

THE PRACTICAL GUIDE

AWS for Developers

Building and Deploying on the Cloud

Serverless · Containers · Data · IaC · CI/CD
Security · Observability · Cost · Generative AI

LAMBDA · DYNAMODB · ECS · CDK · BEDROCK

THENEXBOOKS

AWS for Developers

The Practical Guide to Building and Deploying on the Cloud

First Edition

THENEXBOOKS

AWS for Developers
The Practical Guide to Building and Deploying on the Cloud

First Edition, 2026

Copyright © 2026 thenexbooks (thenexbooks.gumroad.com). All rights reserved.

No part of this publication may be reproduced, distributed, or transmitted in any form or by any means without the prior written permission of the publisher, except for brief quotations in reviews and certain other non-commercial uses permitted by copyright law. Code examples may be used freely in your own projects.

The information in this book is provided "as is" without warranty of any kind. While every precaution has been taken in its preparation, the publisher assumes no responsibility for errors or omissions, or for damages — including cloud charges — resulting from the use of the information contained herein. Cloud services, features, quotas, and prices change frequently; always consult the official documentation and pricing pages for the services and regions you use, and delete resources you no longer need.

This book is an independent publication and is not affiliated with, sponsored by, or endorsed by Amazon Web Services, Inc. or its affiliates. Amazon Web Services, AWS, and the names of AWS services are trademarks of Amazon.com, Inc. or its affiliates. GitHub, Docker, PostgreSQL, Node.js, TypeScript, Python, OpenTelemetry, Terraform, and other product names are trademarks of their respective owners; their use here is for identification only. Lumora is a fictional company.

Preface

Most developers meet AWS in fragments. A tutorial shows how to launch a virtual machine, a blog post explains a Lambda function, a colleague pastes an IAM policy that "just works". Each fragment makes sense on its own, yet building and running a real application on AWS still feels like assembling a puzzle without the picture on the box. Which compute option should you choose? How do identities and permissions actually fit together? Why did last month's bill surprise you?

This book is the picture on the box. It is written for developers who want to build, deploy, secure, observe, and pay sensibly for real applications on AWS — not to memorize every service, but to understand the handful of building blocks that most applications are made of, and the principles that connect them. We explain *why* each service exists, when to use it, when not to, and what tends to go wrong in production.

Throughout the book we build one application end to end, so every new service has a concrete job to do. By the final chapters you will have designed and deployed a serverless backend, a containerized service with a delivery pipeline, and a production-ready multi-account setup.

Who This Book Is For

- **Application developers** who write JavaScript/TypeScript or Python and want to build and deploy on AWS with confidence.
- **Backend and full-stack engineers** moving from on-premises or another cloud to AWS.
- **Frontend and mobile developers** who need to understand the cloud backends their apps depend on.
- **Tech leads and architects** who want a clear, practical reference for designing, reviewing, and costing AWS workloads.

What You Will Learn

- Cloud fundamentals, a safe account setup, and IAM explained properly — identities, policies, and least privilege.
- How to program AWS with the SDKs: credentials, retries, pagination, and error handling.
- Compute choices — EC2, containers on ECS and Fargate, Lambda, and EKS — and how to choose between them.
- Data on AWS: S3, DynamoDB, RDS and Aurora, ElastiCache, and analytics with Athena and OpenSearch.
- Networking, messaging, events, workflows, and content delivery at the edge.
- Infrastructure as code with CloudFormation, SAM, and the AWS CDK; CI/CD with GitHub Actions and safe deployments.
- Observability, application security, cost optimization, and the Well-Architected Framework.

- Generative AI on AWS with Amazon Bedrock, and three complete, end-to-end projects.

How This Book Is Organized

Part I — Cloud Foundations (Chapters 1–4) covers the concepts, account setup, IAM, and the SDKs. **Part II — Compute** (Chapters 5–9) explores EC2, containers, Lambda, API Gateway, and how to choose. **Part III — Storage and Data** (Chapters 10–14) covers object storage, NoSQL and relational databases, caching, and analytics. **Part IV — Integration, Networking, and Delivery** (Chapters 15–18) explains VPCs, messaging, workflows, and the edge. **Part V — DevOps, Security, and Operations** (Chapters 19–24) covers infrastructure as code, CI/CD, observability, security, and cost. **Part VI — AI and Real-World Projects** (Chapters 25–28) adds generative AI and brings everything together. The appendices provide a CLI and SDK cheat sheet, templates and a service reference, a production checklist, and a glossary.

