MODULE 14 · STUDY NOTES

Building Serverless Architectures and Microservices

When to go serverless, how microservices decompose a monolith, and how Lambda, containers, Step Functions, and API Gateway compose into scalable, event-driven architectures

Earlier modules built applications on servers you provision and manage — EC2 instances, Auto Scaling groups, load balancers, and databases inside a VPC. This module flips that model. **Serverless** services hand the servers, their sizing, scaling, patching, and availability to AWS, so you deploy only your code and pay for what you actually use. The module first frames *when* serverless is the right call and catalogs the serverless services across compute, integration, and data. It then decomposes a monolith into **microservices** — small, autonomous, specialized services — and builds them with **AWS Lambda** for short, event-driven work and **AWS container services** (Amazon ECS, Amazon EKS, AWS Fargate) for longer or heavier work. Because many small services need coordinating, **AWS Step Functions** orchestrates them as state machines, and **Amazon API Gateway** publishes the APIs that are the front door to it all. The module closes by mapping these services onto **AWS Well-Architected Framework** best practices, using the café's move of a resource-hungry cron job into a scheduled Lambda function as the running example.

LEARNING OBJECTIVES

- Define serverless architectures
- Identify the characteristics of microservices
- Architect a serverless solution with AWS Lambda
- Define how containers are used in AWS
- Describe the types of workflows that AWS Step Functions supports
- Describe a common architecture for Amazon API Gateway
- Use the AWS Well-Architected Framework principles when building serverless architectures

14.1 Thinking serverless

To appreciate what serverless removes, start with the architecture it replaces. A classic **three-tier web application in a VPC** routes a web domain to an **Application Load Balancer**, which fronts a **web tier** of EC2 instances spread across two Availability Zones in an Auto Scaling group. That tier calls a second load balancer in front of an **app tier** (also multi-AZ, also auto-scaled) that holds the business logic, which in turn reads and writes a **data tier**: an Amazon RDS **Multi-AZ** primary database that replicates synchronously to a standby in a second AZ. The customer patches and monitors the web and app EC2 instances; AWS manages the RDS instances. The tiers are decoupled and independently scalable with the network as the boundary — but standing them up means building and maintaining many **undifferentiated** components (load balancers, message queues, authentication) that look the same for every application.

AWS began building **serverless** services after noticing in 2013 that customers ran EC2 instances for very short periods. A service is serverless when you can deploy an application **without thinking about servers or their sizing** — AWS

manages the runtime environment and right-sizes the service, so resources can be consumed for as little as double-digit milliseconds. A Lambda function, for example, deploys with one click and is ready within seconds. Pricing is **usage-based** (compute time plus storage), which eliminates paying for idle capacity — a decisive advantage for startups keeping overhead low.

WHY TEAMS GO SERVERLESS-FIRST

- **No server management** — no instances, operating systems, or servers to configure, patch, or scale
- **Pay-for-value** — billed by usage and storage, down to milliseconds; nothing to pay when idle
- **Continuous scaling** — services scale on a metric automatically (e.g., DynamoDB adjusts a table's read/write capacity to traffic)
- **Built-in high availability** — serverless data stores span three AZs; Fargate containers run isolated from other containers' failures
- **Fits event-driven and microservice designs** — services can be invoked by events, emit events, and are built to work with them

That last benefit is central to the module. An **event-driven architecture (EDA)** is built from small, decoupled services that **publish, consume, or route events** — events being the messages passed between services. Serverless services are invoked by events and emit events natively: a single Lambda function can handle **SQS** queue messages, **Kinesis Data Streams** or **DynamoDB Streams** streaming events, or **S3** object events, with no polling code of your own.

