requests, data processing, and data transfer. The exact price is not the foundational concept; the important point is that **architecture decisions create continuing consumption**.

Chapter 1: AWS as Programmable Infrastructure 1-4 The cost tail left after a test server is deleted



003

Figure 1-003: The cost tail left after a test server is deleted

For example, deleting a test EC2 instance may not remove its detached storage volume, snapshots, load balancer, public IPv4 address, log retention, or NAT gateway. A beginner may see that “the server is gone” while the system continues to generate charges.

Consumption-based pricing creates two responsibilities from the beginning:

1. Every resource should have a clear owner and purpose.
2. Every temporary resource should have a cleanup decision.

Detailed cost analysis belongs later, but engineers should treat cost as a system signal rather than an accounting surprise.

The Well-Architected questions

The AWS Well-Architected Framework groups recurring design questions into six pillars:

Chapter 1: AWS as Programmable Infrastructure 1-5 Well-Architected pillars as recurring review lenses



Apply every lens to each significant design and change.

004

Figure 1-004: Well-Architected pillars as recurring review lenses

- **Operational excellence:** Can the team operate, change, and improve the workload safely?
- **Security:** Are identities, data, systems, and events protected appropriately?
- **Reliability:** Can the workload perform its intended function and recover from failure?
- **Performance efficiency:** Are resources and technologies matched to demand?
- **Cost optimization:** Is the required outcome achieved without unnecessary consumption?
- **Sustainability:** Is resource use efficient enough to reduce avoidable environmental impact?

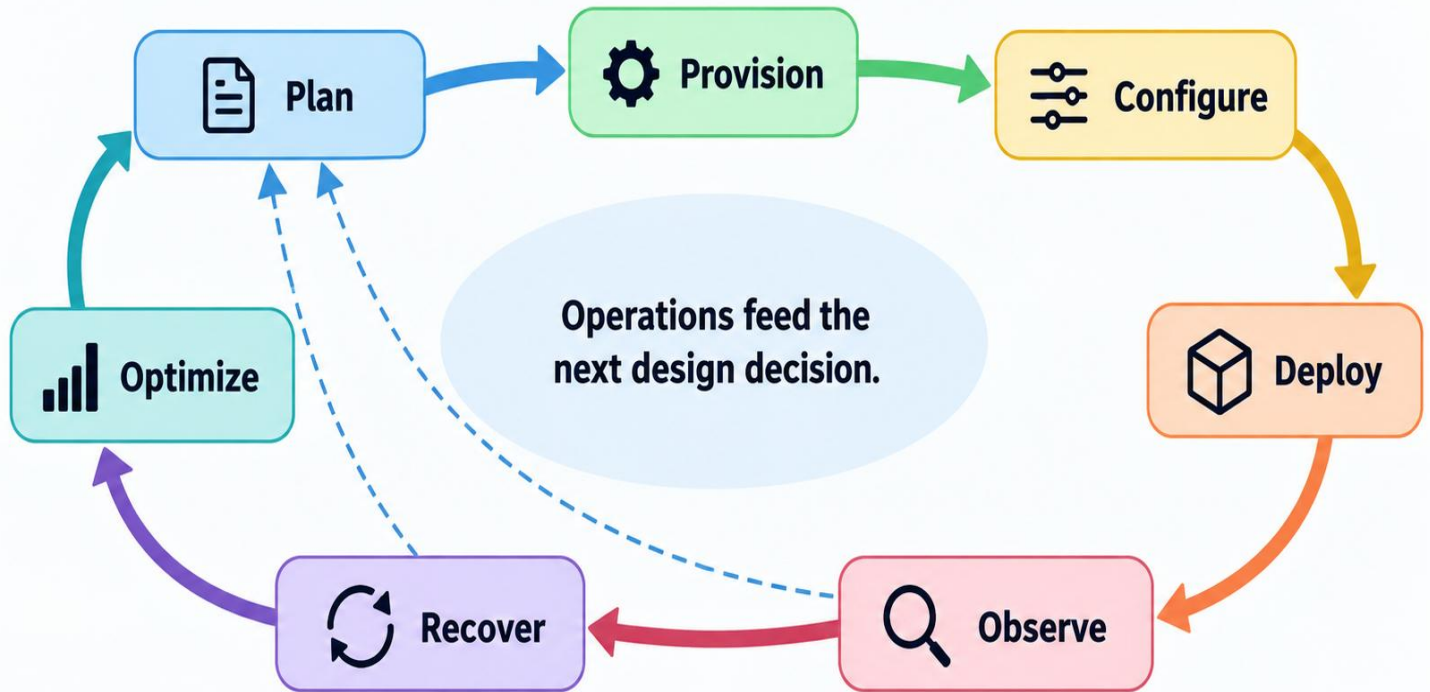
These pillars are not a final inspection checklist. They are questions to ask during design and every significant change. For example, adding a cache may improve performance, but it also creates security, operational, reliability, and cost questions.

The workload lifecycle

A systems engineer commonly contributes throughout the workload lifecycle.

Chapter 1: AWS as Programmable Infrastructure

1-6 Workload lifecycle and feedback from operations to design



005

Figure 1-005: Workload lifecycle and feedback from operations to design

1. **Plan.** Clarify functional needs, expected traffic, data sensitivity, recovery needs, ownership, and constraints.
2. **Provision.** Create accounts, networks, identities, compute, storage, and managed services.
3. **Configure.** Apply routes, security rules, operating-system settings, service parameters, certificates, and application configuration.
4. **Deploy.** Release a traceable application artifact through a controlled process.
5. **Observe.** Collect evidence about health, usage, errors, changes, and user experience.
6. **Recover.** Replace failed capacity, restore data, roll back changes, or fail over according to a tested plan.
7. **Optimize.** Improve security, reliability, performance, operability, and cost based on evidence.

Cloud services shorten provisioning time, but they do not remove these stages. In fact, because change is easier, disciplined change control becomes more important.

Practical comparison: one web application, three execution models

Assume a team must host a small HTTP application.

One EC2 instance

The team installs the operating system packages and application on a virtual machine. This offers broad compatibility and direct troubleshooting access. The team must patch the guest OS, replace failed capacity, manage process startup, design scaling, and avoid making the instance the only copy of important state.