Every chapter ends with a **Try It Yourself** section and a **Key Takeaways** box.

The Running Example: Lumora

Throughout the book we build **Lumora**, a fictional online learning platform. Instructors publish courses made of video lessons, readings, and quizzes; learners enroll, stream lessons, track their progress, take quizzes, and earn certificates. Lumora has a web app, a mobile app, an instructor dashboard, and an AI study assistant — enough moving parts to give every service in this book a real job.

A Word About Cost and Safety

Following along creates real resources in a real AWS account, and most resources cost money while they exist. Before you start, set up a budget alert as described in Chapter 2, work in a dedicated learning account, and delete what you create at the end of each exercise. We deliberately avoid quoting prices, because they vary by region and change over time — instead we explain what drives the cost of each service so that you can estimate it with the official pricing pages and calculator.

Conventions Used in This Book

> **TIP**
>
> A recommended practice or a technique that makes your work easier.

> **NOTE**
>
> Background information, version details, or a useful aside.

> **WARNING**
>
> A common mistake, a security risk, or something that can cause unexpected charges or outages.

Code appears in `monospace`. Examples use TypeScript on Node.js with the AWS SDK for JavaScript v3, plus Python with boto3 where helpful. Sample account IDs, domains, and keys follow AWS documentation conventions (for example `111122223333`) and are not real. Features that are in preview are labeled as such.

Contents

Preface

PART I — CLOUD FOUNDATIONS

1 Why AWS? Cloud Concepts Every Developer Needs

2 Setting Up: Accounts, IAM Identity Center, the AWS CLI, and Budgets

3 IAM in Depth: Identities, Policies, and Least Privilege

4 Programming AWS: SDKs, Credentials, Retries, and Pagination

PART II — COMPUTE

5 Amazon EC2 Essentials for Developers

6 Containers on AWS: ECR, ECS, and Fargate

7 AWS Lambda Fundamentals

8 Building HTTP APIs with API Gateway and Lambda

9 Choosing Compute: EC2, Containers, Serverless, and EKS

PART III — STORAGE AND DATA

10 Amazon S3: Object Storage Done Right

11 Amazon DynamoDB: Modeling and Access Patterns

12 Relational Databases: Amazon RDS and Aurora

13 Caching with Amazon ElastiCache

14 Analytics and Search: Athena, Glue, and OpenSearch

PART IV — INTEGRATION, NETWORKING, AND DELIVERY

15 Networking with Amazon VPC

16 Messaging and Events: SQS, SNS, and EventBridge

17 Orchestrating Workflows with Step Functions

18 Delivery at the Edge: CloudFront, Route 53, and ACM

PART V — DEVOPS, SECURITY, AND OPERATIONS

19 Infrastructure as Code: CloudFormation and AWS SAM

20 Infrastructure as Code with the AWS CDK

21 CI/CD on AWS: Pipelines, GitHub Actions, and Safe Deployments

22 Observability: CloudWatch, X-Ray, and OpenTelemetry

23 Security for Applications: Cognito, Secrets Manager, KMS, and WAF

24 Cost Optimization and the Well-Architected Framework

PART VI — AI AND REAL-WORLD PROJECTS

25 Generative AI on AWS with Amazon Bedrock

26 Project: The Lumora Serverless Backend

27 Project: The Lumora Grading Service on ECS with CI/CD

28 Project: Production Readiness, Multi-Account Setup, and Next Steps

APPENDICES

A AWS CLI and SDK Cheat Sheet

B IaC Templates and Service Reference

C Production Checklist

D Glossary

PART I

Cloud Foundations

The concepts, the account setup, the identity and permission model, and the programming model you will rely on in every chapter that follows.

1 · Why AWS? Cloud Concepts Every Developer Needs

2 · Accounts, IAM Identity Center, the CLI, and Budgets

3 · IAM in Depth

4 · Programming AWS with the SDKs

CHAPTER 1 · PART I — CLOUD FOUNDATIONS

Why AWS? Cloud Concepts Every Developer Needs

Before we write a single line of infrastructure code, we need a working mental model of what AWS actually is: a global fleet of data centers exposed as APIs, a split of security duties between you and AWS, a spectrum of managed services, and a billing model driven by usage. This chapter builds that model and introduces Lumora, the learning platform we will build throughout the book.