TABLE 14.1 AWS serverless services across the application stack

LAYER / CATEGORY	SERVICES	WHAT THEY PROVIDE
Compute	AWS Lambda, Lambda@Edge, AWS Fargate	Run functions or containers with no servers to manage
API publishing	Amazon API Gateway, AWS AppSync	REST/HTTP/WebSocket APIs; AppSync serves GraphQL, collecting data from many stores in one request
Messaging	Amazon SNS, Amazon SQS	SNS = publish/subscribe (app-to-app, app-to-person); SQS = queues that decouple components
Orchestration & events	AWS Step Functions, Amazon EventBridge	Step Functions sequences services; EventBridge is an event bus that routes events and checks schemas
Data stores	S3, EFS, DynamoDB, Aurora / Redshift / Neptune / OpenSearch Serverless	Objects, shared files, key-value/document, relational, graph, and search — each scaling on usage
Auth · hosting · delivery	Amazon Cognito, AWS Amplify, Amazon CloudFront	Cognito user pools and external IdPs; Amplify builds/hosts apps; CloudFront delivers static content from S3

THREE-TIER IN A VPC	SERVERLESS THREE-TIER
you manage the servers	*AWS manages the servers*
▪ ALB → EC2 web tier (multi-AZ, Auto Scaling) ▪ ALB → EC2 app tier (multi-AZ, Auto Scaling) ▪ Amazon RDS Multi-AZ primary with synchronous standby ▪ You patch and monitor the EC2 tiers	▪ CloudFront serves the static front end from S3 ▪ Cognito issues a JWT; API Gateway validates it ▪ Lambda runs the business logic ▪ DynamoDB is the data tier

COMMON PITFALL

Serverless is not an automatic default. It shines for spiky, event-driven, low-idle workloads, but the right choice still works **backward from the business need** — a steadily busy, long-running, or memory-heavy workload can cost less on containers or instances. Don't default to serverless any more than you'd default to servers, and expect the decision to evolve as needs change.

14.2 Architecting serverless microservices

A **microservices architecture** builds an application as independent components, each running one application process as a service and communicating through **well-defined, lightweight APIs**. Because each service is built for a single business capability and runs independently, it can be updated, deployed, and scaled on its own.

AUTONOMOUS	SPECIALIZED
runs and scales on its own	*does one thing well*
▪ Developed and deployed without affecting other services ▪ Scales independently to meet demand ▪ Shares no code or implementation ▪ Communicates only over well-defined APIs	▪ Performs one business function (e.g., login) ▪ Owned by a small team that picks its tools ▪ Is stateless for fast startup and scaling ▪ Owns its own data store (supports ACID)

Consider a **forum** application handling **users, topics, and messages**. As a **monolith**, those processes run as one tightly coupled service: a demand spike in any one process forces scaling the entire application, adding features gets harder as the code base grows, and one process failure endangers the whole app's availability. Re-architected as microservices, each process becomes an independent component talking over lightweight API operations; each performs a single function that can serve multiple applications, and each is updated, deployed, and scaled on its own. If a service grows too complex over time, it can be split again into smaller services.

SIX BENEFITS OF MICROSERVICES

- **Team agility** — small, independent teams own a service and move in quick cycles
- **Reusable code** — a service becomes a building block for new features, no rewriting from scratch
- **Flexible scaling** — scale each service to its own demand and measure its cost precisely
- **Technological freedom** — each team picks the best tool for its job (no one-size-fits-all)
- **Resilience** — total failure degrades functionality instead of crashing the whole app
- **Simplified deployment** — CI/CD pipelines make trying ideas and rolling back cheap

TABLE 14.2 Common serverless microservice compute patterns on AWS

PATTERN	COMPUTE	WHEN TO USE
RESTful API	API Gateway + AWS Lambda	Stateless request/response; work that finishes within Lambda's 15-min limit
Containers	AWS Fargate behind an ALB (or ECS/EKS)	Work over 15 minutes, or a runtime Lambda doesn't provide
Streaming	AWS Lambda + Amazon Kinesis	Continuous stream data; Lambda scales with Kinesis

Each pattern pairs the compute with a **serverless data store**. In a web three-tier design, microservices operate in the **app and data tiers**: combining **API Gateway** (the API), **Lambda** (compute), and **DynamoDB** (data store) yields a serverless microservice where each API request fulfills one business capability. Because Lambda functions cap at 15 minutes, they are ideal for this compute role — but note that microservices form only *part* of a three-tier solution and are not a standalone full architecture.

> **COMMON PITFALL**
>
> Breaking a monolith into microservices is not free upside. The autonomy is real, but so is the coordination cost: many interdependent services must be chained, retried on failure, kept stateless yet made to track workflow state, and monitored — which is exactly why the module then introduces **Step Functions**. Microservices trade one big problem for many small ones.