This is often a sensible starting point for legacy software or software that expects a normal server. It is not highly available merely because it runs in AWS.

An ECS service

The team packages the application as a container image and asks Amazon ECS to keep a desired number of tasks running. With Fargate, AWS also manages the underlying server capacity. The team still owns the image, task permissions, network access, health checks, logs, scaling behavior, and data placement.

This is often suitable for long-running services that benefit from standard container packaging without requiring Kubernetes.

A Lambda function

The team writes a handler that runs when API Gateway receives a request. AWS creates and scales execution environments as needed. The team must design within the function model: bounded execution time, stateless processing, explicit permissions, concurrency, retries, and downstream limits.

This is often suitable for event-driven or variable request workloads, but it is not automatically simpler if the application assumes a long-running process or local durable state.

The main lesson is that **moving to a more managed execution model removes some infrastructure tasks while making application and integration design more important.**

Chapter 1: AWS as Programmable Infrastructure

1-7 One HTTP application implemented on EC2, ECS, and Lambda

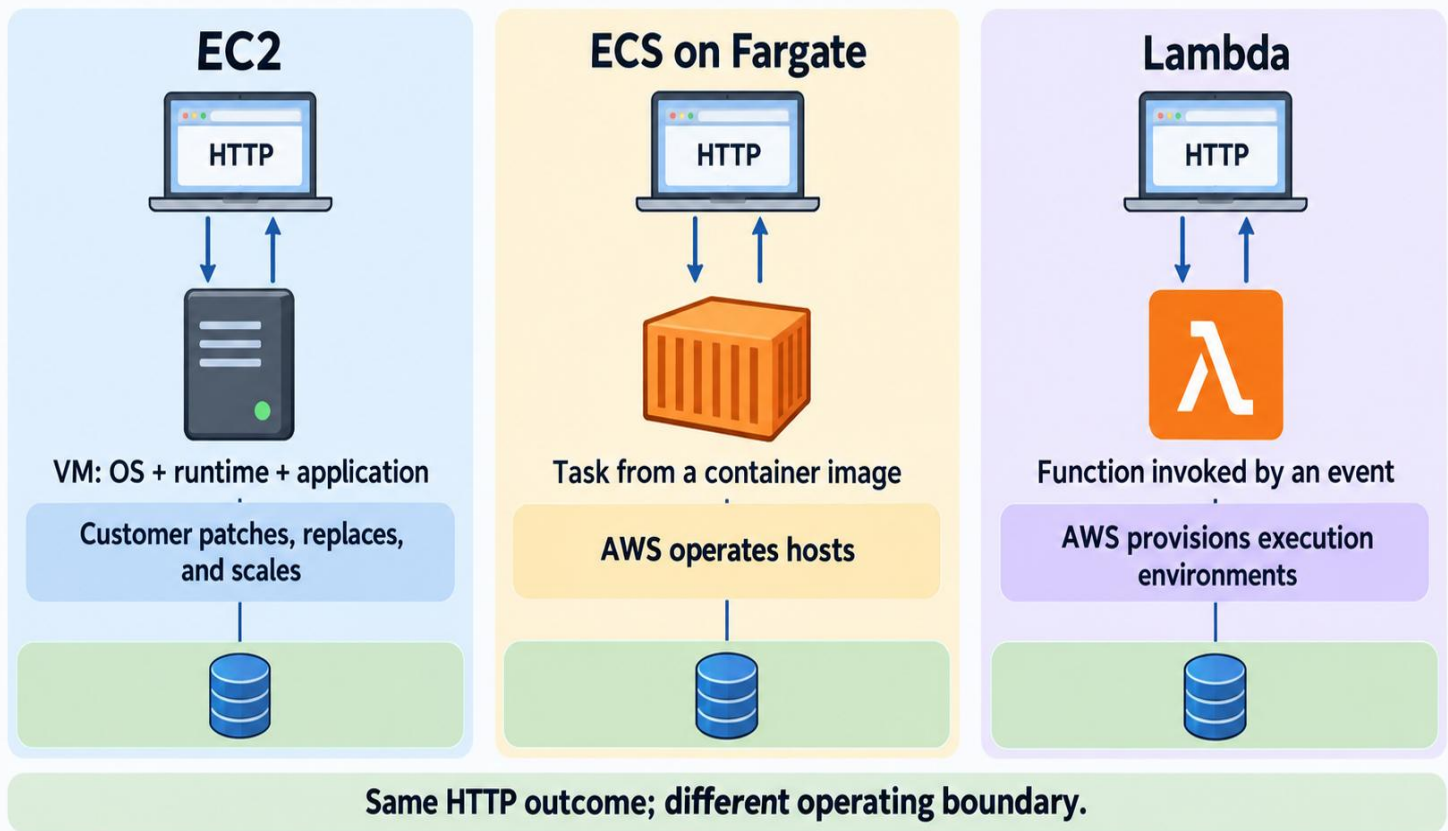


Figure 1-006: One HTTP application implemented on EC2, ECS, and Lambda

Misconceptions that cause significant errors

“The cloud is just someone else’s data center.” AWS still uses physical data centers, but customers operate resources through APIs, elastic capacity, managed services, and explicit failure domains. That operating model changes how systems are designed and controlled.

“Managed means AWS handles everything.” AWS may patch a host or database engine, but the customer still owns permissions, data, application behavior, network exposure, service configuration, and recovery validation.

“If a service is available, the application is highly available.” Application availability depends on every critical layer: traffic entry, compute capacity, dependencies, data, identity, quotas, and deployment behavior.

“Automation makes operations safe.” Automation makes an action repeatable and fast. Safety depends on review, scope, idempotency, testing, observability, and rollback.

What to retain: AWS gives engineers programmable building blocks. The first design decision is the responsibility boundary, and every later decision must consider scope, failure, security, operation, and cost.

Chapter Summary

- AWS turns infrastructure changes into authorized API requests, which makes repeatability and scope control essential.
- Managed services move the operating boundary; they do not remove customer responsibility for applications, data, access, and recovery.
- System qualities describe different outcomes, so high availability, backup, durability, and performance must not be treated as synonyms.
- Architecture remains an ongoing lifecycle of planning, provisioning, deployment, observation, recovery, and improvement.

