MODULE 5 · STUDY NOTES

Computing (Containers and Serverless)

Serverless compute with AWS Lambda, REST APIs and Amazon API Gateway, containers with Amazon ECS, Fargate, EKS, and ECR, and orchestrating it all with AWS Step Functions

Module 4 covered Amazon EC2 — virtual machines you provision and operate. This module covers the two approaches that let you operate with far less server management: containers and serverless. Serverless computing, delivered by **AWS Lambda**, runs your code with no servers to provision, scale, or patch — you upload a function and AWS scales it and charges only for what runs. Around Lambda sit the pieces that make serverless applications real: **APIs** and the **REST** design that lets programs talk over HTTP, and **Amazon API Gateway**, the managed front door that publishes and secures those APIs. Containers take the opposite trade: you keep more control and gain portability by packaging an application and its dependencies into a unit that runs the same anywhere — supported by **Amazon ECS**, **AWS Fargate**, open-source Kubernetes through **Amazon EKS**, and the **Amazon ECR** image registry. Finally, **AWS Step Functions** ties multiple services into serverless workflows. Throughout, the SysOps question is the same: how much of the stack do you want to manage, and which approach fits the workload?

LEARNING OBJECTIVES

- Explain the purpose and function of AWS Lambda
- Describe the purpose and function of application programming interfaces (APIs), including RESTful APIs
- Explain the benefits and function of Amazon API Gateway
- Explain the purpose and function of containers and the AWS services that support container usage
- Explain the purpose and function of AWS Step Functions

5.1 Serverless computing and AWS Lambda

AWS offers many compute options: **Amazon EC2** provides virtual machines, container services such as **Amazon ECS** run Docker containers, and a third approach — **serverless computing** — requires no servers to provision or manage at all. Serverless computing lets you build and run applications and services **without thinking about servers**: serverless applications do not require you to provision, scale, or manage them. AWS delivers serverless compute through **AWS Lambda**. The advantage is a clean division of labor — developers focus on their application functions, and SysOps administrators no longer manage or operate servers or runtimes, so their tasks reduce to deploying and monitoring the applications.

TRADITIONAL DEPLOYMENT AND OPERATIONS	SERVERLESS DEPLOYMENT AND OPERATIONS
you own every layer	*AWS handles the servers*
▪ Provision an instance	▪ Build and deploy applications
▪ Update the operating system	▪ Monitor and maintain applications
▪ Install the application platform	▪ No instances to provision or OS to update
▪ Build and deploy applications	▪ No platform to install
▪ Configure automatic scaling and load balancing	▪ No scaling or load balancing to configure
▪ Regularly patch, secure, and monitor servers	▪ No servers to patch, secure, or monitor
▪ Monitor and maintain applications	

AWS Lambda lets you run code without provisioning or managing servers for virtually any type of application or backend service. You upload your code, and Lambda takes care of everything required to run and scale it with high availability. You configure the code to be triggered automatically by other AWS services, or invoke it directly from any web or mobile application — and you pay only for the compute time you consume, with no charge when the code is not running. There are no new languages, tools, or frameworks to learn: Lambda supports **Java, Node.js, C#, Python, and Ruby**, and you can use any third-party library.

AWS LAMBDA KEY PRODUCT FEATURES

- **Manages infrastructure and administration** — you upload code and Lambda runs it with high availability; you never update the underlying OS or resize and add servers as usage grows
- **Built-in fault tolerance** — maintains compute capacity across multiple Availability Zones in each Region
- **Automatic scaling** to the rate of incoming requests, with no configuration and no limit on the number of requests; because code is stateless, Lambda starts as many instances as needed
- **Security and networking** — integrates with IAM, runs in a VPC by default, and can be configured to reach resources in your own VPC using custom security groups and network ACLs
- **Pay per use, with a flexible resource model** — pay only for requests served and compute time used (metered in 100 ms increments); you choose the memory, and Lambda allocates proportional CPU power, network bandwidth, and disk I/O
- **Workflow-ready** — coordinate multiple functions for complex or long-running tasks with AWS Step Functions

Developing and deploying a Lambda function follows a repeatable path: define a **handler** in your code (where Lambda begins running), **create the function** (code plus its configuration and resource requirements), **configure access** with an IAM role (optionally using security groups and network ACLs), **upload your code**, **test** it while reviewing logs, and **monitor** the function through **Amazon CloudWatch**, which reports metrics such as request count, latency, and errors.