14.3 Building serverless architectures with AWS Lambda

TABLE 14.3 Operational tasks: server in a VPC vs. serverless

OPERATIONAL TASK	SERVER IN A VPC	SERVERLESS
Configure an instance	Yes	No
Update the operating system	Yes	No
Install the application platform	Yes	No
Build and deploy apps	Yes	Yes
Configure auto scaling and load balancing	Yes	No
Continuously secure and monitor instances	Yes	No
Monitor and maintain apps	Yes	Yes

The chart makes serverless's value concrete: the only operational tasks left to you are **building and deploying** your app and **monitoring and maintaining** it in production. Everything about the instance, OS, platform, scaling, and instance security disappears, letting developers focus on the code that makes the business unique.

AWS LAMBDA ESSENTIALS

- Runs code **without provisioning or managing servers**, on high-availability infrastructure AWS administers
- You configure **runtime language**, **memory**, and **timeout**; memory (128 MB–10,240 MB) also sets vCPU and network bandwidth
- **15-minute maximum** duration — an AWS hard limit that can't be raised
- Deploy as a **.zip archive** or a **container image**; Lambda runs many instances in parallel within concurrency limits
- **Pay only for compute time consumed** — no charge when the code isn't running

A Lambda function runs either inside a VPC **owned by the Lambda service** or in an Amazon **CloudFront regional edge cache**. When invoked in-Region, the service spins up an isolated **Firecracker** microVM (an AWS technology that boots in under a second) on an EC2 instance in the Lambda VPC; after the run, the microVM is reused for the next request. **Lambda@Edge** extends this to CloudFront: you author **Node.js or Python** functions in **US East (N. Virginia)** and run them at edge locations closer to the viewer to cut latency — for example, rewriting a request so CloudFront returns a jacket image in the color a cookie recorded. **CloudFront functions** go even lighter but have **no network access** and far smaller execution-time and package-size limits.

By default a function is **not** connected to the VPCs in your account. To reach private resources you attach it to your VPC, and Lambda creates managed **Hyperplane elastic network interfaces (ENIs)** that provide **VPC-to-VPC NAT (V2N)** from the Lambda VPC into your VPC — one direction only. A VPC-attached function has **no internet access** unless you route outbound traffic to a **NAT gateway** in a public subnet. Watch for scaling bottlenecks: because functions scale fast, they can **saturate database connections**, so for MySQL and Aurora Amazon RDS databases put an **RDS Proxy** between Lambda and the database to pool connections.

TABLE 14.4 Lambda invocation models

MODEL	BEHAVIOR	TRIGGERS / EXAMPLES
Synchronous	Caller waits for the response	API Gateway or a function URL; web/mobile microservices, ML inference (e.g., add to cart)
Asynchronous	Lambda queues the event, returns success, processes it later	S3, SNS, scheduled EventBridge rules; file/image processing, package-status updates
Event source mapping	Lambda polls a stream/queue and invokes with a batched payload (max 6 MB)	DynamoDB, Kinesis, SQS, DocumentDB streams and queues

For **synchronous** calls the two front doors differ in one important way. With **API Gateway**, the client hits the gateway, which invokes the function and relays the response — and if the function changes behind it, the client needs **no update**. A **function URL** is a dedicated HTTP(S) endpoint Lambda generates (fixed once created, secured by **resource-based policies**, and CORS-capable) that the client calls directly — simple to set up, but if the URL ever changes the **browser app must be updated**.

For **asynchronous** calls you hand the event to Lambda; it places the event on a queue, returns success immediately, and a separate process delivers it to the function. You can configure error handling and send invocation records downstream to **SQS** or **EventBridge** to chain components. **Event source mappings** cover services that don't invoke Lambda directly: the mapping polls a stream or queue, **batches** records into one payload (up to **6 MB**), and invokes the function — for example, a DynamoDB orders row flipping to *delivery in progress* triggering a customer notification and financial processing.

