MODULE 2 · STUDY NOTES

# Introduction to Developing on AWS

*The systems development lifecycle, the steps to set up an AWS development environment, working with the AWS SDKs, and the Amazon Q Developer coding assistant*

Before Sofía can build a web presence for the café, she has to decide *where and how* she will write and run her code — and that is exactly what this module sets up. It begins with the **systems development lifecycle (SDLC)**, the model that frames how software moves from idea to maintenance, along with the methodologies (waterfall and agile) and the development phases (code, build, test, deploy, maintain) that structure the work. It then walks the concrete **four steps to start developing on AWS** — create an account, set up permissions with IAM, install a development environment, and interact with AWS services — and drills into the two programmatic front doors that matter most to a developer: the **AWS CLI** (and browser-based **CloudShell**) and the **AWS SDKs**, including how the SDKs make requests, expose client and resource APIs, target Regions, and report errors. Finally it introduces **Amazon Q Developer**, the generative AI coding assistant that supports every phase of the lifecycle.

LEARNING OBJECTIVES

- Recognize the systems development lifecycle (SDLC)
- Describe how to start developing on AWS
- Identify the benefits of using an integrated development environment (IDE)
- Indicate how to use AWS SDKs
- Identify the benefits of using Amazon Q Developer

## 2.1 Introduction

This module follows **Sofía**, who wants to start developing a web presence for the café. She already has **Python** development skills and is learning to build solutions in the cloud; before she writes any code, she needs to decide on a **development environment** to develop and run it. The module gives her — and you — the tools and vocabulary to make that decision.

The course builds one application across its labs, and the parts relevant to this module are the developer's entry points. From the **AWS Management Console** you can launch **AWS CloudShell**, and from CloudShell you can use the **AWS CLI** or the **AWS SDK for Python (Boto3)** to interact with AWS resources — for example, an **Amazon S3** bucket. Those same tools reappear throughout the module.

WHAT THIS MODULE COVERS

- The **systems development lifecycle (SDLC)** and how software is developed
- The **steps to get started** developing on AWS
- The **fundamentals of working with the AWS SDKs**
- An **introduction to Amazon Q Developer**
- A demonstration of **installing the AWS CLI** and a lab exploring **AWS CloudShell and an IDE**

## 2.2 Systems development lifecycle

Software development and delivery is hard: developers must quickly build solutions that are **secure**, **reliable**, and able to **scale** to meet customer demand — all while protecting the security and integrity of the system. Enterprises have to bridge the gap between the **stability of their operations** and **rapid feature development**, delivering software *quickly, securely, and reliably* with **zero tolerance for outages**. Bridging that gap takes a methodology.

The **SDLC** is a conceptual model used in project management. It describes the stages of an information-system development project, from an initial feasibility study through the maintenance of the finished application. A typical SDLC follows the steps **plan → define → design → develop → deploy → maintain**. This course focuses primarily on the **Develop** phase and touches briefly on **Deploy**. In the Develop phase, the new system is built: new components and programs are obtained and installed, users are trained, and every aspect of performance is tested — with bugs fixed and adjustments made as needed.

| WATERFALL (TRADITIONAL) | AGILE |
|---|---|
| *the classic, sequential approach*<br>▪ Sequential development method<br>▪ Each phase has distinct goals completed before the next begins<br>▪ Teams may wait months for customer feedback — often until commercialization<br>▪ Best when requirements are stable and well understood | *fast-paced and iterative*<br>▪ Newer, fast-paced, iterative framework<br>▪ Push work to customers quickly to collect feedback and improve each iteration<br>▪ Work happens in **sprints** — short cycles, typically **1–4 weeks**<br>▪ Introduces **MVP**, release candidate, velocity; methods include **Scrum**, Crystal, DSDM |

Because agile requirements change frequently, **developing on AWS** makes those changes easier to accommodate efficiently. Regardless of methodology, the development work itself is organized into five major phases.

**TABLE 2.1** The five major phases of software development

| PHASE | SDLC STAGE | KEY ACTIVITIES |
|---|---|---|
| **Code** | Develop | Write source code and check changes into a source repository (e.g., a Git repository); peer-review with code reviews or pair programming |
| **Build** | Develop | Compile the code; run unit tests, style checkers, and code metrics; create container images |
| **Test** | Develop | Deploy to a production-like environment for integration testing with other systems, load testing, UI testing, and penetration testing |
| **Deploy** | Deploy | Release code to production, reducing risk and minimizing the impact of a bad change |
| **Maintain** | Maintain | Monitor code in production to quickly detect unusual activity or errors |

Note that **Code** (source), **Build**, and **Test** all fall under the **Develop** stage of the SDLC; **Deploy** and **Maintain** follow. Each phase raises confidence that the code will behave as intended when released, and **each phase can be automated** independently — without automating the entire release process.

