MODULE 7 · STUDY NOTES

Developing Event-Driven Serverless Solutions

How AWS Lambda runs your code without servers — event sources and invocation models, the two kinds of permissions Lambda needs, authoring and configuring functions, deployment, and the developer tools that monitor and trace them

In the café project, Sofía is ready to launch the dynamic version of the website: she needs to replace the mock API endpoints with real ones so the application can read and write data. The bridge between the REST API in Amazon API Gateway and the data in Amazon DynamoDB is an **AWS Lambda function** — code that runs only when an endpoint is called, with no server to stand up. That is the heart of this module. Lambda is AWS's serverless compute service: you bring code, wire it to **event sources**, and Lambda runs it on demand, scales it automatically, and bills only for the compute time consumed. The module builds the whole mental model — where serverless sits in the evolution from on-premises servers to containers; what a Lambda function is made of; the push and pull ways events invoke it; the two kinds of IAM permissions it depends on; how to author, configure, and tune the handler; how to deploy with .zip archives, container images, versions, aliases, and layers; and how CloudWatch and AWS X-Ray let a developer monitor, log, and trace functions across a distributed application.

LEARNING OBJECTIVES

- Explain serverless computing
- Describe how AWS Lambda works
- Recognize Lambda invocation models
- Identify how to use AWS Identity and Access Management (IAM) to grant Lambda permissions
- Indicate the steps to author and configure a Lambda function
- Explain how to deploy Lambda functions
- Identify why AWS X-Ray is a critical developer tool
- Create Lambda functions

7.1 Introducing serverless computing

Cloud computing lets you **abstract the infrastructure layer** so you can focus on business logic instead of operations. Serverless computing takes that idea furthest: it eliminates infrastructure-management tasks such as server and cluster provisioning, patching, OS maintenance, and capacity planning. You do not manage instances, operating systems, or servers — everything needed to run and scale your application with high availability is handled for you. And because your code runs only when it is needed, **you don't pay for infrastructure while the code isn't running**. Reduced overhead also means you can experiment and innovate faster.

TABLE 7.1 The compute spectrum – more is handled for you at each step

MODEL	WHAT YOU STILL DEAL WITH	WHAT YOU GAIN
On premises	Heavy capital expense, guessed capacity, months to deploy, years of maintenance	Full control (and a low innovation factor)
Virtual servers	Provisioning and maintaining instances and the OS	Hardware independence, elastic scale, faster provisioning
Containers	Building images; higher-level packaging	Platform independence, higher utilization, isolation and sandboxing
Serverless	Only building, deploying, and monitoring your application	No OS instances, flexible scaling, pay per usage, built-in fault tolerance, zero maintenance

With servers you configure instances, patch the OS, set up scaling and load balancing, and monitor everything; with serverless, that list collapses to just **building/deploying** and **monitoring/maintaining** your application. Modern serverless applications follow a builder's pattern of *small pieces, loosely joined*: serverless compute with **event-driven** patterns, **purpose-built** serverless data stores, **managed services for integration** (API proxy, messaging, orchestration, with prebuilt Lambda integrations), and **automation by using code** (infrastructure as code and deployment frameworks).

MONOLITH – DOES EVERYTHING	MICROSERVICE – DOES ONE THING
easy to start, hard to grow	*higher initial effort, then agility*
▪ One artifact to build, test, and deploy — simplest at first ▪ A single technology stack everyone must use ▪ A single database whose most demanding use case drives cost and capacity ▪ Shared release pipeline causes merge friction; every feature adds coordination overhead	▪ Scale each service independently ▪ Choose the best technology stack and data store per service ▪ Deploy changes independently, as often as desired ▪ Focus on and innovate within a specific service

Moving to microservices takes real up-front effort, but after the learning curve, agility accelerates feature releases and reduces the effort to keep them running. Serverless is an efficient way to build this style of application: serverless compute handles scaling and high availability, purpose-built serverless databases match capacity to each service's data, managed integration services decouple components through asynchronous event-driven patterns, and serverless deployment frameworks automate the pipeline.

> **COMMON PITFALL**
>
> "Serverless" does not mean "no servers." Servers still run your code — you just never provision, patch, scale, or maintain them, and you don't pay for them while your code is idle. The value is the removal of undifferentiated operational lifting, not the disappearance of the underlying infrastructure.