WORKED EXAMPLE – THE LAMBDA HANDLER, EVENT, AND CONTEXT

The **handler** is your function's entry point; Lambda runs it on each invocation and passes two arguments. The **event** object is a JSON document carrying the input data and information about the invoking service; the **context** object exposes methods and properties about the invocation, function, and runtime. A handler like `lambda_handler(event, context)` reads values from the event (for instance `event['length']`), and it's a best practice to keep the handler small and push real work into **separate business-logic methods** so the handler loads faster. The console default handler name, `lambda_function.lambda_handler`, encodes the file (`lambda_function.py`) and the function (`lambda_handler`). Amazon Q Developer can suggest function code in the console or your IDE.

LAMBDA LAYERS

- A **layer** is a .zip archive of supplementary code or data — library dependencies, a custom runtime, or config files
- **Shrink deployment packages** by moving dependencies out of the function package
- **Separate logic from dependencies** — update each independently
- **Share** one layer across any number of functions in your account
- Smaller packages can unlock the **console code editor** for quick edits

COMMON PITFALL

Attaching a Lambda function to your VPC does not also give it internet access — it does the opposite. A VPC-attached function loses default internet access and reaches out only if you provide a **NAT gateway** path. And a function's **IAM role** isn't optional plumbing: the role is what grants the function permission to call other AWS services such as S3 and DynamoDB.

14.4 Building microservice applications with AWS container services

WHEN TO CHOOSE CONTAINERS OVER LAMBDA

- **Runs longer than 15 minutes** — beyond Lambda's hard timeout
- **Needs more than 10 GB of memory** — beyond Lambda's memory ceiling
- **Legacy migration** — move on-premises or EC2 apps without refactoring or a new runtime
- **Cost** — continuous workloads run at a fixed container cost, while Lambda cost climbs with invocations, duration, and memory

The cost crossover is real. Using the module's illustrative figures: a Lambda function at 3 million requests/month averaging 120 ms on 1,536 MB (Arm) costs about **$7.80/month**, but raise it to 5 GB and 5,000 ms and it jumps to about **$1,000.60/month** — whereas five **Fargate** containers running continuously at 5 GB cost about **$180.65**. Above some transactions-per-second point, always-on containers beat per-invocation Lambda. (These prices are illustrative.)

VIRTUAL MACHINES	CONTAINERS
heavier, OS included ▪ Package bundles app code, dependencies, and a full **guest OS** ▪ Runs on a **hypervisor** over the host hardware ▪ Larger images, so fewer fit on one host	*lightweight, share the host OS* ▪ App code + dependencies; **no guest OS** ▪ Runs on a lightweight **container engine** ▪ Small, fast to start; more fit per host

You build a container by writing a **Dockerfile** — text build instructions for the image — then running a build command, testing locally, and pushing the image to a **container repository**; typically many containers deploy from one image. Because manual deployment doesn't scale to thousands of containers, teams use a **container orchestration service** that pulls the image, deploys the required count across a compute platform, and tracks the health of hosts and containers. Common engines are **Docker Engine, containerd, and Red Hat OpenShift**, and the **Open Container Initiative (OCI)** standardizes image, runtime, and distribution so containers port across platforms unchanged.

TABLE 14.5 AWS container service building blocks

ROLE	SERVICE(S)	NOTES
Registry	Amazon ECR	Managed image registry; private (IAM-scoped) and public repositories
Orchestration	Amazon ECS, Amazon EKS	ECS is AWS-opinionated; EKS runs Kubernetes; each has an Anywhere on-premises option
Compute	Amazon EC2, AWS Fargate, AWS Lambda	EC2 nodes run in your VPC; Fargate nodes are AWS-managed; Lambda can run containers up to 10 GB

WHY AWS FARGATE

- **No cluster or server management** — no provisioning, patching, or cluster-packing to optimize
- **Per-second billing** — pay only for the CPU and memory you use
- **Automatic scaling** of tasks on CPU, memory, or other metrics (via Amazon ECS capacity providers)
- **Approachable** — scaling and security are built in, so teams new to containers can start fast

On **Amazon ECS** you first create a **cluster** of Fargate and/or EC2 compute nodes, then define an **ECS task definition** — a JSON blueprint naming each container image, its CPU/memory, and the launch type. A **task** is the smallest unit of compute (one or more co-placed containers); run a task definition as a one-off **task** or as a **service** that maintains a desired task count and replaces any that fail. An **Application Load Balancer** in the service distributes requests at the application layer with **path-based routing** — for example, forwarding `/api/login` to the login container.