What the Cloud Really Gives a Developer

It is easy to describe cloud computing as "someone else's computer," and that description is not entirely wrong. But it misses what makes AWS useful to a developer day to day. The value is not that the servers live in someone else's building. The value is that every piece of infrastructure — a virtual machine, a database, a message queue, a TLS certificate, a DNS record, a firewall rule — can be created, changed, and destroyed through an API call, in seconds or minutes, and billed only for as long as it exists.

When infrastructure becomes programmable, developers can own more of the path from code to production, experiment cheaply, and throw away environments they no longer need.

Four characteristics are worth naming explicitly, because we will lean on each of them throughout the book:

- **On-demand self-service.** You provision resources yourself, without waiting for another team or a vendor. The same APIs are available to a solo developer and to a large enterprise.
- **Elasticity.** Capacity can grow and shrink with demand. A course launch that brings a burst of traffic can be absorbed by scaling out, and the extra capacity can disappear once the burst is over.
- **Pay for what you use.** Most services charge by some measure of usage — time, requests, storage, or data moved — instead of an up-front purchase. That is powerful and also dangerous: forgotten resources keep costing money.
- **Managed building blocks.** AWS runs a large catalog of higher-level services (databases, queues, workflow engines, identity, AI models) so that you do not have to operate them yourself. Choosing the right level of abstraction is one of the most important skills this book aims to teach.

Why AWS specifically?

Several large cloud providers offer comparable fundamentals, and many of the concepts in this book transfer between them. AWS is a reasonable default for many teams because of the breadth of its service catalog, its long operating history, its global footprint, and the size of its ecosystem: libraries, tooling, documentation, community answers, and people who already know it. For a developer, the practical benefits are that most problems you hit have been hit before, most languages have a mature SDK, and most third-party tools integrate with it.

Meet Lumora

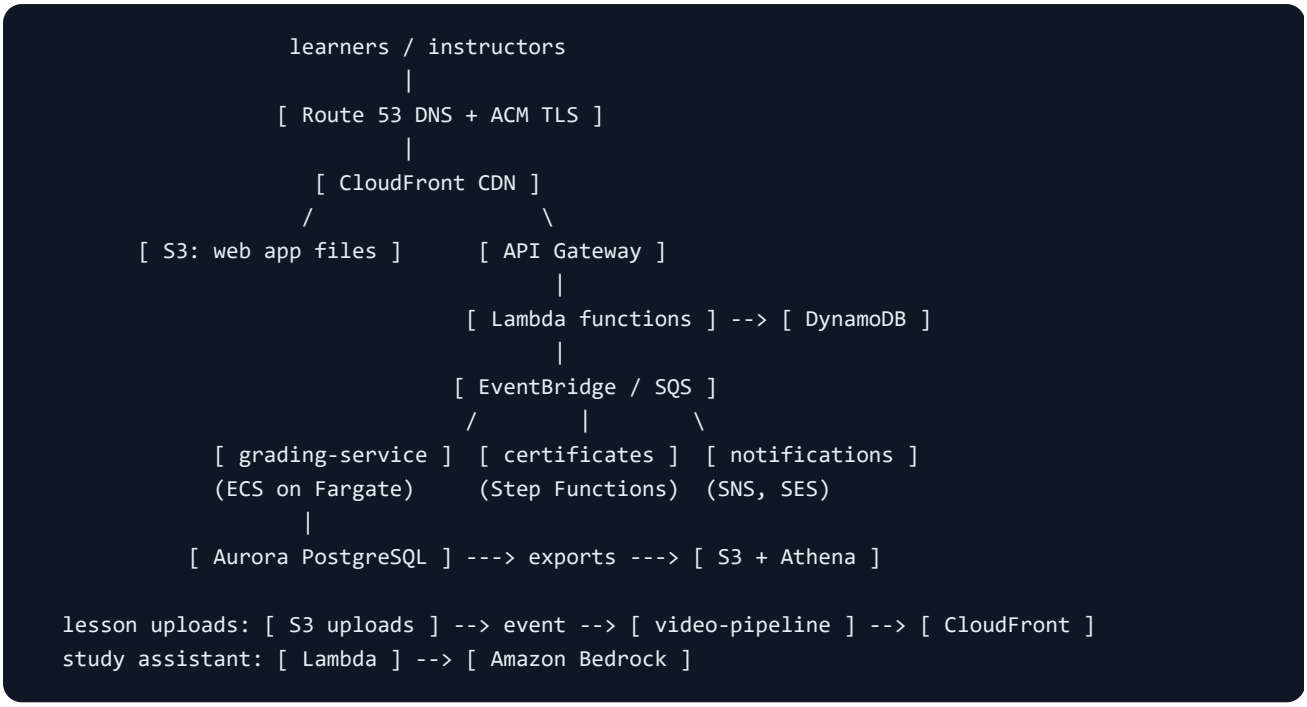
Abstract concepts stick better when they are attached to a real system. Throughout this book we will design, build, secure, and operate **Lumora**, a fictional online learning platform. Lumora is small enough to understand in one sitting but rich enough to exercise most of the AWS services a working developer encounters.