7.2 Introducing Lambda

AWS Lambda is the compute service for serverless. You use it to run code without provisioning or managing servers: it invokes your code in response to events you configure, scales automatically based on demand, and includes built-in monitoring and logging with **Amazon CloudWatch**. To use Lambda you upload existing code (or write it in the Lambda editor), set your functions to run when events occur in other AWS services, at HTTP endpoints, or as part of in-app activity, and let Lambda run the code only when it is activated — using only the compute resources needed. You pay only for the compute time you use.

LAMBDA FEATURES

- **Bring your own code** — write in Node.js, Java, Python, C#, Go, or Ruby (plus custom runtimes and libraries); code isn't tightly coupled to AWS, so it ports in and out easily.
- **Integrates with and extends other AWS services** — from inside a function you can call an AWS SDK or invoke a third-party API.
- **Flexible resource and concurrency models** — you set memory; AWS handles CPU, network, and I/O throughput, scaling in response to events.
- **Flexible permissions model** — IAM grants fine-grained control over who can invoke functions and what functions can access.
- **Built-in availability and fault tolerance** — high availability is integrated, with no extra configuration.
- **No paying for idle** — functions run only when invoked, so you don't pay for idle capacity.

Common use cases show the range. A **web application** serves front-end code from S3 and calls a REST API through **API Gateway**, which invokes Lambda to read **DynamoDB** and return data; **real-time stream processing** feeds a stream into **Kinesis**, whose events invoke Lambda to compute trend data. Other cases: serverless **backends** for web, mobile, IoT, and third-party APIs; **data processing** on changes in data or state (S3, DynamoDB, Kinesis, SNS, and CloudWatch can invoke Lambda, and Step Functions can orchestrate it); **chatbots** with Amazon Lex; custom **Alexa** skills; and **IT automation** on events or a schedule. Almost anything you'd do with servers, you can do with Lambda.

WHAT MAKES UP A LAMBDA FUNCTION

- **Access permissions** — an execution role defining what the function can do.
- **Initiating events** — the event sources configured to invoke it.
- **Runtime and deployment package** — a chosen runtime plus your application code and its dependencies and libraries.
- **Configuration** — parameters such as memory, timeout, and concurrency.
- When invoked, Lambda provides a **runtime environment** based on the runtime and configuration you selected.

Lambda functions run in **on-demand, ephemeral environments** — each is initialized on demand, lasts a brief time, and is then removed; Lambda creates and tears them down, and you don't control them directly. **Concurrency** is the number of function invocations running at one time, and providing environments on demand is how Lambda scales horizontally. Factors that influence concurrency: **reserved concurrency** (an optional per-function value that reserves a slice of the Regional quota and caps the function's maximum concurrent instances), the **Regional quota** (the total concurrent invocations allowed across all functions in an account per Region — a soft account quota), the **burst quota** (an unmodifiable per-Region limit that keeps concurrency from spiking too fast), the **request rate and function duration**, and the **event source** (Lambda manages concurrency differently by event type).

COMMON PITFALL

Because environments are temporary and Lambda-controlled, never assume state persists between invocations. Design functions to be **stateless** — but do take advantage of environment reuse when it happens: initialize expensive dependencies (database or HTTP connections, configuration) once and reference them locally on later invocations to improve performance.

7.3 Invoking Lambda functions

Every event source invokes a function using one of two models. In the **push model**, an event source directly invokes the function when the event occurs. In the **pull (polling) model**, an event source puts information into a stream or queue; Lambda polls it and, when it detects records, invokes your function with **batches of records**. Push events split further into synchronous and asynchronous invocation, and the invocation type is fixed per AWS service — when you use a service as a trigger, you have no control over which type it uses.

Synchronous invocation makes the caller wait for the response and returns a `200` acknowledgement, with **no built-in retry** — your application code owns the retry strategy (an API Gateway client, for instance, expects an immediate answer). Synchronous sources include API Gateway, ELB, Cognito, Lex, Alexa, CloudFront, CloudFormation, and Kinesis Data Firehose. **Asynchronous** invocation has Lambda **queue** the event and return a `202` acknowledgement; the caller isn't aware of what happens afterward, Lambda **retries twice** on error, and failed events can go to a **dead-letter queue**. Asynchronous sources include S3, SNS, SES, CloudFormation, CloudWatch Logs, CloudWatch Events, and AWS Config.