Review Questions

1. How does API-driven provisioning change the risks and responsibilities of infrastructure work?
2. Which responsibilities remain with the customer when code runs on Lambda?
3. How are availability and durability different?
4. Why should cost and operational evidence influence architecture from the beginning?

Chapter 2 — Accounts, Regions, Availability Zones, and Resource Scope

Every AWS resource belongs to an administrative and geographic scope. Before designing, changing, or troubleshooting anything, identify the active account, Region, Availability Zone when relevant, and whether the resource is global, Regional, or zonal.

What You'll Be Able to Do

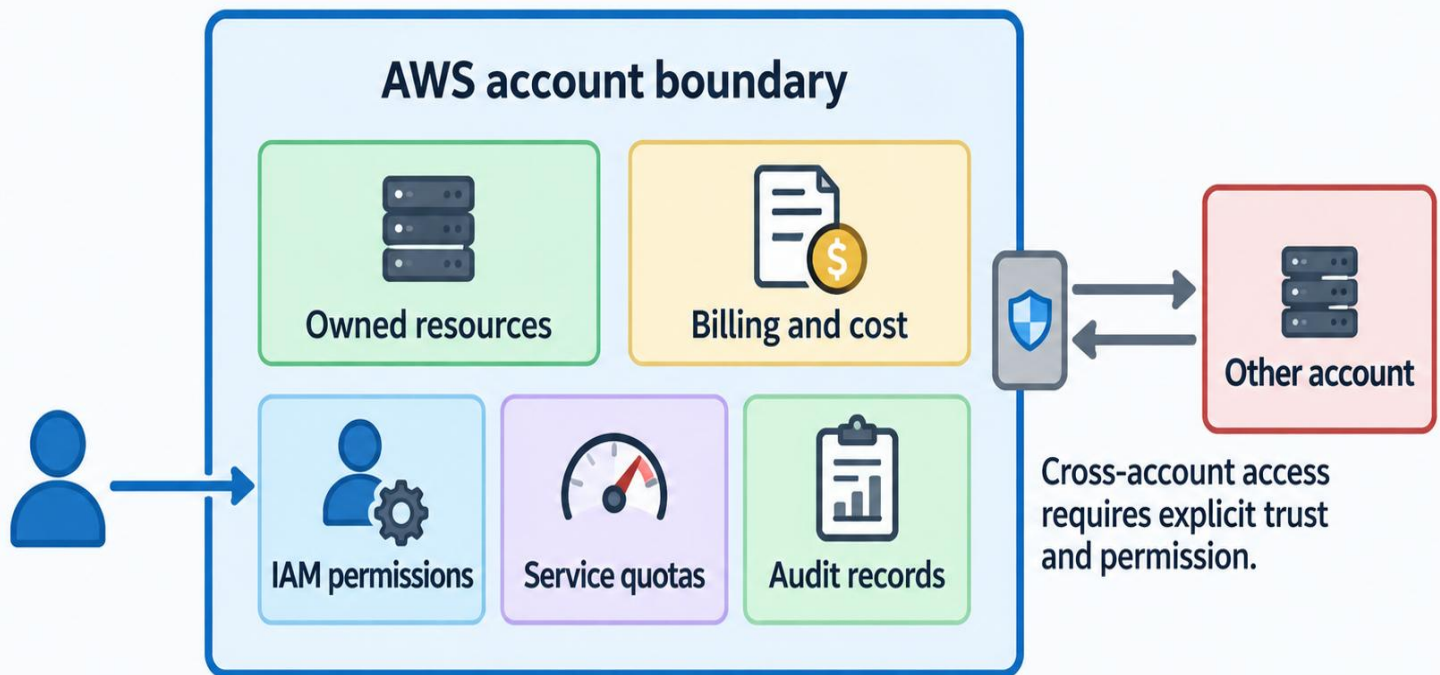
- Identify the account, Region, Availability Zone, and resource scope involved in an AWS design or incident.
- Explain why an AWS account is an ownership and isolation boundary, not only a login or billing container.
- Read the main components of an ARN and use them to reason about resource identity.
- Include Region selection, quotas, and failure domains in deployment planning.

The AWS account is an isolation and ownership boundary

An **AWS account** is a container for resource ownership, billing, identities, quotas, and security boundaries. It is not merely a username or a console login.

Chapter 2: Accounts, Regions, Availability Zones, and Resource Scope

2-1 AWS account as ownership, billing, quota, and isolation boundary



007

Figure 2-007: AWS account as ownership, billing, quota, and isolation boundary

Resources created in one account normally belong to that account. Permissions must explicitly allow access from another account. Usage is measured against account-level billing and quotas. Audit records identify the account in which an action occurred. For these reasons, professional environments commonly separate workloads or lifecycle stages into multiple accounts, although the detailed governance model belongs later in the guide.

A useful analogy is a legal company rather than a folder. A folder groups files; a company owns assets, incurs costs, grants authority, and has boundaries with other companies. Similarly, an AWS account owns resources and establishes an administrative boundary.