On Lumora, **instructors** publish **courses** made of **lessons**: short videos, reading material, and quizzes. **Learners** browse the catalog, create an **enrollment**, stream lessons, and have their **progress** tracked. They take **quizzes**, and each submission is a **quiz attempt** that must be graded. When a learner finishes a course, the platform issues a **certificate**. **Payments** are handled by an external payment provider, so Lumora never stores card data, and **notifications** (emails and in-app messages) keep everyone informed.

Lumora has four front doors: a single-page web application, a mobile app, an instructor dashboard, and an AI study assistant that answers questions about course material. Behind them sit several backend components, each of which becomes a teaching vehicle in a later chapter:

Lumora component	What it does	Main AWS services	Chapters
Web app	Static SPA for learners	S3, CloudFront, Route 53, ACM	10, 18
`courses-api`	Catalog, enrollment, progress	API Gateway, Lambda, DynamoDB	7, 8, 11
`video-pipeline`	Lesson upload and processing	S3, Lambda, Step Functions	10, 17
`grading-service`	Grades quiz attempts	ECS on Fargate, ECR	6, 27
`reporting`	Relational reports, analytics	Aurora PostgreSQL, Athena	12, 14
`notifications`	Emails and event fan-out	SQS, SNS, EventBridge, SES	16
`certificates`	Issue course certificates	Step Functions	17
Study assistant	AI answers about lessons	Amazon Bedrock	25
Identity and secrets	Sign-in, keys, encryption	Cognito, Secrets Manager, KMS	23

Here is a high-level sketch of how the pieces fit together. Do not worry if many of the boxes are unfamiliar; by the end of the book each one will be something you have built or configured yourself.



This architecture mixes styles on purpose. Most of Lumora is serverless because a young product benefits from paying per request. The grading service runs in containers because grading code may need specific runtimes and longer execution. Reporting uses a relational database for flexible queries. Real systems are rarely pure.

AWS Global Infrastructure

Every AWS resource you create lives somewhere physical. Understanding where — and what failures that location protects you from — is the foundation for designing systems that stay up.

Regions

A **Region** is a separate geographic area, such as `us-east-1` (Northern Virginia) or `eu-west-1` (Ireland). Each Region is designed to be isolated from the others: an outage in one Region should not take down another. Most AWS services are *regional*, which means that a DynamoDB table, a Lambda function, or an SQS queue you create in `us-east-1` exists only there. If you switch the console to another Region, your resources seem to disappear. They have not been deleted; you are simply looking at a different Region.

In this book, examples use `us-east-1` because it is the convention in AWS documentation. You can choose any Region that suits you; just be consistent, and remember to substitute your Region wherever you see ours.

Availability Zones

Each Region contains multiple **Availability Zones** (AZs). An AZ is one or more discrete data centers with independent power, cooling, and networking, located far enough from the other AZs in the Region to reduce the chance of a shared failure, yet close enough for low-latency connections between

them. AZs are the unit of fault isolation you design around most often.