WORKED EXAMPLE – SCHEDULE EC2 STOP AND START WITH LAMBDA

Suppose you want to reduce EC2 usage by stopping instances at night and starting them again before the workday. A scheduled **Amazon CloudWatch Events** event runs a Lambda function to stop the instances at, for example, 22:00 GMT; the function runs under an IAM role granting permission to stop them, and the instances enter the stopped state. Later, a second scheduled event at 05:00 GMT runs another Lambda function under an IAM role that permits starting the instances, and they return to the running state. The whole schedule is automated with no servers to manage. (The per-Region limit for concurrent Lambda runs is 1,000.)

Lambda layers let a function pull in additional code and content. A layer is a **ZIP archive** that contains libraries, a custom runtime, or other dependencies, so you can use libraries in your function without bundling them in the deployment package. Layers keep the deployment package small, help you avoid errors from packaging dependencies with your code, and let you share libraries with other developers. A function can use **up to five layers** at a time, and the total unzipped size of the function and all its layers cannot exceed the 250 MB deployment-package limit. Layers are extracted to the `/opt` directory in the runtime environment, and you add them with the `update-function-configuration` command. For Node.js, Python, and Ruby, you can even develop function code directly in the Lambda console as long as the deployment package stays under 3 MB.

TABLE 5.1 AWS Lambda limits (as of the module)

LIMIT	VALUE
Maximum memory per function	3 GB
Maximum runtime (function timeout)	15 minutes
Deployment package size (unzipped, function plus layers)	250 MB
Layers per function	Up to 5
Concurrent executions per Region	1,000
Billing granularity	100 ms increments

> **COMMON PITFALL**
>
> "Serverless" does not mean serverless. There are servers — you simply do not provision, scale, patch, or manage them, because AWS does. Remember the two hard numbers the module stresses: a Lambda function can use at most **3 GB** of memory and can run for at most **15 minutes**. Those limits, together with the 250 MB package size and the 1,000 concurrent-execution ceiling per Region, are the details a troubleshooting question tends to hinge on.

5.2 APIs and REST

Most people interact with software through a **graphical user interface (GUI)** — the AWS Management Console is a GUI for AWS services. An **application programming interface (API)** instead makes software accessible to developers in code, so programs can interact with other programs or with infrastructure. An API works as an **intermediary**: communication starts with an **API call**, a message delivered to another server as a request and returned as a response. APIs support multiple programming languages, and the **AWS Command Line Interface (AWS CLI)** is itself a tool that provides direct access to the public APIs of AWS services.

Some APIs follow the principles of **representational state transfer (REST)**. REST is a design for loosely coupled, network-based applications: communication happens over **HTTP**, and resources are exposed at specific **URLs**. Web APIs that adhere to these principles are described as **RESTful** — a very popular design that has largely replaced **SOAP** (Simple Object Access Protocol) as the standard for web services.

REST DESIGN PRINCIPLES THE MODULE HIGHLIGHTS

- **Uniform interface** — a simple, uniform way for the client to access the server, with each distinct resource reached at a single endpoint or URI
- **Stateless** — the server does not track the client's state across requests; any session information is known only to the client
- **Cacheable** — clients can cache the responses they receive from the REST server
- **Layered system** — the client may connect to an intermediate server, letting the REST server be distributed to support load balancing
- **Code on demand (optional)** — the server can pass code, such as JavaScript, for the client to run (for client-side validation or lightweight sub-requests); it remains optional and is not widely adopted