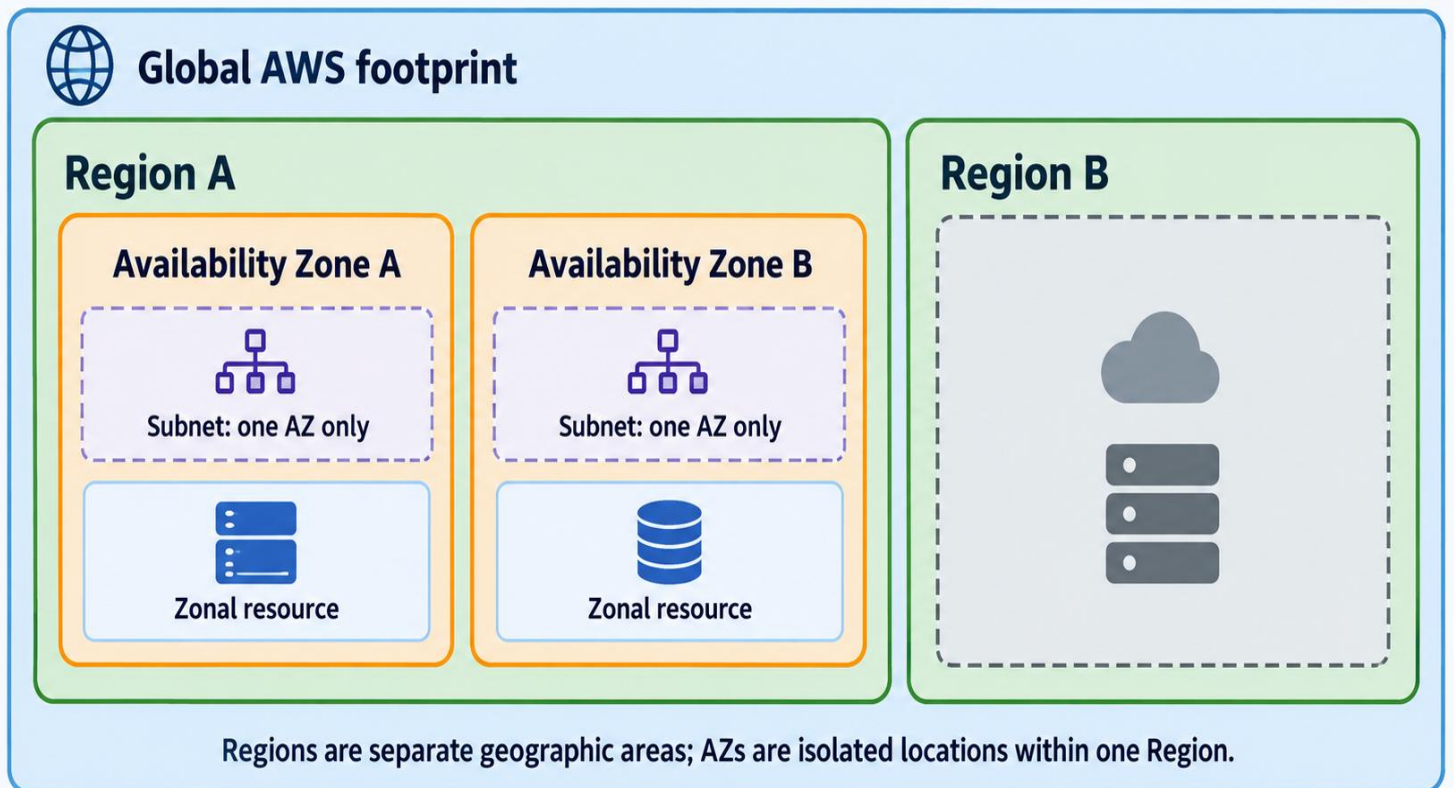
When an engineer says, “I cannot find the instance,” the first question should often be, “**Which account are you in?**”

Regions and Availability Zones

An **AWS Region** is a separate geographic area in which AWS operates services. A Region has a code such as `ap-northeast-1`; engineers use the code in commands, APIs, templates, and endpoints.

Chapter 2: Accounts, Regions, Availability Zones, and Resource Scope

2-2 Global AWS, Regions, Availability Zones, and resources



008

Figure 2-008: Global AWS, Regions, Availability Zones, and resources

An **Availability Zone**, usually shortened to **AZ**, is an isolated infrastructure location within a Region. A Region contains multiple AZs connected by high-bandwidth, low-latency networks. A subnet belongs to exactly one AZ, and many architecture patterns distribute capacity across AZs so that one location is not the only failure domain.

Think of a Region as a metropolitan area and Availability Zones as separate campuses within that area. The campuses are close enough for fast communication but separated so that one local facility problem is less likely to affect all of them.

Multi-AZ and multi-Region designs protect against different failure scopes. Multi-AZ design primarily addresses failures within one Region. Multi-Region design addresses a broader class of events but adds major complexity in data replication, traffic routing, consistency, deployment, security, and testing.