On **Amazon EKS** you create a cluster of worker **nodes** (Fargate and/or EC2) and work through Kubernetes objects rather than containers directly. A **Pod** is the smallest deployable unit (one container, or several interdependent ones); a **PodSpec** names the image for a single Pod, while a **Deployment** owns a **ReplicaSet** that keeps a set number of Pods running. A **Kubernetes Service** routes network requests to Pods that share a label.

TABLE 14.6 Choosing between Amazon ECS and Amazon EKS

TOPIC	AMAZON ECS	AMAZON EKS
Complexity	Simplifies cluster creation and maintenance	More control via a more complex interface
Scaling	Automated scaling based on demand	Manual configuration of autoscaling groups
Toolset	Amazon ECS toolset	Kubernetes toolset
Team experience	New to container clusters	Familiar with Kubernetes clusters

COMMON PITFALL

AWS Fargate is not a competitor to ECS and EKS. Fargate isn't an orchestrator — it's a **serverless compute option for the nodes** inside an ECS or EKS cluster (the alternative to EC2 nodes). You still pick ECS or EKS to orchestrate; Fargate just removes the servers under them, and a cluster can even mix Fargate and EC2 nodes.

14.5 Orchestrating microservices with AWS Step Functions

WHY MICROSERVICES NEED ORCHESTRATION

- **Dependencies** — chaining services sequentially or in parallel, gated on each other's success
- **Error scenarios** — retrying after timeouts or failures; a function may have run fully, partially, or not at all
- **Idempotency** — retried events mean code must process the same request repeatedly without side effects
- **Coordination** — auto-scaling the coordination layer, maintaining workflow **state**, and passing data between steps
- **Visibility** — monitoring and troubleshooting across many services

AWS Step Functions is a serverless **orchestration** service that coordinates distributed applications and microservices as visual workflows. You build a **state machine** (workflow) as a series of event-driven **states** (steps); Step Functions triggers and tracks each step, **retries on error**, logs every step's state for debugging, and can **restart a workflow from its point of failure**. It manages the workflow's state and checkpoints, transfers and filters data between states, and can be invoked from **API Gateway, EventBridge, Lambda, or another state machine** — and nested state machines cut complexity and let you reuse common processes.

TABLE 14.7 Step Functions workflow types: Standard vs. Express

ATTRIBUTE	STANDARD	EXPRESS
Duration	Long-running, up to **1 year**	Short, up to **5 minutes**
Run model	Exactly-once	Sync: at-least-once · Async: at-most-once
Run history	Full history in the console	Results sent to CloudWatch Logs
Progress	Persisted on every state transition	Not persisted on every transition
Pricing	Per state transition	Per request and duration
Best for	Auditable, long workflows (order fulfillment)	High-event-rate work (streaming, IoT ingestion)

Common uses are **microservice orchestration, data processing, machine learning, and security automation**. For long-running work, Standard Workflows integrate with **Fargate**; for high-volume work needing an immediate answer,

Synchronous Express workflows can be launched directly from API Gateway with the connection held open until they finish. **Asynchronous Express** workflows only confirm that they started — you poll **CloudWatch Logs** for the result. Data-processing workflows scale to millions of concurrent executions and use the **Parallel** and **Map** states for parallelism; ML workflows orchestrate SageMaker steps; security automation leans on automatic retries with **exponential backoff**. A common tactic is to debug in a Standard Workflow (for its visual history) and then copy it to Express for production.

TABLE 14.8 State machine state (step) types

CATEGORY	STATE	PURPOSE
Work	Task	A unit of work an AWS service (e.g., Lambda) or an HTTP endpoint performs
Work	Activity	Work done by a worker hosted anywhere (EC2, Lambda, mobile, any HTTP app)
Work	Pass	Passes or filters input to output without doing work
Work	Wait	Delays the workflow for a relative or an absolute time
Transition	Choice	Branches to the next state using comparison-based choice rules
Transition	Parallel	Runs separate branches of nested state machines at once
Transition	Map	Runs workflow steps for each item in a dataset, in parallel
Stop	Succeed / Fail	Ends the run, marking it successful or failed (or via the End parameter)

A state machine is written in **Amazon States Language**, a JSON structure whose **StartAt** field names the first state and whose **States** map holds each named state with a **Type** and a **Next** pointer (or an **End**, or a transition to a Succeed/Fail state). Work states can also use a **wait-for-callback** option with a **task token** — the workflow pauses until an external worker calls back.