## 2.3 Steps to get started developing on AWS

Before you can develop applications on AWS, a few things must be in place. You need an **AWS account** and an **IAM user** (which involves setting up permissions to access specific services and operations); you must **install a development environment** for your language of choice; and you must install the **AWS SDK** for that language — or the **AWS CLI** — so you can interact with AWS services. The four steps below organize the setup. (Permissions are covered in depth in a later module.)

> **FOUR STEPS TO GET STARTED**
>
> - **Create an AWS account** — your entry point to AWS
> - **Set up AWS permissions** with **IAM**
> - **Install the development environment** — your language plus an IDE and AWS IDE Toolkit
> - **Interact with AWS services** using the **AWS SDKs** and the **AWS CLI**

To create an account, open `aws.amazon.com` and choose **Create an AWS Account** (if you have signed in recently, choose **Sign in to the Console** instead). You select whether the account is **personal** or **professional** — both have the **same features**, and personal is fine for learning. For a professional account, use the **company's phone number and an organizational email**, because a personal phone and email can make the account less secure. Make sure you can access the email you use (to receive the confirmation message), read and accept the **AWS Customer Agreement**, and choose **Create Account and Continue**.

When you first create an account you get a single sign-on identity with **complete access to all services and resources** — the **AWS account root user**, accessed with the email and password used to create the account. A best practice is to **not use the root account unless required**; instead, create an **IAM user** that works with access credentials. **IAM (Identity and Access Management)** securely controls access to AWS resources: it governs **who** can use your resources (**authentication**) and **what** they can use and **in what ways** (**authorization**). With IAM you create and manage **users, groups, roles, and policies**, and it also enables **identity federation** between a corporate directory and AWS services.

> **IAM IN PRACTICE – ONE DEVELOPER**
>
> Suppose you are responsible for an organization's users. You create an IAM **user** for each developer — say **John Doe**, who has AWS credentials — and place John's user in the **Developer group**. You define a **role**, a trusted AWS entity carrying the Developer permissions policies, and associate a **policy** that grants access to specific services and operations. Developers **assume the role** to get the permissions they need; when requirements change, you update the **policy** on the role, and every developer who assumes it automatically gets the updated permissions.

> **INSTALL THE DEVELOPMENT ENVIRONMENT**
>
> - Install your **programming language** of choice and an **integrated development environment (IDE)** to write, run, debug, and deploy applications
> - AWS provides **IDE Toolkits** for popular IDEs, including **Eclipse**, **IntelliJ IDEA**, **PyCharm**, **Microsoft Visual Studio**, **Visual Studio Code**, **Azure DevOps**, and **Rider**
> - Follow each technology vendor's instructions to install the platform and IDE (see “Tools to Build on AWS”)

You can interact with AWS through the **Management Console** or **programmatically**. Under the hood, **all AWS services are managed through a common, REST-like API**, and AWS provides an API — with a service **endpoint** — for each service. You can reach those APIs **four interchangeable ways**: call them **directly**; use the **Management Console** (a rich

graphical implementation of the API calls, though a brand-new feature is occasionally not yet in the console); use the **AWS CLI**; or use an **AWS SDK**. The CLI and SDKs add flexibility — you can build your own tools and scripts. For example, you might **create an EC2 instance through an SDK call**, **query it with a CLI command** (`aws ec2 describe-instances`), and **terminate it from the console**.

**TABLE 2.2** Command line tools for calling AWS services

| TOOL | WHAT IT IS |
|---|---|
| **AWS CLI** | Open source tool to access AWS from the command line on Windows, macOS, or Linux/Unix — functionality equivalent to the console |
| **AWS Tools for PowerShell** | PowerShell modules (built on the SDK for .NET) that script operations on AWS resources from the PowerShell command line |
| **AWS SAM Local** | An AWS CLI tool for local development and testing of serverless applications (AWS Serverless Application Model) |
| **AWS Amplify** | A framework to create, configure, and deploy scalable mobile and web apps; provisions and manages the backend and automates releases |

AWS CLI commands follow a fixed shape. Every command begins with `aws`, followed by the **service command** (the top-level service, e.g., `ec2`), the **operation subcommand** (the action, e.g., `run-instances`), any **parameters** (arguments the operation needs — argument names are prefixed with `--`, such as an AMI ID), and optional **options** (e.g., `--query` to limit the response to just the new instance ID).