TABLE 5.2 Components of a REST request

COMPONENT	DETAIL
Endpoint	The resource location, in the form of a URL, telling the server which resource the client wants
Method	`GET` reads a resource · `POST` creates a resource · `PUT` updates an existing resource · `DELETE` deletes a resource
Header	The request's metadata — for example, `-H "Content-Type: application/json"`; authentication credentials are also sent here
Body (data)	The data the client sends to the server; typically part of `POST` and `PUT` requests, and rarely included in a `GET`

REST REQUEST WITH CURL

Client URL (**cURL**) is a command-line tool for sending and receiving files over HTTP or HTTPS — simple and useful for testing REST API access. A POST example: `curl -i -X POST -d @file.json -H "Content-Type: application/json" https://www.example.com/someresource/`. Here `-i` includes the response headers, `-X` sets the method (POST), `-d` supplies the body from a file (file.json), and `-H` passes header information such as the content type. Authentication credentials would also be supplied in the header (not shown). A typical response returns in **JSON**, for example a body of `{ "Status": "Accepted" }`.

Every REST API response includes an **HTTP status code**, and knowing what the codes mean helps you troubleshoot requests. There are hundreds of codes, but they are grouped by the **first of the three digits**, which indicates the general nature of the response — the 2xx family means success, 4xx means a client error, and 5xx means a server error.

TABLE 5.3 Commonly seen HTTP status codes

CODE	MEANING
200	Success — the request was received and accepted by the server
401	Unauthorized (client error) — authentication is required, or the provided credentials were not accepted
403	Forbidden (client error) — the request was properly made, but the server is not allowing it
404	Not Found (client error) — the resource is unavailable or could not be accessed
500	Internal Server Error — an unspecified server-side error
503	Service Unavailable — the service is temporarily unavailable

COMMON PITFALL

Match the method to the intent and the status code to its family. `GET` reads and rarely carries a body; `POST` and `PUT` carry the body because they create or update. For status codes, read the first digit: 4xx is a client-side problem (401 unauthorized, 403 forbidden, 404 not found) and 5xx is a server-side problem (500 internal error, 503 unavailable) — a distinction that quickly narrows down where a failing API call went wrong.

5.3 Amazon API Gateway

Amazon API Gateway is a fully managed AWS service that lets developers **create, publish, maintain, monitor, and secure APIs at any scale** — for example, a REST API for an application you run on AWS. It handles all the work of accepting and processing concurrent API calls: **traffic management, authorization and access control, monitoring, and API version management**. As a fully managed service it takes care of scaling, access control, and monitoring for you, and you pay only for the API calls you receive and the data transferred out.

An API built with API Gateway has a **frontend** and a **backend**. The frontend — encapsulated by **method requests and method responses** — is what client applications call to make requests. The backend — encapsulated by **integration requests and responses** — communicates with the other AWS services that provide the functionality the API exposes and take action accordingly. As applications evolve and grow, their APIs proliferate, which is exactly why monitoring and managing them matters.

BENEFITS OF AMAZON API GATEWAY

- **Low cost** — pay only for calls made to your APIs and data transferred out, with no minimum fees or upfront commitments
- **Low latency** — integrate with Amazon CloudFront to serve requests from a worldwide network of edge locations
- **Traffic control** — throttling helps backend operations withstand traffic spikes
- **Monitoring** — after deployment, a dashboard visually tracks calls to your services and shows performance metrics
- **Versioning** — run multiple versions of the same API simultaneously to quickly iterate, test, and release
- **Security** — authorize access with IAM, or use a **Lambda authorizer** to verify requests that use OAuth tokens or other mechanisms
- **Serverless APIs** — tight AWS Lambda integration lets you build completely serverless APIs, with data-transformation capabilities to generate requests in the language target services expect

TABLE 5.4 A serverless web application (API Gateway example)

COMPONENT	ROLE IN THE APPLICATION
Amazon S3	Hosts the static website content — HTML, CSS, JavaScript, and image files loaded in the user's browser
Amazon Cognito	Registers and authenticates users; provides the user directory and secures the backend API
Amazon API Gateway	Exposes the RESTful API the browser calls and routes requests to the backend
AWS Lambda	Serverless compute — receives dynamic API calls and performs the application's business logic
Amazon DynamoDB	NoSQL database that provides the persistence layer; the Lambda function stores and reads data here