WORKED EXAMPLE – THE STOCK-TRADE STANDARD WORKFLOW

The module's stock-trade state machine shows the pieces working end to end. A Lambda **Task** returns the trade details → a **Choice** checks whether a trade is recommended (if not, go to Succeed) → another **Choice** tests whether the trade amount exceeds a limit such as $100 → if it does, a **Task** invokes **Amazon SNS** to email an approver with a task token → the approver's **callback** carries the token back and drives a Trade-approved **Choice** → if approved, a Lambda **Task** executes the trade → a final **Choice** tests success and routes to a **Succeed** or a **Fail** state. Every dependency, branch, retry, and human-approval pause lives in the workflow, not scattered across the functions.

COMMON PITFALL

Don't reach for Express Workflows just because you want speed. Express is fast and cheap for high event rates, but it is **at-least-once/at-most-once**, doesn't persist state on every transition, and doesn't keep full console history. When you need **exactly-once** execution, an audit trail, or runs beyond 5 minutes, use **Standard** — and even Express workflows are often *developed* as Standard for the visual debugging, then copied over.

14.6 Extending serverless architectures with Amazon API Gateway

WHY FRONT MICROSERVICES WITH AN API

- **Standardize communication** — bridge apps written in different languages and data formats, hiding implementation detail
- **Protect the microservice** — require authorization, validate request formats, throttle requests, restrict resource access
- **Monetize and measure** — track per-client usage for billing and supply usage statistics per client

Amazon API Gateway is a fully managed service to create, publish, maintain, monitor, and secure **REST, HTTP, and WebSocket** APIs at any scale, acting as the entry point to backends on **EC2, Lambda, ECS, VPC or on-premises apps, or public endpoints** — and it integrates directly with some AWS services (Kinesis, Step Functions, DynamoDB). It handles hundreds of thousands of concurrent requests and adds traffic management, authorization (integrated with **Cognito**, plus **Lambda authorizer** plug-ins), versioning and **stages**, **usage plans** with API keys (e.g., 1,000 requests/day and a maximum of 5 RPS) for throttling and billing, and **response caching** to cut backend calls and latency.

TABLE 14.9 Choosing an API Gateway API type

API TYPE	BEST FOR	KEY TRAITS
REST	Full API management and control	Usage plans, payload validation, private endpoints, resource policies; CORS; stateless
HTTP	Serverless microservices (Lambda/HTTP backends)	Lower latency and lower cost than REST; CORS; stateless
WebSocket	Real-time applications (chat, live data)	Persistent, bidirectional session between client and backend; stateful

RESTful APIs are collections of **routes and methods** — the HTTP verbs **GET, POST, PUT, PATCH, DELETE** mapping to CRUD (for example, `POST /class-roster` adds a student, `GET /class-roster` lists them). API Gateway reaches backends three ways: an **HTTP proxy integration** passes whole requests to a public HTTP endpoint; a **first-class integration** calls an AWS service API directly (send to an SQS queue, start a Step Functions state machine); and a **VPC link** reaches load balancers and services inside a VPC — a **Network Load Balancer** for REST APIs, or an NLB/ALB and services such as EC2 and ECS for HTTP APIs.

WORKED EXAMPLE – DECOMPOSING AN ONLINE-SHOPPING MONOLITH

The module's activity splits a shopping monolith into three services, each matched to its requirements. **Shopping cart** (single-digit-second responses, millisecond transactions) → an **HTTP API** to a **Lambda** function backed by a **DynamoDB** cart table. **Payments** (must be ported from on-premises with minimal change) → an **HTTP API** to an **ALB** distributing to **ECS** payment containers that call a public banking API and store to a DynamoDB payment table — a container precisely because the legacy code moves as-is. **Delivery** (can take up to 5 days) → not a microservice at all, but a **Step Functions Standard Workflow**: the state machine drops the request on an **SQS** queue and suspends; a delivery API polls the queue and **calls back**; a Choice state then updates DynamoDB and emails the customer via **SNS**, marking success — or, if undelivered, cancels, refunds via another SQS queue, and marks failure.