> AWS CLOUDSHELL
>
> - A browser-based CLI available directly in the **AWS Management Console**
> - **Pre-authenticated** — it inherits your console credentials, so there are **no key files to manage**
> - Runs on a fully managed **Amazon Linux 2** instance with the latest tools — no patching or updating
> - **Pre-installed** with the AWS CLI, SDKs, and other tools
> - **No cost**, with **1 GB of persistent storage per Region**; customizable per Region for scripts, files, and preferences

The module includes a **demonstration** of installing the AWS CLI, and **Lab 2.1: Exploring AWS CloudShell and an IDE** has you use the console and CloudShell to interact with an **Amazon S3** bucket and explore an IDE — putting these getting-started tools to work.

## 2.4 Fundamentals of working with the AWS SDKs

The **AWS SDKs** are packages, available in many programming languages and platforms, that let your **application code call AWS services**. Say you are building a music-sharing application that needs to upload a user's files to an **Amazon S3** bucket — your code can use the AWS SDK to upload the files programmatically, without hand-writing raw API requests.

**TABLE 2.3** The AWS SDK families and where they run

| SDK FAMILY | AVAILABLE FOR |
|---|---|
| **AWS SDK** | C++, Go, Java, JavaScript (in the browser and in Node.js), .NET and Xamarin, PHP, Python (Boto3), Ruby |
| **AWS Mobile SDK** | Android, iOS, .NET and Xamarin, Unity |
| **AWS IoT Device SDK** | Arduino Yún, C++, Embedded C, Java, JavaScript, Python |

#### HOW AN SDK CALL REACHES A SERVICE

- You write an application using an AWS SDK in your language of choice
- Each AWS SDK provides one or more **APIs** for working with AWS services
- The SDK constructs an **HTTP(S) request** for the service API and sends it to the service **endpoint** (the URL entry point for the web service)
- The service performs the request and sends a **response**
- The SDK processes the response and propagates it back to your application

AWS SDKs offer **two levels of API**. The **service client API** is low-level: it provides client classes that are a **direct, 1:1 mapping** of a service's API, with **one method per operation** and objects representing the request parameters and the response data. It gives you **full control** over the requests you make. The **resource API** is a **higher-level, object-oriented abstraction**: instead of one client exposing an entire service, it provides **one class per conceptual resource** (such as an S3 bucket or object), with methods to act on the resource and attributes to read from it. Not every SDK offers a resource API — the **SDK for Python (Boto3)** does.

#### CLIENT VS. RESOURCE API – LISTING S3 OBJECTS IN PYTHON

With the **client API**, you create `boto3.client('s3')`, call `list_objects_v2(Bucket='DOC-EXAMPLE-BUCKET')`, and iterate the returned `dict` over `response['Contents']`, reading each object's `Key` and `LastModified`. With the **resource API**, you create `boto3.resource('s3')`, get a `Bucket('DOC-EXAMPLE-BUCKET')`, and iterate `bucket.objects.all()`, reading each `object.key` and `object.last_modified`. Both list the same bucket; the resource API just reads more like plain objects.

AWS infrastructure is organized into **Regions** (physical locations, each with multiple Availability Zones), and your application accesses services that reside in a specific Region. How you set the Region depends on the SDK. You can specify it **when you instantiate the service client** — for example, `boto3.client('dynamodb', region_name='us-east-1')`. With the SDKs for **Java** and **.NET** you must create a **separate client instance per Region**; other SDKs such as **Boto3** can instead use the **default Region** set in `~/.aws/config` (for example, `region=us-west-2`).

When you call a service, it returns an **HTTP status code**. A successful request returns **HTTP 200**; an unsuccessful one returns an **error response**, and you use the error code to decide how to react. An **HTTP 400-series (client) error** must be **handled in your application** — for example, Amazon S3 returns a **404** if a bucket does not exist, so your app could create the bucket and then retry the operation. An **HTTP 500-series (server) error** indicates an internal server error that you should **retry**. Every AWS SDK implements **automatic retry logic** with a **configurable maximum** number of attempts, plus an **exponential backoff** algorithm that waits progressively longer after each failed attempt.