Many managed services spread themselves across AZs automatically. DynamoDB, S3 (in its standard storage classes), SQS, and Lambda all handle this for you. Other services make it your choice: an EC2 instance lives in exactly one AZ, and an RDS database is single-AZ unless you enable a Multi-AZ deployment. The rule of thumb is simple: **anything that must survive the loss of a data center needs to run in at least two AZs**, either because the service does it for you or because you configured it.

One subtle detail: AZ *names* such as `us-east-1a` are mapped independently for each account, so your `us-east-1a` may not be the same physical zone as another account's `us-east-1a`. When you need to coordinate across accounts, use the **AZ ID** (for example `use1-az1`), which identifies the same physical location for everyone.

```
# List the Regions enabled for your account
aws ec2 describe-regions --query "Regions[].RegionName" --output table

# List the Availability Zones in a Region, with their names and IDs
aws ec2 describe-availability-zones --region us-east-1 \
  --query "AvailabilityZones[].[ZoneName,ZoneId,State]" --output table
```

We will set up the AWS CLI properly in Chapter 2, so keep these commands in mind rather than running them now.

Edge locations and other footprints

Beyond Regions, AWS operates a much larger number of **edge locations** (points of presence) that bring content close to users. CloudFront caches Lumora's web app and lesson videos at the edge, and Route 53 answers DNS queries there too. Extensions such as Local Zones and Outposts place a subset of services near specific cities or inside your own facilities. Most developers meet the edge through CloudFront (Chapter 18).

Global versus regional services

A handful of services are global rather than regional. IAM, which controls who can do what in your account, is global: a role you create is usable from every Region. CloudFront distributions and Route 53 hosted zones are global too. Amazon S3 is a special case worth remembering: bucket *names* share a global namespace, so no two accounts can own a bucket with the same name, but each bucket's data lives in the Region you choose. That is why sample bucket names in this book include the account ID, as in `lumora-lesson-uploads-111122223333`.

Choosing a Region

Choosing a Region is a real design decision, not a formality. Weigh these factors:

- **Latency to your users.** Physical distance matters. If most of Lumora's learners are in Europe, a European Region will generally feel faster for API calls that cannot be cached at the edge.
- **Data residency and compliance.** Some organizations and regulations require personal data to stay within a particular country or jurisdiction. Your Region choice is often the main control for that.
- **Service and feature availability.** Not every service or feature is available in every Region, and new features often launch in a subset first. Check the Regional services list before committing, especially for newer services such as those used for generative AI.
- **Price.** Prices for the same service can differ between Regions. Check the pricing pages for the Regions you are considering rather than assuming they are identical.

For most new projects, pick one Region close to your users, build everything there, and design for multiple AZs within it. Multi-Region architectures are powerful but add real complexity around data replication and consistency; they are worth it when your availability or latency requirements demand it, not by default. We discuss disaster recovery strategies in Chapter 28.

> **TIP**
> If resources "vanish" in the console, check the Region selector in the top navigation bar before anything else. With the CLI, run `aws configure get region` or add `--region` explicitly to confirm where you are looking.

Everything Is an API

The AWS Management Console, the AWS CLI, the SDKs, CloudFormation, the CDK, and third-party tools such as Terraform all end up doing the same thing: sending signed HTTPS requests to AWS service endpoints. The console is a friendly client of those APIs, not a separate system. Once you internalize this, many things become easier to reason about. A permission error in the console and a permission error in your Lambda function come from the same place, and anything you can click, you can automate.

Each request is signed with credentials using a scheme called **Signature Version 4** (SigV4). You will almost never implement signing yourself; the SDKs and CLI do it for you. What matters is that every request carries an identity, AWS evaluates that identity's permissions for the specific action and resource, and the request succeeds or fails accordingly. Chapter 3 explains that evaluation in depth.

Control plane and data plane

It helps to distinguish two kinds of API calls. **Control plane** operations create, configure, and delete resources: `CreateTable`, `CreateFunction`, `PutBucketPolicy`. **Data plane** operations use those resources: `GetItem`, `Invoke`, `GetObject`. The distinction matters in practice. Data plane operations are designed for high request rates and are what your application calls on every user request. Control

plane operations are typically lower volume, may take longer to complete, and usually belong in your deployment tooling rather than in request-handling code. A Lambda function that creates a DynamoDB table on each invocation is almost always a design mistake.