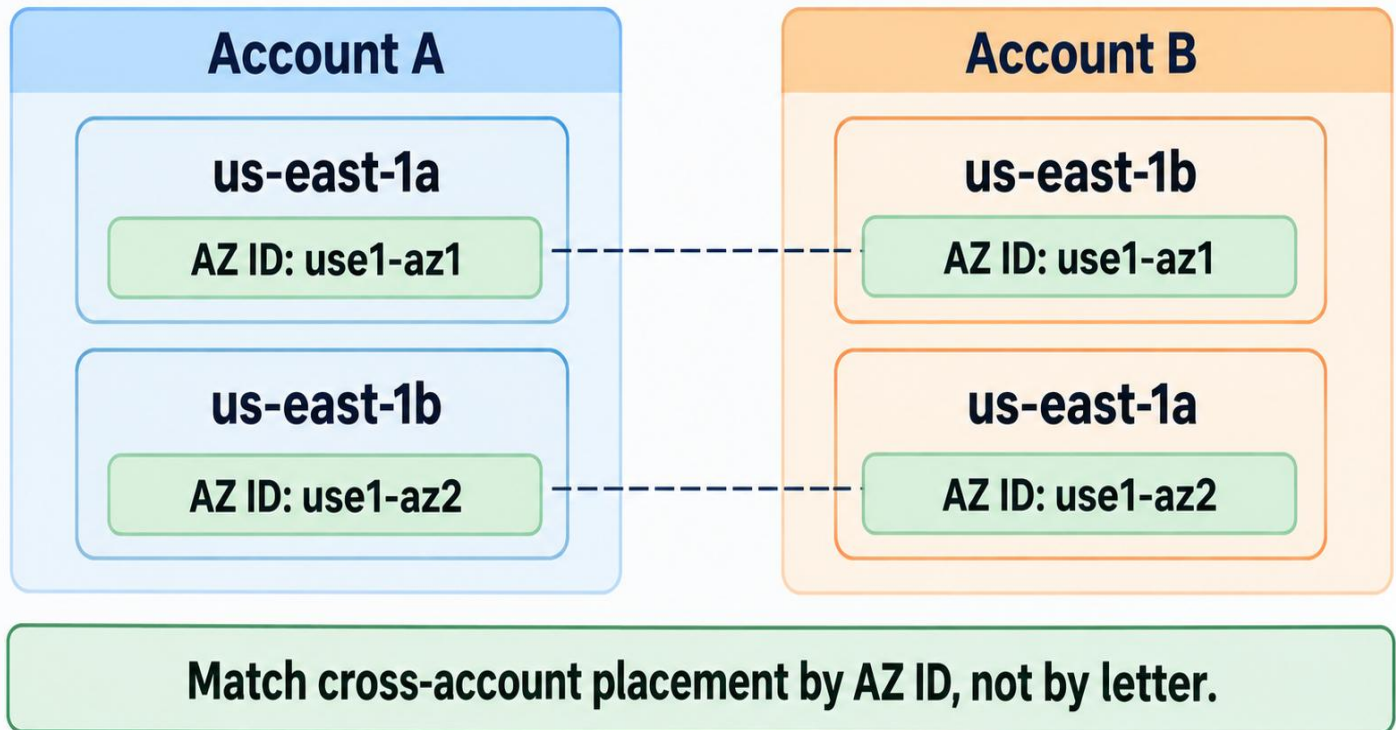
AZ names and AZ IDs

Names such as `us-east-1a` can map to different physical AZs in different AWS accounts. AWS also provides an **Availability Zone ID**, such as `use1-az1`, that identifies the same physical AZ consistently across accounts.

Chapter 2: Accounts, Regions, Availability Zones, and Resource Scope

2-2.1 AZ names versus stable AZ IDs across two accounts

Example mapping



009

Figure 2-009: AZ names versus stable AZ IDs across two accounts

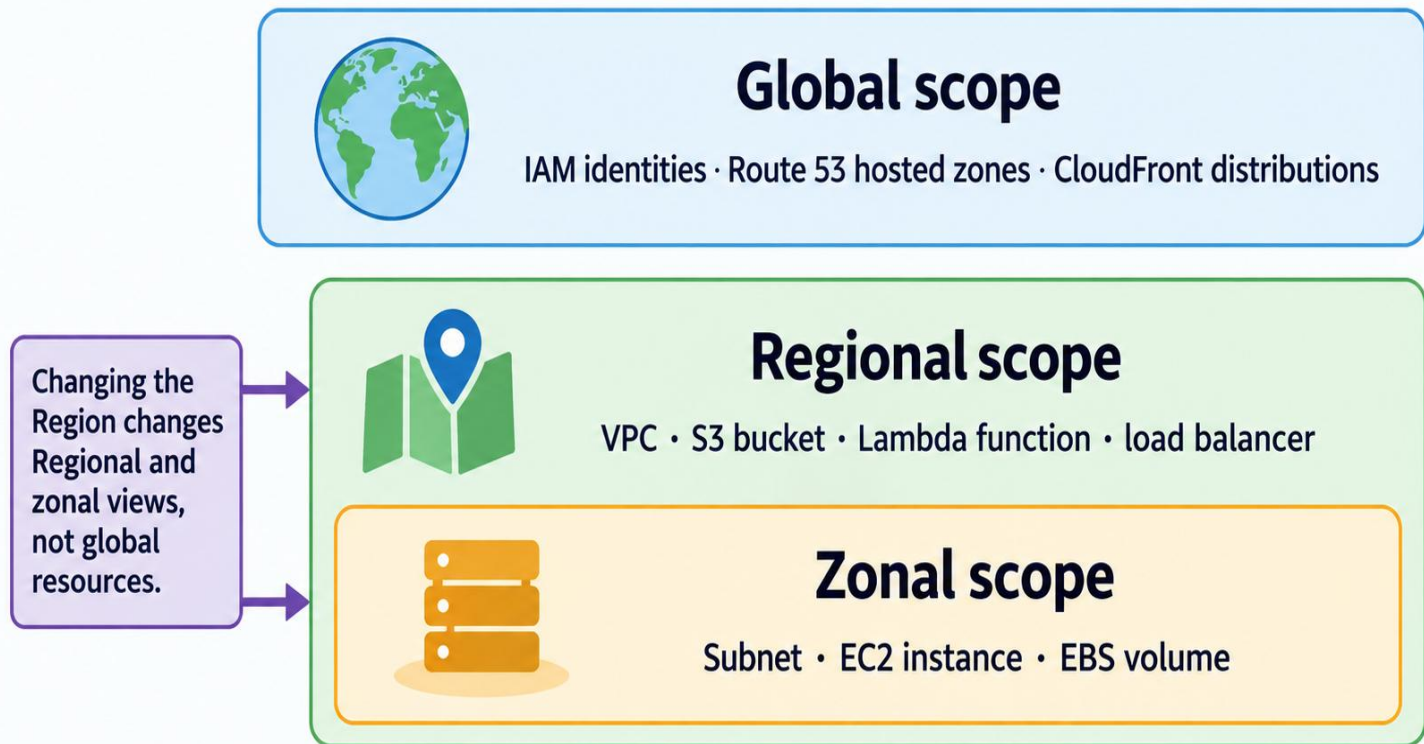
This matters when two accounts must align resources physically, such as shared networking or certain cross-account architectures. Within one simple account, the familiar AZ name is usually sufficient. In cross-account documentation, use AZ IDs when physical alignment matters.

Resource scope: global, Regional, and zonal

The scope belongs to a resource or capability, not always to an entire product name. A service can contain resources with different scopes.

Chapter 2: Accounts, Regions, Availability Zones, and Resource Scope

2-3 Global, Regional, and zonal resource scope



010

Figure 2-010: Global, Regional, and zonal resource scope

Scope	Meaning	Common examples	Operational consequence
Global	The resource is not selected independently per normal Region	IAM identities, Route 53 hosted zones, CloudFront distributions	Changing the console Region may not change what is displayed
Regional	The resource exists in one selected Region	VPCs, S3 buckets and objects, Lambda functions, ECS clusters, load balancers	Commands must use the correct Region; another Region can show an empty result
Zonal	The resource is tied to one Availability Zone	Subnets, EC2 instances, EBS volumes	Zonal failure and cross-zone placement directly affect the resource

Some nuances are important:

- An EC2 service API is Regional, but an individual instance is launched into a subnet in one AZ.
- An Application Load Balancer is a Regional resource that uses enabled subnets in multiple AZs.
- An S3 general-purpose bucket is created in a Region, although its name may participate in a broader namespace.
- IAM is generally treated as global within an AWS partition, but temporary credential services can have Regional endpoints.