In the **polling model**, Lambda uses **event source mappings** to read from a stream or queue — the Lambda service itself extracts the data and invokes the function. **Stream-based** sources (DynamoDB Streams, Kinesis Data Streams) organize records into **shards**; Lambda polls for records and attempts to invoke the function, and if a failure occurs it will not read new records from that shard until the failed batch expires or succeeds — so an erroring record can **block processing** (records retry until success or a **24-hour** expiration). You can configure error handling to bypass failing records and send them to an on-failure destination for offline processing. **Queue-based** sources (Amazon SQS) return a failed or timed-out message to the queue; Lambda keeps retrying until the message succeeds or the retry limit or retention period is reached, and continually failing messages can go to a dead-letter queue.

WORKED EXAMPLE – INVOKING A FUNCTION FROM THE CLI

To invoke synchronously, run `aws lambda invoke --function-name my-function ... response.json`; the acknowledgement is a `200` status. To invoke asynchronously, add `--invocation-type Event`; the acknowledgement is a `202` status, meaning the request was received. In both cases you use the same invoke operation and specify the invocation type — but when an AWS service is the trigger, that type is predetermined for you. You create the pull-model link with the `CreateEventSourceMapping` operation, which tells Lambda to read items from the source and invoke the function. Polling itself is free: you are charged only if actions result from a poll.

COMMON PITFALL

Don't confuse the retry behaviors. Synchronous = no built-in retry. Asynchronous = two automatic retries plus an optional dead-letter queue. Streams = retry until success or 24-hour expiry, with the risk of a blocked shard. Queues = message returns and retries to a configurable limit, then a dead-letter queue. Exam scenarios hinge on matching the source to the right failure handling.

7.4 Setting permissions for Lambda

Lambda requires **two types of permissions**, both handled through IAM. First, **event sources need permission to invoke** a function. Second, **functions need permission to interact** with other AWS services and resources. Keeping these two straight — inbound versus outbound — is the core of Lambda security on the exam.

TABLE 7.2 The two permissions Lambda depends on

PERMISSION	DIRECTION	WHAT IT GRANTS
Resource-based (function) policy	Inbound — who can invoke	Tells Lambda which push event sources may take the `lambda:InvokeFunction` action; also enables cross-account invocation
Execution role	Outbound — what the function can do	An IAM policy defining allowed actions on other services, plus a trust policy letting Lambda assume the role

An **IAM resource policy** (also called a function policy) tells the Lambda service which push event sources have permission to invoke the function, and makes it easy to grant access **across AWS accounts** — for example, allowing an S3 bucket in account A to invoke a function in account B. In an example resource policy, the `Effect` is `Allow`, the `Principal` is the service `s3.amazonaws.com`, the `Action` is `lambda:InvokeFunction`, and a condition restricts the source to a specific bucket ARN.

The **Lambda execution role** specifies what the function is permitted to do — you select or create it when you create the function. Two policies attach to it: an **IAM policy** that defines the actions the function can take on another service or resource (such as writing to a DynamoDB table), and a **trust policy** that allows the Lambda service to assume the role. Policies aren't attached directly to a function; they attach to a role that Lambda assumes. To grant Lambda the `AssumeRole` action you must have permission for the `iam:PassRole` action. **Polling event sources also need the execution role** so the function can read from the queue or stream.

> **WORKED EXAMPLE – EXECUTION-ROLE POLICY LINES AND ADDING PERMISSION**
>
> A typical execution-role **IAM policy** allows CloudWatch Logs actions such as `logs:CreateLogGroup`, `logs:CreateLogStream`, and `logs:PutLogEvents`, so the function can create a log group and stream and write log events. The matching **trust policy** allows the principal `lambda.amazonaws.com` to take the `sts:AssumeRole` action, letting Lambda invoke the function on your behalf. To grant a service permission to invoke a function from the CLI, run `aws lambda add-permission --function-name my-function --action lambda:InvokeFunction --statement-id sns --principal sns.amazonaws.com`, which produces the corresponding IAM policy statement.

> **COMMON PITFALL**
>
> A single execution role is not enough on its own. It governs only what the function can *do*. Something must still be allowed to *invoke* the function — that's the resource-based policy. When a scenario says an S3 bucket, SNS topic, or another account should trigger a function, reach for a resource policy; when the function itself must write to DynamoDB or read a queue, reach for the execution role.