Amazon Resource Names

Resources are identified by **Amazon Resource Names** (ARNs). You will see ARNs in policies, logs, error messages, and API responses, so it is worth learning to read them. The general format is:

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

# Examples from Lumora
arn:aws:dynamodb:us-east-1:111122223333:table/lumora-courses
arn:aws:lambda:us-east-1:111122223333:function:lumora-get-course
arn:aws:sqs:us-east-1:111122223333:lumora-grading-queue
arn:aws:iam::111122223333:role/lumora-courses-api-role
arn:aws:s3:::lumora-lesson-uploads-111122223333
arn:aws:s3:::lumora-lesson-uploads-111122223333/courses/c-101/lesson-1.mp4
```

Notice the empty fields. IAM is global, so its ARNs have no Region. S3 bucket ARNs have neither Region nor account, because bucket names are globally unique. The partition is `aws` for standard commercial Regions; isolated partitions such as AWS GovCloud (US) and the China Regions use different partition names. When you write code that constructs ARNs, avoid hard-coding the partition if your software might ever run outside standard Regions.

A first taste of the SDK

To make "everything is an API" concrete, here is the same Region listing from earlier, written with the AWS SDK for JavaScript v3. Chapter 4 explains clients, credentials, and error handling properly; for now, notice that the SDK call mirrors the CLI command almost exactly.

scripts/list-regions.ts

```
import { EC2Client, DescribeRegionsCommand } from "@aws-sdk/client-ec2";

const client = new EC2Client({ region: "us-east-1" });

async function main(): Promise<void> {
  const result = await client.send(new DescribeRegionsCommand({}));
  const names = (result.Regions ?? []).map((r) => r.RegionName).sort();
  console.log(names.join("\n"));
}

main().catch((err) => {
  console.error("Failed to list regions:", err);
  process.exit(1);
});
```

And the equivalent in Python with boto3:

scripts/list_regions.py