COMMON PITFALL

API Gateway has no minimum fees and no upfront commitments — you pay only for API calls received and data transferred out. Its tight Lambda integration is what makes **completely serverless APIs** possible: API Gateway is the managed front door, Lambda runs the code, and there are no servers to manage on either side. In the serverless web application, S3 serves the static site, Cognito authenticates users, API Gateway exposes the REST API, Lambda runs the logic, and DynamoDB persists the data.

5.4 Containers on AWS

Just as virtual machines virtualize hardware, **containers virtualize an operating system**. Amazon EC2 runs virtual machines in which an entire OS — such as Linux or Windows — is virtualized. Containers are smaller and do **not** carry a whole operating system; instead they share a virtualized OS and run as **resource-isolated processes**, which ensures quick, reliable, and consistent deployments. A container holds everything the software needs to run: **libraries, system tools, code, and the runtime**. Put simply, a container is a method of OS virtualization, and it packages an application together with its dependencies.

BENEFITS OF USING CONTAINERS

- **Environmental consistency and portability** — code, configuration, and dependencies are packaged into a single object that runs the same in testing and production and deploys on any compute resource, regardless of the underlying software, operating system, or hardware
- **Process isolation** — each container is isolated, with no shared dependencies or incompatibilities, which improves operational efficiency
- **Resource control** — run multiple applications on the same instance and specify the exact memory, disk space, and CPU each container uses
- **Speed and scale** — containers boot quickly and have a small footprint, so you can rapidly create, terminate, and scale applications up and down
- **Developer productivity and versioning** — break an application into microservices in separate containers and upgrade each independently with no library conflicts; a Docker image's manifest (Dockerfile) lets you track versions, inspect differences, and roll back

Docker is a software platform that packages software into standardized units called **containers**, allowing you to build, test, and deploy applications quickly. Docker is installed on each server that will host containers and provides simple

commands to **build, start, or stop** them. Because a container holds everything the software needs, you can deploy and scale into any environment. Docker is best used when you want to standardize environments, reduce conflicts between language stacks and versions, use containers as a service, run microservices with standardized code deployments, or provide portability for data processing.

The deployment models illustrate the point. **On premises**, a single host operating system with Docker installed can run several applications, each in its own container; every container relies on the OS virtualization Docker provides but keeps its own isolated binaries and dependency libraries. **On AWS**, applications can be spread across multiple guest operating systems, where each guest OS is an EC2 instance — some instances run multiple containers, while an application can also run in a non-container environment without Docker. And importantly, a **single application can span multiple containers**.

CONTAINER MANAGEMENT CHALLENGES

- **Container sprawl** — the number of containers grows rapidly and their distribution is not properly managed
- **Version tracking and ownership** — keeping track of version differences and who owns each container; ECS clusters and targeted launches, plus ECR repositories, help enforce version control and ownership
- **Bin packing** — limited space on the hosts forces decisions about the best placement of each container to maximize performance
- **Zombie containers** — a container that refuses to stop because it cannot process the init command; in ECS the `initProcessEnabled` flag runs an init process as PID 1 so containers exit gracefully
- **Disappearing containers** — containers you still need but that are no longer running

COMMON PITFALL

Containers are not miniature virtual machines. A VM virtualizes hardware and carries a full guest operating system; a container virtualizes the OS and **shares** it, packaging only the libraries, tools, code, and runtime the application needs. That is why containers are smaller, boot faster, and pack more applications onto one instance — and why a single application can span multiple containers, a shape a VM-centric mental model misses.

5.5 Container services on AWS: ECS, Fargate, EKS, and ECR

Amazon Elastic Container Service (Amazon ECS) is a highly scalable, high-performance container management service that supports Docker containers and runs applications on a **managed cluster of EC2 instances**. It is built on technology refined from years of running highly scalable services, and you can launch tens — or tens of thousands — of Docker containers in seconds.