COMMON PITFALL

REST APIs are not the modern default. For serverless microservices proxying to Lambda, **HTTP APIs** are usually the better pick — lower latency and cost — precisely because they *lack* the heavier REST management features. Choose **REST** only when you need those features (usage plans, request validation, private endpoints, resource policies), and **WebSocket** when the app needs a stateful, real-time session.

14.7 Applying Well-Architected principles to serverless architectures

The **AWS Well-Architected Framework** has **six pillars**, each with best practices and questions to weigh when architecting a cloud solution. AWS also publishes a specialized **Serverless Applications Lens** covering common serverless scenarios. The module highlights a subset of best practices that map directly onto the services it taught.

TABLE 14.10 Well-Architected best practices applied in this module

FOCUS AREA	BEST PRACTICE	HOW IT APPLIES HERE
Failure management	Dead-letter queues; roll back	Send failed Lambda transactions to an SQS DLQ; roll back synchronous work with Step Functions
Identity & access	Control API and application access	Cognito user pools, API Gateway Lambda authorizers, resource policies; least-privilege roles and small functions
Data protection	Encrypt in transit and at rest; app security	Encrypt sensitive data before processing; validate requests against a JSON schema at API Gateway
Selection (performance)	Optimize performance	Load-test steady and burst rates; tune Lambda memory; edge vs. regional endpoints; test Standard vs. Express
Cost-effective resources	Optimize cost	1-ms billing rewards faster functions; right-size memory, which scales CPU/network/IOPS
Optimizing over time	Use direct service integrations	Drop a Lambda that only forwards data — let API Gateway/Step Functions/EventBridge integrate directly

A few of these deserve detail. For **failure management**, asynchronous calls that fail should be captured and retried or data is lost; Lambda can route failed transactions to a per-function **SQS dead-letter queue**, and because Kinesis and DynamoDB Streams retry the **whole batch** (a poison-pill record can block a shard until it expires), you send that record's metadata to a DLQ to unblock processing. For **optimizing over time**, the clickstream example is instructive: client → API Gateway → Lambda → Kinesis Data Firehose → S3 works, but having the client put data **directly** on Firehose removes the API Gateway and Lambda cost entirely.

WORKED EXAMPLE – THE CAFÉ'S VERSION 7 (SERVERLESS REPORTING)

The café's daily order-report emails ran as a **cron job on the web-server instance**, which was resource-intensive and slowed the site. Following the principle that non-business-critical reporting should be decoupled, the team moved it to a **managed, serverless** design: a **DataExtractor** Lambda function reads the café's sales data from the **Amazon RDS** database — enabled by giving the function its own security group and adding it as an inbound rule on the RDS instance's security group — and a **salesAnalysisReport** Lambda runs the report **on a daily schedule created with an EventBridge rule**. The result scales well, costs less, and keeps reporting off the web server.

COMMON PITFALL

Not every integration needs a Lambda function in the middle. If a function only shuttles data between services without applying custom logic, it's likely **unnecessary cost and latency**. API Gateway, Step Functions, EventBridge, and Lambda destinations can integrate with many services **directly** — the well-architected move is often to remove the middleman entirely.