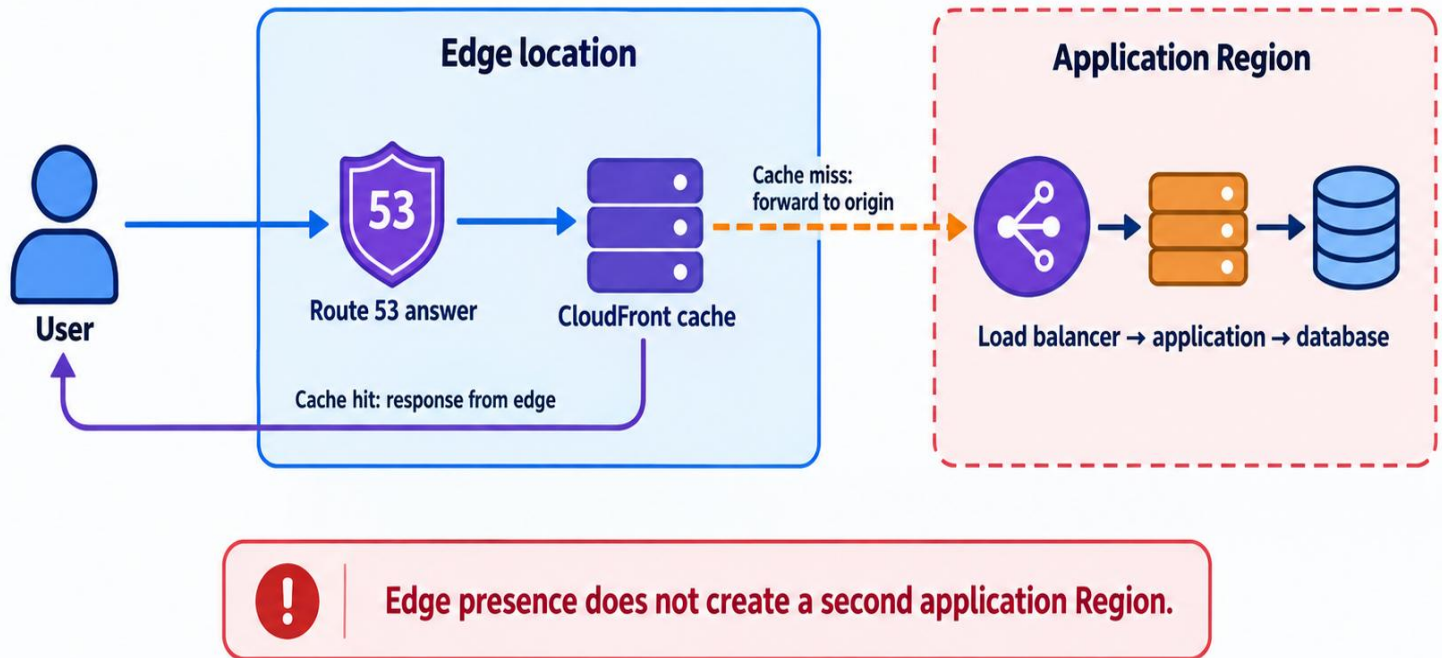
When troubleshooting, ask **which exact resource is scoped where**, rather than memorizing a simplistic “service X is global” rule.

Edge infrastructure is not a second application Region

AWS operates **edge locations**, which are sites close to users that support services such as content delivery and DNS. Amazon CloudFront can cache content at the edge, and Route 53 answers DNS queries through a globally distributed system.

Chapter 2: Accounts, Regions, Availability Zones, and Resource Scope

2-4 Edge request processing versus the application Region



011

Figure 2-011: Edge request processing versus the application Region

Edge presence can reduce latency and protect an origin, but it does not automatically make the application itself multi-Region. If the only database and application tier remain in one Region, an edge cache does not create a second writable application stack.

For the reference web system, a request might follow this path:

1. Route 53 answers the DNS query.
2. CloudFront receives the HTTPS request at an edge location.
3. CloudFront serves a cached response or forwards the request to a Regional origin.
4. A Regional load balancer sends the request to a healthy application target in one of two AZs.
5. The application accesses a Regional managed database.

The components have different scopes and different failure behavior. Correct architecture documentation should make those boundaries visible.

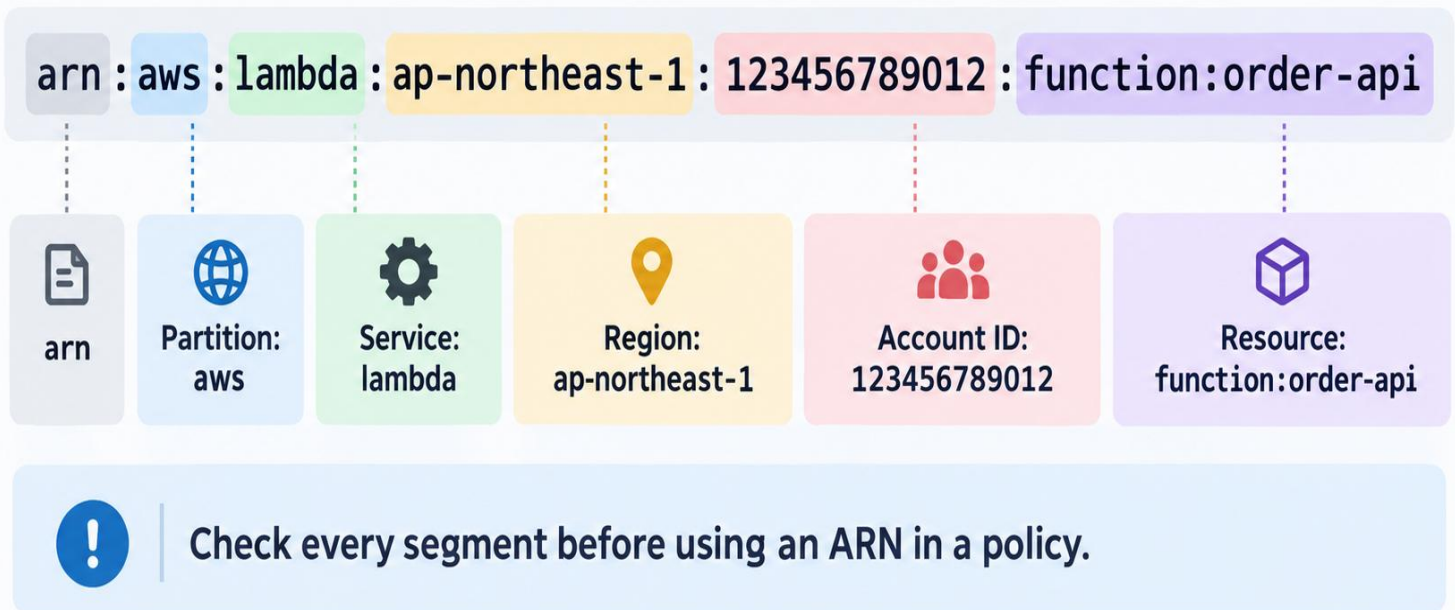
Resource identifiers and ARNs

AWS resources commonly have a friendly name, a generated ID, or an **Amazon Resource Name (ARN)**. An ARN is a standardized identifier used in policies, logs, APIs, and cross-service references.