AMAZON ECS BENEFITS

- **Flexible scheduling** — use the built-in scheduler or a third-party one such as Apache Mesos, and perform task, service, or daemon scheduling
- **Integration** — the ECS APIs make it straightforward to integrate third-party solutions and your software-delivery process
- **Networking and isolation** — launches containers in your own VPC so you can use VPC security groups and network ACLs; no compute is shared with other customers
- **Security** — assign granular IAM permissions per container to restrict access to each service and control which resources a container can reach
- **Task definition** — a declarative JSON template that specifies the containers a task needs: the Docker repository and image, memory and CPU requirements, shared data volumes, and how containers are linked; launch as many tasks as you want from a single definition

AWS Fargate is a technology for Amazon ECS that lets you run containers **without having to manage servers or clusters**. Using Fargate-provisioned containers, ECS automatically scales, load balances, and schedules your containers for availability — removing the need to choose server types, decide when to scale clusters, or optimize cluster packing. ECS still handles scheduling and orchestration directed by your **task definition**, and you still own the task definition and tasks; what Fargate takes over is the placement engine, cluster management, and the underlying infrastructure such as agents and AMIs. Fargate offers **50 combinations** of CPU and memory (from 2 GB up to 8 GB of memory per vCPU); you pay by task size on a **per-second** basis with a one-minute minimum charge.

ECS has **two launch types** — the **EC2 launch type** and the **AWS Fargate launch type** — and choosing between them is a launch-time decision with no plugin to install. With the Fargate launch type you package your application in containers, specify CPU and memory, define networking and IAM policies, and launch; Fargate capabilities are native to ECS, so there are no new APIs to learn and a single cluster can run both Fargate and EC2 tasks.

USE AWS FARGATE WHEN	USE THE EC2 LAUNCH TYPE WHEN
fastest, hands-off path ▪ You want a service or app running quickly — write the task definition, choose Fargate, and go ▪ You need to migrate a monolithic app to the cloud with less focus on the cluster ▪ Your workload is highly variable and unpredictable — Fargate adds capacity automatically within your bounds, so you avoid overprovisioning	*more control* ▪ You are a heavy user of EC2 Spot or have purchased Reserved Instances ▪ Your containers run on Microsoft Windows (not currently supported by Fargate) ▪ You need granular control of instances for compliance, governance, customization, or specialized hardware

Kubernetes is open-source container management and orchestration software. It manages a cluster of EC2 compute instances and runs containers on them with processes for deployment, maintenance, and scaling. Containers run in logical groupings called **pods**, and you can run and scale one or many containers together as a pod. The Kubernetes **control plane** decides when and where to run pods, manages traffic routing, and scales pods based on utilization or metrics you define; it automatically restarts pods (or the instances they run on) when they fail, and gives each pod an IP address and a DNS name. A key advantage is portability — you can run your containerized applications anywhere, on premises or in the cloud, without changing your operational tooling.

You can run Kubernetes yourself on EC2 instances, or use **Amazon Elastic Container Service for Kubernetes (Amazon EKS)** for an automatically provisioned, **managed Kubernetes control plane**. EKS runs the Kubernetes management infrastructure across **multiple Availability Zones** to eliminate a single point of failure and is **certified Kubernetes conformant**, so existing tooling and plugins work and standard Kubernetes applications can migrate to it. Getting started takes four steps: **provision an EKS cluster** (EKS deploys the control plane automatically), **deploy worker nodes**, **connect** your preferred Kubernetes tooling to the cluster, and **run** your Kubernetes applications.

Amazon Elastic Container Registry (Amazon ECR) is a fully managed Docker container registry that makes it easy to **store, manage, and deploy** Docker container images. It is integrated with ECS: specify the ECR repository in your task definition, and ECS retrieves the appropriate images. ECR supports **Docker Registry HTTP API version 2**, so you can use Docker CLI commands and your existing tools from any Docker environment — cloud, on premises, or local. It stores images in **Amazon S3** (inheriting its high availability and durability), organizes repositories with **namespaces**, and uses **IAM** for access control. Images transfer over **HTTPS** and are automatically **encrypted at rest** with S3 server-side encryption, and third-party integrations are supported.