7.5 Authoring and configuring Lambda functions

The **function handler** is the entry point Lambda calls to start running your code. It always takes two objects. The **event object** passes event information to the handler: it uses a predefined format for AWS integrations and events (an API Gateway event carries path, query string, and request body; an S3 event carries bucket and object details) and can also be a custom, user-defined object for testing. The **context object** passes runtime information and lets your code interact

with the Lambda runtime environment; its contents vary by language runtime, but at minimum it includes `awsRequestId` (identifies a specific invocation, useful for error reporting and support), `logStreamName` (the CloudWatch log stream your statements go to), and the `getRemainingTimeInMillis()` method (milliseconds left before the function times out).

> **WORKED EXAMPLE – A PYTHON HANDLER**
>
> A Python function defines `def lambda_handler(event, context):`, reads fields from the event (for example a first name and last name), builds a greeting message, and returns a response object containing that message. Given an event carrying a first and last name, the response is a message like "Hello John Smith!". Each language has its own rules for how a handler is defined and referenced within the deployment package.

TABLE 7.3 Performance-related configurations

CONFIGURATION	WHAT IT DOES
Memory	Amount of memory and proportional CPU allocated (128 MB–10,240 MB); Lambda scales CPU linearly with memory
Timeout	Maximum time a function may run before it is ended (1 second–15 minutes)
Concurrency	Invocations that can run at once; 1,000 per Region per account by default (soft), with an optional per-function limit
Provisioned concurrency	Number of environments to keep warm; avoids cold starts (startup latency), priced separately
Monitoring and operations	Enable X-Ray active tracing and CloudWatch Lambda Insights for runtime performance metrics and logs

TABLE 7.4 Resource- and code-related configurations

CONFIGURATION	WHAT IT DOES
Triggers	The event sources that invoke the function
Permissions	Who may invoke the function and what the function may do to other resources
Destinations	An SNS topic, SQS queue, another function, or EventBridge bus that receives records on success or on failure
Asynchronous invocation	Retry attempts (0–2), how long to keep an event waiting (up to 6 hours), and a dead-letter queue
Runtime, env variables, tags, code signing	The runtime or language; key-value config read by the code without changing it; labels for search and cost tracking; verification that code is signed and unaltered
VPC, proxies, file systems, state machines	Access resources in a custom VPC; RDS database proxies for connection pooling; mounted EFS file systems; Step Functions state machines that invoke the function

Memory and timeout also drive **billing** — you are charged per request and by **duration**, with the per-millisecond price rising as configured memory rises — which is exactly what tuning balances.

BEST PRACTICES FOR AUTHORING FUNCTIONS

- **Treat functions as stateless** — save no state in the function's own context, but do **reuse the environment**: initialize expensive dependencies and connections once and reference them locally on later invocations.
- **Include only what you need** — minimize package size and dependencies to shorten startup; add just the SDK modules you use (for example, DynamoDB and S3), not the entire SDK.
- **Separate core business logic** from the handler so code is portable and unit-testable; write modular, single-purpose functions.
- **Log to CloudWatch and return results** — include logging statements, and give Lambda information about the outcome of each run.
- **Use environment variables** for operational parameters (for example, an S3 bucket name) instead of hardcoding, including sensitive values.
- **Avoid recursion** — a function calling itself can spawn runaway invocations and lose control of concurrency.
- **Don't call one function from another** — use destinations or orchestrate with Step Functions.

Tuning balances speed against cost. Higher memory configurations cost more per millisecond but can **decrease duration and concurrency needs**, so a CPU-intensive function may run both faster and cheaper at a higher memory level — more memory is not automatically more expensive. Use the open-source AWS Lambda Power Tuning utility (and CloudWatch Logs to view memory usage) to find the balance for your workload rather than trimming memory on instinct, and load-test to choose the right timeout, especially when the function makes network calls to downstream resources that must keep up with Lambda's scaling.

7.6 Deploying Lambda functions

To deploy with the Lambda API, CLI, or SDKs you create a **deployment package** (also required for compiled languages or to add dependencies). There are two package types: a **.zip archive** — choose a runtime, compress your code and dependencies, and upload — or a **container image** — choose a runtime and Linux distribution, package code and dependencies as an image, and push it to Amazon Elastic Container Registry (Amazon ECR). The .zip package limits are **50 MB** compressed (direct upload), **250 MB** uncompressed (including layers), **3 MB** in the console editor, and **512 KB** per individual file.

For a **.zip archive** you choose a runtime and upload compressed code plus dependencies in one of three ways: the **console editor** (small code, no custom libraries), an **upload from your IDE** (custom libraries), or **from an S3 bucket** (bucket, object key, optional version). For a **container image** you choose a runtime and Linux distribution, package code and dependencies as an image, and `docker push` it to Amazon ECR; on create or update Lambda pulls, optimizes, and deploys the image, and the function reaches **Active** status.