**TABLE 2.4** Elements of an AWS error response

| ELEMENT | MEANING |
|---|---|
| **Error** | Container for all error elements |
| **Code** | String that uniquely identifies the error condition, read by programs that handle errors by type (e.g., `NoSuchKey`) |
| **Message** | Generic, human-readable description of the error condition in English |
| **Resource** | The resource that is involved in the error |
| **RequestID** | ID of the request that is associated with the error |

**TABLE 2.5** How each SDK signals errors

| SDK | HOW IT SIGNALS ERRORS |
|---|---|
| **Java** | Throws unchecked `AmazonServiceException` (the service received the request but returned an error, with HTTP status, AWS error code, message, and request ID) or `AmazonClientException` (a problem inside the client, e.g., no network connection) |
| **.NET and Xamarin** | Throws `Amazon.Runtime.AmazonServiceException` and `Amazon.Runtime.AmazonClientException` — similar to the Java errors |
| **Python** | Throws `botocore.exceptions.ClientError` |
| **JavaScript** | Passes errors to an asynchronous callback with the signature `function(error, data) { ... }` |

> **COMMON PITFALL**
>
> Match the error to the response: a **400-series** error is a **client** problem you fix in code; a **500-series** error is a **server** problem you **retry** (with exponential backoff). Reversing the two — retrying a 400 that will always fail, or giving up on a transient 500 — is a common mistake.

## 2.5 Introduction to Amazon Q Developer

AWS introduced **Amazon Q Developer** to ease the challenges of writing code. It is a **generative AI-powered coding assistant** for developers and IT professionals — usable even by those **without extensive experience** — trained on years of high-quality **AWS examples and documentation**, and optionally on **your company's own code and systems**. It can **chat about code**, provide **in-line completions**, **generate new code**, and **scan code for security vulnerabilities**, as well as make **upgrades and improvements** such as language-version updates, debugging, and optimizations. You can describe an application in **natural language** and have Amazon Q generate and suggest the code — a **no-code approach** well suited to simple web applications, automating business processes, and building prototypes and proofs of concept. It is **secure and private by design**.

Amazon Q Developer supports you across **every phase of the SDLC** — in the AWS Management Console, the IDE, and the command line.

**TABLE 2.6** Amazon Q Developer across the SDLC

| SDLC PHASE | HOW AMAZON Q DEVELOPER HELPS |
|---|---|
| **Plan** | Ask questions and get referenceable, contextual guidance; explain code with conversational coding; learn about the Well-Architected Framework |
| **Create** | In-line code recommendations for multiple languages in the IDE or CLI; implement features from comments or code prompts; chat in the IDE |
| **Test & secure** | Generate unit tests; scan project code for security vulnerabilities and get remediation suggestions earlier in the cycle |
| **Operate** | Troubleshoot errors (e.g., in Lambda, EC2, ECS); troubleshoot network connectivity with VPC Reachability Analyzer |
| **Maintain & modernize** | Upgrade projects to more up-to-date language versions with the Amazon Q Developer Agent for code transformation |

#### AMAZON Q IMPLEMENTS A NEW FEATURE – THE WORKFLOW

In your IDE, you can have Amazon Q add a feature: **(1)** choose the Amazon Q icon and **(2)** send a prompt describing the feature; **(3)** Amazon Q analyzes the whole project and returns a **step-by-step plan**; **(4)** choose **Generate code**; **(5) review the changes** in a file-by-file **diff view** (removed code in red, added code in green — a brand-new file is compared against an empty file, so all of it shows as additions); **(6)** choose **Insert code** (or regenerate if you don't like the suggestion); and **(7)** run a **validation** — a clean install ends with **BUILD SUCCESS**, and the code is ready to commit and deploy for testing.

#### SAMPLE EXAM QUESTION, WORKED

*A developer needs a code assistant to help write code for a new application, and does not have much experience with coding or AWS. Which tool meets this use case?* The key phrase is a **code assistant** for someone with limited coding and AWS experience — that describes **Amazon Q Developer** (the correct answer: use Amazon Q Developer to write the code and use the chat feature when needed). The AWS CLI, a GitHub repository of pre-built code, and the AWS SDK all still require the developer to write and assemble the application code themselves.

#### COMMON PITFALL

Amazon Q Developer is not just an autocomplete for the **Create** phase. It also helps you **Plan** (contextual guidance, code explanation), **Test & secure** (generate unit tests, find and fix security vulnerabilities), **Operate** (troubleshoot errors and network reachability), and **Maintain & modernize** (upgrade code with the Q Developer Agent for code transformation).