Chapter 2: Accounts, Regions, Availability Zones, and Resource Scope

2-5 Anatomy of an ARN

ARN anatomy



012

Figure 2-012: Anatomy of an ARN

A typical ARN has this form:

```
arn:partition:service:region:account-id:resource
```

For example:

```
arn:aws:lambda:ap-northeast-1:123456789012:function:order-api
```

This ARN indicates:

- `aws`: the AWS partition
- `lambda`: the service namespace
- `ap-northeast-1`: the Region
- `123456789012`: the owning account
- `function:order-api`: the resource type and name

Not every ARN fills every component. For example, some global resources omit the Region. Do not infer permissions from a friendly name alone; inspect the full identifier and scope.

Choosing a Region

A Region should be selected from workload requirements, not habit. Common decision factors are parallel considerations:

- **Latency:** distance between users, systems, and the Region
- **Data residency:** where data is permitted or required to remain
- **Compliance:** organizational, contractual, or regulatory requirements
- **Service availability:** whether required features are offered in the Region
- **Cost:** Regional differences in service and transfer pricing
- **Integration:** proximity to existing systems and networks
- **Recovery:** whether another Region is part of a defined recovery strategy

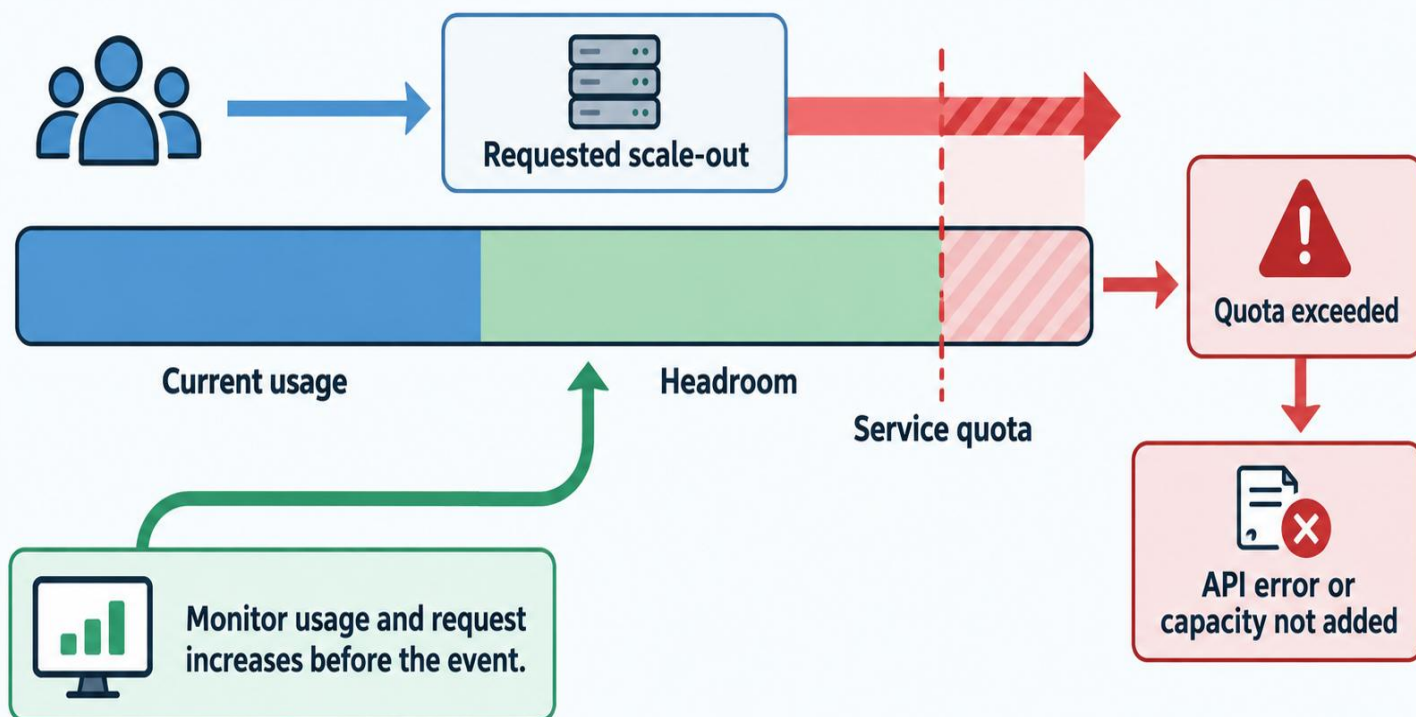
A company may standardize on a small set of approved Regions. That is an organizational policy, not a universal AWS rule.

Quotas and capacity constraints

A **service quota** is a limit on the amount or rate of a resource or API operation. Quotas can apply per account, per Region, or to another defined scope. Examples include counts of resources, API request rates, or concurrency.

Chapter 2: Accounts, Regions, Availability Zones, and Resource Scope

2-7 Quota, current usage, headroom, and scaling failure



013

Figure 2-013: Quota, current usage, headroom, and scaling failure

Quotas are operational constraints, not obscure administrative details. A deployment can fail even when its template is correct if the account has reached a limit. An application can throttle during a traffic event if request or concurrency quotas are too low.

Before a major launch or migration:

1. Identify the services on the critical path.
2. Check current quota usage in the target account and Region.
3. Request increases early where supported.
4. Test behavior when a limit is reached.
5. Monitor quota-related signals during the event.