```
import boto3

ec2 = boto3.client("ec2", region_name="us-east-1")
response = ec2.describe_regions()
for name in sorted(r["RegionName"] for r in response["Regions"]):
    print(name)
```

The Shared Responsibility Model

Security on AWS is a partnership, and the **shared responsibility model** defines who does what. AWS describes the split as security *of* the cloud versus security *in* the cloud.

- **AWS is responsible for security of the cloud:** the physical data centers, the hardware, the global network, and the virtualization and software layers that run the services themselves.
- **You are responsible for security in the cloud:** your data, who has access to it, how your resources are configured, your application code, and — depending on the service — the operating system, network settings, and encryption choices.

The important nuance is that the boundary moves depending on the service you use. The more managed the service, the more AWS takes on, and the smaller your share becomes. It never shrinks to zero: no matter how managed the service, you remain responsible for your data, your identities, and your permissions.

Concern	EC2 instance	Fargate task	Lambda function	DynamoDB table
Physical security, hardware	AWS	AWS	AWS	AWS
Host OS patching	You	AWS	AWS	AWS
Language runtime updates	You	You (image)	Shared	n/a
Network access rules	You	You	You (if in VPC)	n/a (IAM)
Application code	You	You	You	n/a
IAM permissions	You	You	You	You
Data and encryption choices	You	You	You	You

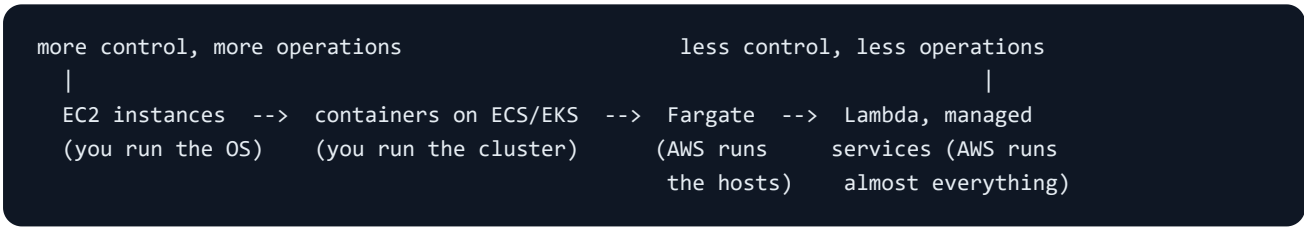
The "Shared" entry for Lambda deserves an explanation. AWS applies patches and security updates to managed runtimes, but when a runtime version reaches its end of support, you must update your function to a newer runtime yourself. Similarly, for containers on Fargate, AWS patches the underlying infrastructure, but the contents of your container image — base image, libraries, and your code — are yours to keep current.

The classic cloud security failures are customer-side configuration: a bucket opened to the public, an access key committed to a repository, an overly broad IAM policy. These are exactly the things you control, and later chapters give you habits for each: least privilege (Chapter 3), no keys in code (Chapter 4), Block Public Access (Chapter 10), and secrets and encryption (Chapter 23).

> **WARNING**
> "The service is managed" does not mean "the service is secure by my definition." A fully managed database can still be readable by the wrong people if your IAM policies or network rules allow it. Treat access configuration as code you own, review it, and test it.

Managed versus Self-Managed Services

Almost every problem on AWS can be solved at several levels of abstraction. Take "run Lumora's API." You could launch EC2 virtual machines, install Node.js, run a process manager, and put a load balancer in front. You could package the API as a container and run it on ECS with Fargate, letting AWS manage the servers. Or you could write Lambda functions behind API Gateway and not think about servers or containers at all. Each step up the ladder hands more operational work to AWS in exchange for less control.



AWS uses the phrase "undifferentiated heavy lifting" for work that every company has to do but that does not make any company special: patching operating systems, configuring database replication, rotating logs, replacing failed disks. The case for managed services is that your team's time is better spent on the things that make Lumora different — course experiences, grading logic, the study assistant — than on keeping a message broker healthy.

The case against is equally real. Managed services come with their own limits and pricing models, are harder to reproduce on a laptop, and make moving between providers harder. At very high, steady utilization, self-managed resources can sometimes cost less, although that comparison often forgets engineering time.

Question	Leans managed	Leans self-managed
Team size and ops skills	Small team, few ops specialists	Dedicated platform team
Traffic shape	Spiky or unpredictable	Steady, high utilization
Control needs	Standard configuration is fine	Need OS or kernel tuning
Portability	Accept some lock-in	Must run on several platforms

Time to market	Critical	Less critical

Lumora's default is "managed first": start with the most managed option that meets the requirements, and step down the ladder only when you hit a concrete limitation. That is a sensible default for most application teams, and it is the approach we follow in this book. When we choose containers for the grading service, it will be for a specific reason, not because containers are fashionable.

> **NOTE**
>
> Lock-in is a spectrum, not a binary. Using S3 or DynamoDB ties your data layer to AWS APIs, while running PostgreSQL on Aurora keeps your SQL portable even though the operational layer is AWS-specific. Decide deliberately which parts of Lumora you are comfortable coupling tightly, and isolate provider-specific calls behind small interfaces in your code where portability matters.

How AWS Billing Works (Conceptually)

AWS pricing changes over time and differs by Region, so this book does not quote prices. Always check the current pricing page for each service, and review the current AWS Free Tier terms, which were restructured in 2025 — do not rely on older blog posts or courses for what is free. What does not change quickly, and what you need as a developer, is an understanding of *what drives cost*. If you know the cost drivers, you can predict how a design choice will show up on the bill.

The common cost dimensions

- **Compute time.** EC2 instances are billed for the time they run, based on instance type. Fargate is billed for the vCPU and memory your tasks request, for as long as they run. Lambda is billed by the number of requests plus compute duration measured in GB-seconds (memory configured multiplied by execution time).
- **Requests.** Many services charge per request or per million requests: API Gateway, S3 GET and PUT requests, SQS messages, and DynamoDB reads and writes in on-demand mode.
- **Storage.** Data at rest is usually billed per GB-month: S3 objects, EBS volumes, database storage, backups and snapshots, and log data retained in CloudWatch Logs.
- **Data transfer.** Data moving *out* of AWS to the internet is charged, and so is some traffic between Regions and between Availability Zones. Data transfer into AWS from the internet is generally not charged. Data transfer is frequently the line item that surprises teams, so learn its rules for any service that moves significant data, such as Lumora's lesson videos.
- **Provisioned capacity.** Some resources cost money simply by existing, whether or not you use them: a running EC2 instance, a provisioned database, a NAT gateway, DynamoDB provisioned capacity, or provisioned concurrency for Lambda.

The distinction between **usage-based** and **provisioned** costs is the most useful one in design discussions. Usage-based costs scale with traffic, which suits a young product like Lumora. Provisioned costs exist even at zero traffic, which is why a forgotten test database or NAT gateway can quietly add up for months.

Pricing models

On-Demand pricing (no commitment) is the default starting point. **Savings Plans** and **Reserved Instances** lower rates in exchange for committing to consistent usage over a term. **Spot Instances** use spare EC2 capacity at a discount but can be reclaimed at short notice, so they suit interruptible work. Chapter 24 covers how to decide.

Estimating Lumora's costs without prices

You can reason about costs by modeling usage. For `courses-api`, estimate monthly requests, Lambda duration and memory, DynamoDB reads and writes per request, and response size. Plug those assumptions into the **AWS Pricing Calculator** for your Region. The value is less the final figure than discovering which assumption dominates. For a video-heavy platform, you will likely find that storing and delivering video matters much more than the API, which tells you where optimization effort will pay off.