## Module summary

- The **SDLC** frames software work as **plan → define → design → develop → deploy → maintain**; this course focuses on **Develop** (and briefly Deploy). The overarching challenge is delivering software quickly, securely, and reliably with zero tolerance for outages.
- **Waterfall** is sequential (finish each phase before the next); **agile** is iterative, shipping in short **sprints** of 1–4 weeks to gather continuous feedback. Developing on AWS makes changing requirements easier to absorb.

- Development runs through five phases — **Code, Build, Test, Deploy, Maintain** — with Code, Build, and Test under the Develop stage; each phase can be automated independently.
- Four steps get you started on AWS: **create an account**, **set up IAM permissions**, **install a development environment** (language plus an IDE and AWS IDE Toolkit), and **interact with AWS services** via the SDKs and CLI.
- **IAM** controls **authentication** (who) and **authorization** (what and how) through users, groups, roles, and policies; avoid the all-powerful **root user** for daily work in favor of IAM users and assumed roles.
- All AWS services share a common **REST-like API** reachable four interchangeable ways — **directly, Console, CLI, SDK** — each service having its own **endpoint**. The **AWS CLI** (and browser-based **CloudShell**, pre-authenticated and free with 1 GB per Region) drives services from the command line.
- The **AWS SDKs** let application code call AWS services; a call builds an HTTP(S) request to the service endpoint and returns a response. SDKs expose a low-level **service client API** (full control) and, in some SDKs like **Boto3**, a higher-level object-oriented **resource API**.
- Set the **Region** when instantiating a client (a separate client per Region for Java and .NET, or the default from `~/.aws/config` for Boto3). Handle errors by code: **400 = client error, fix in the app; 500 = server error, retry** with automatic **exponential backoff**.
- **Amazon Q Developer** is a generative AI coding assistant that spans the whole SDLC — planning, in-line code creation, unit tests and security scanning, operational troubleshooting, and code modernization.

## Review questions

**1. Which phase of the SDLC does this course focus on, and what are the five major phases of software development?**

*Answer:* The course focuses on the **Develop** phase (and briefly Deploy). The five phases of software development are **Code, Build, Test, Deploy, and Maintain** — with Code, Build, and Test falling under the Develop stage.

**2. How does the waterfall methodology differ from agile, and what is a sprint?**

*Answer:* Waterfall is a **sequential** method where each phase must finish before the next begins, so customer feedback can take months. Agile is **iterative**, pushing work to customers quickly to gather feedback; a **sprint** is a short agile iteration, typically 1–4 weeks.

**3. List the four steps to get started developing on AWS, in order.**

*Answer:* 1) Create an AWS account. 2) Set up permissions with IAM. 3) Install the development environment (your language plus an IDE and AWS IDE Toolkit). 4) Interact with AWS services using the AWS SDKs and the AWS CLI.

**4. What two things does IAM control, and why should you avoid the root user for everyday work?**

*Answer:* IAM controls **authentication** (who can use your AWS resources) and **authorization** (what they can use and in what ways). The **root user** has complete access to everything, so best practice is to avoid it for daily work and use IAM users and roles with scoped permissions instead.

**5. What are the four ways to interact with AWS services, and what do all AWS services have in common?**

*Answer:* You can interact **directly** with the API, through the **AWS Management Console**, through the **AWS CLI**, or through an **AWS SDK** — interchangeably. All AWS services are managed through a common **REST-like API**, and each service has its own **endpoint**.

**6. What is the difference between the service client API and the resource API?**

*Answer:* The **service client API** is low-level, with one method per service operation and a direct 1:1 mapping of the service API, giving full control. The **resource API** is a higher-level, object-oriented abstraction with one class per conceptual resource (bucket, object). Not every SDK offers a resource API; **Boto3** does.

---

**7. How should an application respond to an HTTP 400-series error versus a 500-series error?**

*Answer:* A **400-series (client) error** must be handled in the application (for example, create a missing S3 bucket, then retry). A **500-series (server) error** is an internal server error you should **retry** — AWS SDKs retry automatically using an **exponential backoff** algorithm with a configurable maximum number of attempts.

---

**8. What is Amazon Q Developer, and which SDLC phases does it support?**

*Answer:* Amazon Q Developer is a **generative AI-powered coding assistant** that generates and explains code and scans it for security vulnerabilities. It supports **Plan, Create, Test & secure, Operate, and Maintain & modernize** across the software development lifecycle.

---