Exact default values are intentionally not memorized here because they change by service, Region, account history, and time.

Worked scope trace

Consider a public web application with these components:

- Route 53 public hosted zone
- CloudFront distribution
- Application Load Balancer in `ap-northeast-1`
- Application instances in subnets in two AZs
- RDS database deployed for high availability within the Region

Trace the scope:

1. **Route 53 hosted zone:** global control-plane resource used to publish DNS records.
2. **CloudFront distribution:** global distribution using edge locations.
3. **Load balancer:** Regional resource enabled in multiple AZ subnets.
4. **EC2 instances:** zonal resources, each launched into one subnet and AZ.
5. **RDS deployment:** Regional managed database configuration with database capacity placed across AZs according to its deployment mode.

Now consider failures:

- One EC2 instance failure should affect only that target if healthy capacity remains.
- One AZ failure should be survivable only if the load balancer, application capacity, and database deployment are designed across AZs.
- A Regional disruption is not solved merely by using two AZs.

- A DNS or edge layer can direct traffic, but it cannot manufacture a healthy application stack in another Region.

Chapter 2: Accounts, Regions, Availability Zones, and Resource Scope

2-8 Worked scope trace for a public web application

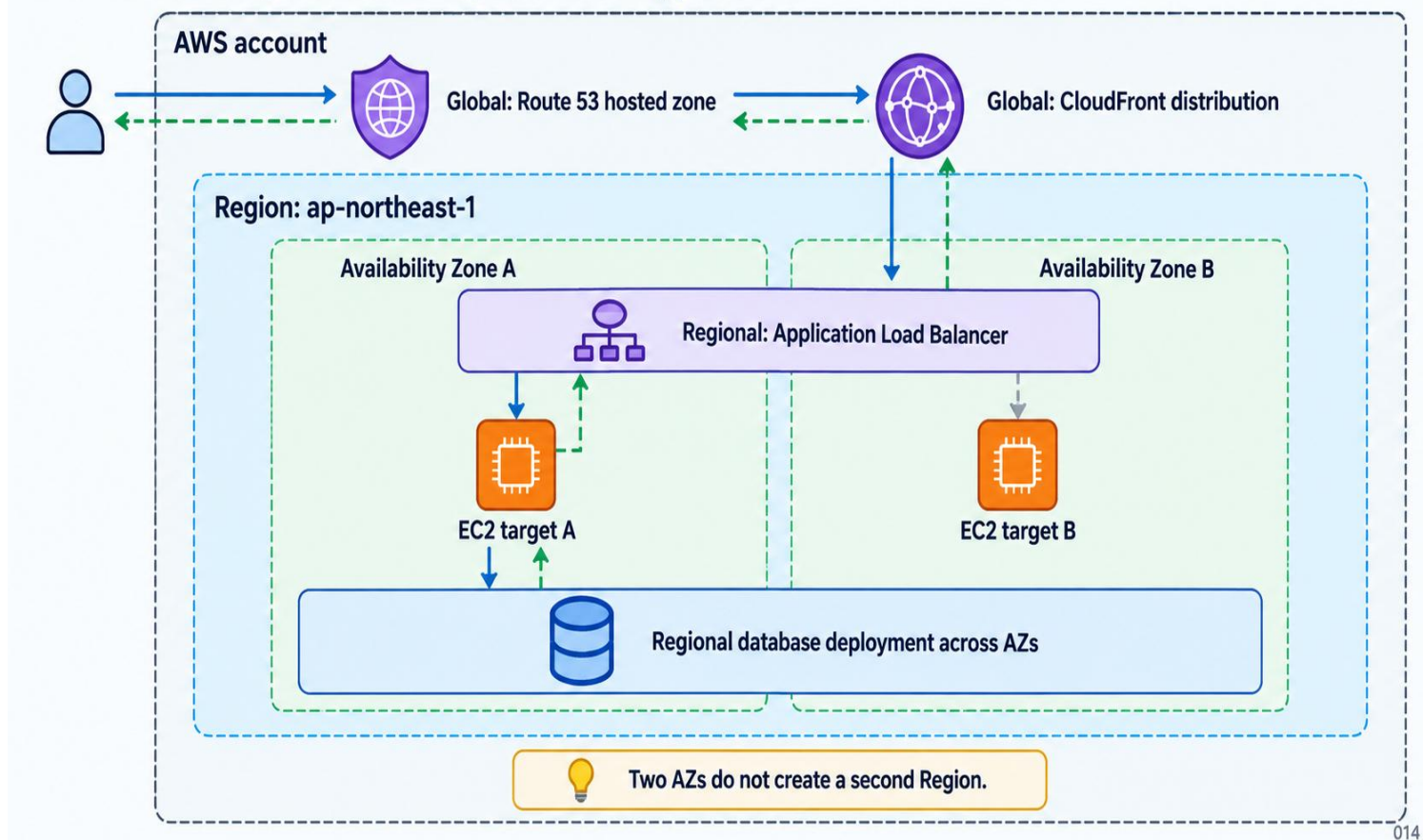


Figure 2-014: Worked scope trace for a public web application

Frequent scope mistakes

“The resource does not exist because the console is empty.” The console may be displaying the wrong account or Region.

“us-east-1a means the same physical zone in every account.” The letter mapping can differ; use the AZ ID when cross-account physical alignment matters.

“A global front door makes the workload multi-Region.” Global DNS or content delivery can route traffic, but a second functioning application and data stack must actually exist.

“All resources from one AWS service have the same scope.” Scope must be checked at the resource level.

What to retain: Account, Region, AZ, and resource scope are the coordinates of AWS operations. Verify them before reading an architecture, running a command, or diagnosing a failure.

Chapter Summary

- Account and geographic scope determine ownership, permissions, endpoints, failure exposure, and where resources can be found.
- Regions contain multiple Availability Zones, while global, Regional, and zonal resources behave at different scopes.
- ARNs and stable identifiers help engineers identify the service, account, Region, and resource involved.
- Quotas and Regional capacity are operational constraints that must be checked before scaling or a major event.

Review Questions

1. Why is an AWS account a stronger boundary than a folder?
2. What failure does a multi-AZ design address, and what failure does it not address?
3. Which ARN components help identify the owner and location of a resource?
4. Why can a valid scaling policy still fail during demand growth?