> **COMMON PITFALL**
>
> Keep the container services straight. **AWS Fargate** is not a separate product with its own APIs — it is a launch type within ECS, the alternative to the EC2 launch type, and one cluster can mix both. **Amazon EKS** runs the managed Kubernetes control plane for you across multiple AZs; running Kubernetes directly on your own EC2 instances is the self-managed alternative. **Amazon ECR** is the registry that stores the Docker images either approach deploys. ECS and EKS run containers; ECR stores their images.

5.6 AWS Step Functions

AWS Step Functions lets you coordinate multiple AWS services into **serverless workflows** so you can build and update distributed applications quickly while writing less code. A workflow is a **series of steps**, and the **output of one step becomes the input to the next**. For example, Step Functions can bring **AWS Lambda and Amazon ECS** together into a single feature-rich application. Step Functions translates your workflow into a **state-machine** diagram that is easy to understand, easy to explain to others, and easy to change — and it automatically triggers and tracks each step and retries when there are errors, so the application runs in order and as expected.

> **HOW AWS STEP FUNCTIONS WORKS**
>
> - **State machine** — you express a workflow as a state machine, and the console shows a graphical representation to help you visualize the logic; individual states make decisions based on their input, perform actions, and pass output to other states
> - **State types** — a state can do work (perform a **task**) or make a **choice** about which branch of the workflow to follow
> - **Tasks** — a task invokes code hosted in a function, container, or instance, and can be run as many times as needed for a period of up to 1 year
> - **Reliability** — Step Functions automatically triggers and tracks each step and retries on errors, so the application runs in order and as expected
> - **Flexibility** — change the tasks or step order, or add and delete tasks, without needing to modify any code

Because it can identify and monitor each step as it happens, Step Functions makes it easy to find and fix problems quickly. Its two headline benefits — you can **build and update distributed applications** and **write less code** — are exactly what a SysOps administrator wants when stitching several services into a reliable, repeatable process.

5.7 Choosing between containers and serverless

Containers and serverless are the two modern compute approaches this module contrasts, and choosing between them comes down to **how much of the stack you want to own**. **Lean toward containers** when portability and vendor-agnosticism matter (you want to avoid being locked in to one provider), when you want to manage the resources, policies, and security yourself, when you need to move software between environments through isolation, or for microservices, Windows containers, and EC2 Spot or Reserved Instances. **Lean toward serverless (AWS Lambda)** when you want the cloud provider to provision the infrastructure and manage capacity, when work is event-driven or intermittent, when you want to pay only for compute actually consumed with automatic scaling, and when you would rather focus on application functions than on operating servers. Neither is universally better; match the approach to the workload.

WORKED EXAMPLE – A SERVERLESS DAILY-REPORT WORKFLOW

The Mom and Pop Café wants a daily report of baking needs emailed automatically, built cost-effectively with Lambda. An **Amazon CloudWatch** event triggers a `salesAnalysisReport` Lambda function at 8:00 PM every day, Monday through Saturday. That function invokes a second Lambda function, `salesAnalysisReportDataExtractor`, which runs an analytical query (using the `PyMySQL` library) against the MySQL `mom_pop_db` database on an EC2 LAMP instance; each function has its own IAM role (`salesAnalysisReportRole` and `salesAnalysisReportDERole`) granting only the access it needs. The query result returns to the first function, which formats the report and publishes it to an Amazon SNS topic (`salesAnalysisReportTopic`); SNS then emails the report to a subscribed address. The pattern — scheduled trigger, Lambda logic, IAM-scoped permissions, and SNS delivery — is a compact template for event-driven serverless automation.