Module summary

- Serverless means deploying without managing or sizing servers; AWS owns the runtime, scaling, patching, and availability. Benefits: no server management, pay-for-value, continuous scaling, and built-in high availability — and it fits event-driven and microservice designs.
- AWS offers serverless services across the stack: compute (Lambda, Lambda@Edge, Fargate), integration (API Gateway, AppSync, SNS, SQS, Step Functions, EventBridge), and data stores (S3, EFS, DynamoDB, Aurora/Redshift/Neptune/OpenSearch Serverless), plus Cognito, Amplify, and CloudFront.
- A microservice is autonomous (independent deploy/scale, no shared code, API-only communication) and specialized (one business function, stateless, own data store); microservices decompose a monolith and operate in the app and data tiers.
- The classic serverless microservice pattern is API Gateway + Lambda + DynamoDB; container and streaming (Kinesis) patterns cover work that doesn't fit Lambda.
- Lambda runs code with no servers: 15-minute max, 128 MB–10,240 MB memory, 6 MB event payload, .zip or container image, paid per compute-ms. It can run at the edge (Lambda@Edge) and attach to your VPC via Hyperplane ENIs (V2N), needing a NAT gateway for internet and RDS Proxy to protect databases.
- Lambda is invoked synchronously (API Gateway or a function URL), asynchronously (S3, SNS, scheduled EventBridge rules), or through event source mappings that poll streams and queues (DynamoDB, Kinesis, SQS, DocumentDB).
- Choose containers over Lambda for work over 15 minutes or 10 GB, legacy migrations, or continuous fixed-cost workloads. AWS container stack: ECR (registry); ECS or EKS (orchestration); EC2, Fargate, or Lambda (compute). Fargate is serverless nodes; ECS auto-scales and is simpler, EKS gives Kubernetes control with manual scaling.
- Step Functions orchestrates microservices as state machines written in Amazon States Language; it retries, checkpoints, restarts, and passes data. Standard workflows are exactly-once and up to 1 year; Express are at-least/at-most-once and up to 5 minutes. States group into work (Task, Activity, Pass, Wait), transition (Choice, Parallel, Map), and stop (Succeed, Fail).
- API Gateway publishes REST (full management, stateless), HTTP (low-latency/low-cost microservices, stateless), and WebSocket (stateful, real-time) APIs; it reaches backends via HTTP proxy, first-class AWS-service integrations, and VPC links, with Cognito auth, usage plans, stages, and caching.
- Decomposing a monolith means matching each piece to the right tool: HTTP API + Lambda + DynamoDB for a fast cart, ECS containers for a ported payment system, and a Step Functions Standard Workflow for multi-day delivery.
- Well-Architected serverless best practices: dead-letter queues and rollbacks for failures; Cognito, authorizers, and least privilege for access; encryption and request validation for data; load-testing and endpoint/memory tuning

> for performance; 1-ms billing and right-sized memory for cost; and direct service integrations to remove unnecessary Lambdas.

Review questions

1. **What qualifies a service as serverless, and what are its four hallmark benefits?**

   Answer: You deploy it without provisioning or sizing servers — AWS manages the runtime, scaling, and availability. The benefits are no server management, pay-for-value pricing, continuous automatic scaling, and built-in high availability / fault tolerance.

2. **List the characteristics that make a microservice autonomous and specialized.**

   Answer: Autonomous: deployed and scaled independently, shares no code, and communicates only over well-defined APIs. Specialized: performs a single business function, is stateless, and owns its own data store (a small team chooses its tools).

3. **What are Lambda's key limits, and what happens when a workload exceeds them?**

   Answer: 15-minute maximum duration, 128 MB–10,240 MB memory, and a 6 MB event-source-mapping payload. Work that runs longer than 15 minutes or needs more than 10 GB of memory moves to containers (Fargate, ECS, or EKS).

4. **Compare the three ways a Lambda function is invoked.**

   Answer: Synchronously (the caller waits — API Gateway or a function URL), asynchronously (Lambda queues the event and returns success — S3, SNS, scheduled EventBridge rules), and via event source mappings (Lambda polls a stream or queue such as DynamoDB, Kinesis, or SQS and invokes with a batched payload).

5. **How do Standard and Express Step Functions workflows differ?**

   Answer: Standard: exactly-once, up to 1 year, full console history, priced per state transition — for auditable, long-running workflows. Express: at-least-once (synchronous) or at-most-once (asynchronous), up to 5 minutes, results in CloudWatch Logs, priced per request and duration — for high-event-rate workloads.

6. **When would you choose REST, HTTP, and WebSocket APIs in API Gateway?**

   Answer: REST when you need full API management (usage plans, payload validation, private endpoints, resource policies); HTTP for serverless microservices needing lower latency and cost; WebSocket for stateful, real-time applications that need a persistent session.

7. **In the shopping-monolith activity, why is delivery built as a Step Functions workflow instead of a Lambda microservice?**

   Answer: Delivery can take up to five days, far beyond Lambda's 15-minute limit, so it is modeled as a long-running Standard Workflow that queues the request on SQS, waits for a delivery callback, then updates DynamoDB and notifies the customer via SNS.