```
Cost model sketch: courses-api (fill in your own assumptions)

  monthly API requests          = R
  Lambda memory (GB)            = M
  average duration (seconds)    = D
  Lambda compute (GB-seconds)   = R x M x D
  DynamoDB reads per request    = r      --> total reads  = R x r
  DynamoDB writes per request   = w      --> total writes = R x w
  average response size (KB)    = S      --> data out     = R x S

  Look up the current unit prices for your Region, multiply, and compare
  which line dominates. Revisit the model when real metrics arrive.
```

Guardrails for your bill

Because costs accrue continuously, find out about spending early. AWS Budgets alerts you when actual or forecasted spend crosses a threshold, Cost Anomaly Detection flags unusual patterns, and Cost Explorer breaks spending down by service, Region, and tags. We set up a budget and tagging in Chapter 2.

> **WARNING**
>
> Budget alerts notify you; they do not stop spending by themselves. AWS generally does not shut down your resources when you exceed a budget. If a script accidentally launches large resources, you will still pay for them until you delete them. Clean up after experiments, and treat unexpected alerts as incidents.

Designing for Failure

At cloud scale, individual components fail routinely. Disks die, instances are retired, networks hiccup, and throttling happens. Good cloud applications assume these failures and are designed to tolerate them.

For a developer, this translates into a handful of concrete habits that appear throughout the book:

- **Spread across AZs.** Use services that are multi-AZ by design, and configure multi-AZ where it is optional for anything important.
- **Retry with backoff.** Transient errors and throttling are normal. The AWS SDKs retry many errors automatically, and your own code should retry thoughtfully (Chapter 4).
- **Make operations idempotent.** If a message is delivered twice or a request is retried, the result should be the same. This is essential for queues, events, and payments (Chapters 7 and 16).
- **Decouple with queues and events.** When the grading service is slow, quiz submissions should wait in a queue rather than fail (Chapter 16).
- **Treat servers as replaceable.** Anything that can be recreated from code should be. Infrastructure as code (Chapters 19 and 20) makes this practical.
- **Observe everything.** You cannot fix what you cannot see. Structured logs, metrics, alarms, and traces are part of the product, not an afterthought (Chapter 22).

These habits underpin the **AWS Well-Architected Framework** and its six pillars: operational excellence, security, reliability, performance efficiency, cost optimization, and sustainability (Chapter 24).

Quotas, Limits, and the Shape of a Service

Every service has **quotas** (historically called limits). Some are hard limits built into the service design: a Lambda function can run for at most 15 minutes, a single S3 object can be up to 5 TB, and a DynamoDB item can be up to 400 KB. Others are per-account, per-Region defaults that exist to protect you and AWS from runaway usage and can often be raised through the **Service Quotas** console or API.

Quotas shape design more than most newcomers expect. The 400 KB item limit is why Lumora stores lesson transcripts in S3 and only metadata in DynamoDB. The 15-minute Lambda limit is one reason video processing is orchestrated with Step Functions and dedicated media services instead of a single function. Check adjustable quotas for your account before a launch.

```
# See the quotas that apply to Lambda in your account and Region
aws service-quotas list-service-quotas --service-code lambda \
  --query "Quotas[].[QuotaName,Value,Adjustable]" --output table
```