COMMON PITFALL

The choice is rarely strictly either/or. **AWS Step Functions** can coordinate Lambda functions and ECS tasks in a single workflow, so a solution can combine serverless glue with container workloads — pick the approach that fits each part of the problem, then let Step Functions orchestrate them together.

Module summary

- Serverless computing runs applications without provisioning, scaling, or managing servers; **AWS Lambda** is the AWS serverless compute service — upload code, and Lambda runs and scales it across multiple AZs with no configuration, charging only for requests and compute time in **100 ms** increments (languages include Java, Node.js, C#, Python, and Ruby).
- Key Lambda limits: **3 GB** maximum memory, **15-minute** maximum runtime, **250 MB** deployment package (function plus up to five layers, unzipped), and **1,000** concurrent executions per Region; a layer is a ZIP archive extracted to /opt.
- An **API** gives programs code-level access to one another; **RESTful** APIs communicate over HTTP and expose resources at URLs, using GET/POST/PUT/DELETE and returning HTTP status codes (200 success, 4xx client errors, 5xx server errors).
- **Amazon API Gateway** is a fully managed service to create, publish, monitor, and secure APIs at scale — pay per call and data out; it integrates with CloudFront, throttles traffic, supports IAM and Lambda authorizers, and pairs with Lambda to build completely serverless APIs.
- **Containers** virtualize the operating system (VMs virtualize hardware), sharing a virtualized OS and packaging everything the software needs; **Docker** packages software into standardized container units, and a single

application can span multiple containers. AWS container services: **Amazon ECS** (Docker management on a managed EC2 cluster via JSON task definitions), **AWS Fargate** (an ECS launch type that runs containers without managing servers or clusters), **Amazon EKS** (managed Kubernetes control plane across multiple AZs), and **Amazon ECR** (fully managed Docker registry storing images in S3).

- Choose containers for control, portability, and vendor-agnosticism (and for Windows or Spot/Reserved workloads); choose serverless when you want the provider to manage capacity for intermittent, event-driven, pay-per-use workloads.
- **AWS Step Functions** coordinates AWS services into serverless workflows expressed as a state machine, where the output of one step is the input to the next, with automatic triggering, tracking, and retries — and you can reorder or add and delete tasks without changing code.

Review questions

1. What does serverless computing remove for a SysOps administrator, and which AWS service provides it?

Answer: It removes provisioning, scaling, and managing servers or runtimes — the administrator's tasks reduce to deploying and monitoring the applications. AWS Lambda provides the serverless compute.

2. List the main AWS Lambda limits the module calls out.

Answer: Maximum memory 3 GB per function; maximum runtime 15 minutes; deployment package 250 MB unzipped (function plus up to five layers); up to five layers per function; 1,000 concurrent executions per Region; billing metered in 100 ms increments.

3. What is a Lambda layer, why use one, and where is it extracted at runtime?

Answer: A ZIP archive of libraries, a custom runtime, or other dependencies. Use it to keep the deployment package small, avoid dependency-packaging errors, and share libraries. It is extracted to the /opt directory, and a function can use up to five layers.

4. Distinguish Amazon ECS, AWS Fargate, Amazon EKS, and Amazon ECR.

Answer: ECS is Docker container management on a managed EC2 cluster; Fargate is an ECS launch type that runs containers without managing servers or clusters; EKS is a managed Kubernetes control plane run across multiple AZs; ECR is a fully managed Docker image registry that stores images in Amazon S3.

5. A workload is event-driven and intermittent, and you want the provider to manage capacity and to pay only for what you use. Containers or serverless, and why?

Answer: Serverless (AWS Lambda) — it provisions and manages the infrastructure, scales automatically, and charges only for requests and compute time. Containers fit better when you need control, portability and vendor-agnosticism, or specific workloads such as Windows containers or EC2 Spot/Reserved capacity.

6. What is AWS Step Functions, and how do the steps relate to one another?

Answer: A service that coordinates AWS services into serverless workflows expressed as a state machine. It is a series of steps in which the output of one step is the input to the next, with automatic triggering, tracking, and retries — and you can reorder or add and delete tasks without modifying any code.