Versioning publishes immutable copies of a function's code and configuration so you can work with variations like development, beta, and production. When you create a function only the `$LATEST` version exists; publishing makes a **snapshot copy** of `$LATEST` as a new version with its own ARN and a new sequential version number. **Aliases** are **mutable pointers** to a specific version, each with its own ARN. Because you can repoint an alias without changing code, aliases let you promote or roll back cleanly. The payoff shows with event sources: if S3 invokes your function by ARN, you'd have to update the bucket notification for every new version — but if you point the notification at a `PROD` alias, you only repoint that alias as you promote new versions, and never touch the S3 configuration.

WORKED EXAMPLE – DEPLOYING AND UPDATING FROM THE CLI

Create a function with `aws lambda create-function`, passing `--function-name`, `--runtime`, `--zip-file`, `--handler`, and `--role`; the output reports the function ARN, version `$LATEST`, memory, timeout, and a `CodeSha256`. Push new code later with `aws lambda update-function-code --function-name my-function --zip-file fileb://my-function.zip`, and the output shows an updated `RevisionId` and `CodeSha256`.

Custom runtimes let you use other languages: Lambda provides a runtime interface for getting events and sending responses, the **bootstrap** file is the runtime's entry point, and you deploy a custom runtime next to your function code or in a layer. **Lambda layers** centrally manage code and data shared across functions — a layer is a .zip archive of libraries, a custom runtime, or dependencies. Layers **reduce deployment-package size and speed up deployments**, and keeping a package small (under 3 MB for Node.js, Python, and Ruby) lets you keep developing in the console editor. A function can use **up to five layers**, and the total extracted size of the function plus all layers cannot exceed **250 MB**.

COMMON PITFALL

This is the module's sample-exam scenario: a developer left dependencies out of the deployment package to stay under the size limit, and the function now fails remotely for missing dependencies. The fix is not the console editor (limited and for code without custom libraries) or environment variables (they don't hold code) — it's to **create a layer containing the dependencies and attach it** to the function.

7.7 Monitoring and debugging tools for application developers

Modern applications strain traditional monitoring: resources are **short-lived**, there are more devices, services, and data, **release velocity** is faster, and **user experience** is now a first-class performance concern (users won't tolerate slow responses). Alerting on errors in one layer of a stack is no longer enough — you need visibility across many independent, distributed components. Two services carry that load for Lambda developers: **Amazon CloudWatch** for logs, metrics, and events, and **AWS X-Ray** for distributed tracing.

Amazon CloudWatch collects monitoring and operational data as **logs, metrics, and events**. It gives a unified view of AWS resources, applications, and services; lets you review logs as a time-ordered flow of events and query them interactively with **CloudWatch Logs Insights**; provides default operational metrics from AWS services; and can **alarm** on metric thresholds to initiate automated actions.

Lambda logs every request and result to a CloudWatch **log group** named `/aws/lambda/<function name>` — a one-to-one relationship between function and log group, created the first time the function runs (or when provisioned concurrency is set). A log group contains **log streams** identified by date, function version, and a unique ID. Each stream holds a sequence of events per invocation: a unique **RequestID**, and **START**, **END**, and **REPORT** events. The RequestID is the primary key for finding events for a specific invocation; the REPORT event summarizes how long the function ran, the billed duration, and configured versus used memory. An **Init duration** in the REPORT tells you the invocation was a **cold start**.

AWS X-Ray traces requests as they travel through your application and draws a **map of its underlying components**. It offers **one-click enablement** for Lambda functions and integrates with other AWS services; an **X-Ray SDK** captures metadata for API calls made to AWS services with the AWS SDK; and it visually detects **latency distribution**, isolates outliers and trends, eases debugging, and lets you **filter and group** requests by error type.

TABLE 7.5 Parts of an X-Ray trace

ELEMENT	WHAT IT CAPTURES
Segments	Data about the work done by the compute resources running your application — resource name and request details
Subsegments	A finer breakdown of a segment — granular timing, downstream calls, inferred segments for downstream services
Annotations	Key-value pairs that **are indexed** and can be used with filter expressions to group traces for analysis
Metadata	Key-value pairs of any type that are **not indexed** — stored in the trace but not used for searching

A trace of an API Gateway request to a Lambda function that queries DynamoDB shows, in its segments and subsegments, exactly where time is spent — the fastest route to a trouble spot. The module demonstrates X-Ray with Lambda, and the lab replaces the café's mock endpoints with `get_all_products` and `create_report` functions.

> **COMMON PITFALL**
>
> Annotations and metadata look similar but differ. **Annotations are indexed** and searchable with filter expressions — use them for fields you'll group and query traces by. **Metadata is not indexed** — use it for detail you want recorded but won't search on. Picking metadata for something you later need to filter by is the common mistake.

Module summary

- Serverless removes infrastructure management (no instances, OS, or capacity planning) along a spectrum — on-premises → virtual servers → containers → serverless — leaving you to build, deploy, and monitor, and to pay only when code runs.
- **AWS Lambda** runs code in response to events, scales automatically, has built-in fault tolerance, and monitors with CloudWatch. Functions and event sources are its two core components; functions run in on-demand ephemeral environments, and concurrency is how many run at once.
- Event sources push (a service invokes directly) or pull (Lambda polls a stream or queue via an event source mapping). Three invocation models: synchronous (200, no built-in retry), asynchronous (202, two retries, dead-letter queue), and polling (streams retry to a 24-hour expiry and can block a shard; queues retry to a limit) — predetermined for AWS-service triggers.
- Two permissions: a resource-based policy grants event sources the lambda:InvokeFunction action (who can invoke); an execution role (IAM policy + trust policy for lambda.amazonaws.com) grants what the function can do and lets polling sources read the stream or queue.
- The handler receives an event object and a context object (awsRequestId, logStreamName, getRemainingTimeInMillis()). Configure memory (128 MB–10,240 MB, CPU-linked), timeout (1 s–15 min), concurrency (1,000/Region soft), and provisioned concurrency for cold starts; keep functions stateless and packages small, and tune memory against duration cost.
- Deploy a .zip (50 MB zipped, 250 MB unzipped, 3 MB in console) or a container image on Amazon ECR; versions are immutable ($LATEST mutable), aliases are mutable pointers, and layers (≤5, 250 MB) share code. Monitor

> with CloudWatch logs (/aws/lambda/<function>, START/END/REPORT, Init = cold start) and X-Ray traces (segments, subsegments, indexed annotations, non-indexed metadata).

Review questions

1. **What are the two core components of Lambda, and how does the push model differ from the pull model?**

   Answer: Functions (your code) and event sources (what publishes events). In the push model an event source invokes the function directly; in the pull/polling model the source puts data in a stream or queue and Lambda, via an event source mapping, polls it and invokes the function with batches of records.

2. **Which two permissions does Lambda require, and what does each control?**

   Answer: A resource-based (function) policy grants event sources permission to invoke the function via lambda:InvokeFunction (inbound). An execution role — an IAM policy plus a trust policy that lets lambda.amazonaws.com assume it — grants what the function can do to other services (outbound); polling sources also need it to read the stream or queue.

3. **Give the value ranges for memory, timeout, and the default per-Region concurrency, and say what provisioned concurrency is for.**

   Answer: Memory 128 MB–10,240 MB (CPU scales linearly with it), timeout 1 second–15 minutes, and a default soft limit of 1,000 concurrent invocations per Region per account. Provisioned concurrency keeps environments warm to avoid cold-start latency and is priced separately.

4. **How do versions and aliases work together, and why does pointing an S3 event source at an alias help?**

   Answer: Versions are immutable snapshots of code and configuration ($LATEST is the mutable working copy); an alias is a mutable pointer to a version. If S3 invokes the function by version ARN you must update the bucket notification each release, but if it points at an alias (e.g., PROD) you just repoint the alias as you promote versions and never touch the S3 configuration.

5. **A function fails remotely because dependencies were omitted to stay under the package size limit. What's the fix, and why not the alternatives?**

   Answer: Create a Lambda layer containing the dependencies and attach it (up to 5 layers, 250 MB extracted). The console editor is limited and meant for code without custom libraries, and environment variables hold configuration values, not code — so neither resolves missing dependencies.

6. **Where does Lambda write logs, what marks a cold start, and how do X-Ray annotations differ from metadata?**

   Answer: Lambda logs to a CloudWatch log group named /aws/lambda/<function name>, with START/END/REPORT events per invocation; an Init duration in the REPORT event marks a cold start. In an X-Ray trace, annotations are indexed key-value pairs you can filter and group traces by, while metadata is not indexed and is stored but not searchable.