MODULE 6 · STUDY NOTES

Compute

EC2 virtual machines and their lifecycle, pricing and cost optimization, containers, serverless with Lambda, and PaaS with Elastic Beanstalk

Compute is where the cloud stops being an idea and starts running your code. AWS offers more than a dozen compute services because different workloads want different trade-offs between control and convenience: Amazon EC2 hands you a resizable virtual machine and full control of the OS; containers (ECS, EKS, Fargate, ECR) share one OS to launch workloads in milliseconds; AWS Lambda removes servers entirely and bills per 100 ms of execution; and Elastic Beanstalk turns uploaded code into a running, load-balanced, auto-scaled web application. This module — the biggest in the course — walks the full EC2 launch checklist and lifecycle, then the pricing models and the four pillars of cost optimization, and finally the container, serverless, and PaaS alternatives, so you can match each workload to the compute service it actually needs.

LEARNING OBJECTIVES

- Provide an overview of different AWS compute services in the cloud
- Demonstrate why to use Amazon Elastic Compute Cloud (Amazon EC2)
- Identify the functionality in the EC2 console
- Perform basic functions in Amazon EC2 to build a virtual computing environment
- Identify Amazon EC2 cost optimization elements
- Demonstrate when to use AWS Elastic Beanstalk
- Demonstrate when to use AWS Lambda
- Identify how to run containerized applications in a cluster of managed servers

6.1 Compute services overview

The AWS compute catalog is broad: Amazon EC2 (resizable virtual machines), Amazon EC2 Auto Scaling (launch or terminate instances automatically on conditions you define), Amazon ECR (store and retrieve Docker images), Amazon ECS (Docker container orchestration), VMware Cloud on AWS (hybrid cloud without custom hardware), AWS Elastic Beanstalk (a simple way to run and manage web applications), AWS Lambda (serverless — pay only for compute time used), Amazon EKS (managed Kubernetes), Amazon Lightsail (simple websites and applications), AWS Batch (batch jobs at any scale), AWS Fargate (containers without managing servers or clusters), AWS Outposts (select AWS services inside your own data center), and the AWS Serverless Application Repository (discover, deploy, and publish serverless applications). This module concentrates on **EC2, Lambda, the container services, and Elastic Beanstalk**.

TABLE 6.1 The four broad categories of compute service

CATEGORY	CHARACTER	SERVICES
Virtual machines (IaaS)	Maximum flexibility; you keep many server-management duties — choose the OS, size, and resource capabilities. Familiar territory for on-premises IT professionals.	Amazon EC2 — one of the first AWS services and still one of the most popular
Serverless	Zero-administration compute: run code without provisioning or managing servers; pay only for compute time consumed. Enables massively scalable cloud-native architectures at lower cost than servers running 24/7.	AWS Lambda
Container-based	Multiple workloads on a single operating system; containers spin up faster than VMs, offering responsiveness.	Amazon ECS, Amazon EKS, AWS Fargate, Amazon ECR
Platform as a service	AWS manages the OS, application server, and other infrastructure; you focus on the application code.	AWS Elastic Beanstalk

CHOOSING THE OPTIMAL COMPUTE SERVICE

- The optimal service depends on the **use case** — consider your application design, your usage patterns, and which configuration settings you want to manage.
- Selecting the wrong compute solution lowers **performance efficiency**; a good starting place is simply understanding the available options.
- Best-practice loop: evaluate the available options → understand the configuration options → collect compute-related metrics → use the elasticity of resources → **re-evaluate** based on the metrics.
- Legacy code often dictates the starting architecture — but you can still evolve toward proven cloud-native designs as metrics come in.

6.2 Amazon EC2

Running servers on premises is expensive: hardware is procured on project plans rather than real usage, data centers cost money to build and staff, and enough capacity must be permanently provisioned for peak load — leaving servers idle much of the time. **Amazon EC2** provides secure, resizable virtual machines in the cloud that host the same workloads as traditional servers: application, web, database, game, mail, media, catalog, file, computing, and proxy servers. The name decodes the service: **Elastic** (change the number and size of servers easily), **Compute** (CPU and memory for running applications), **Cloud** (instances are hosted in AWS). You get full administrative control over the **guest operating system** — Windows (2008–2019) or Linux (Red Hat, SUSE, Ubuntu, Amazon Linux) — and can launch any number of instances of any size into any Availability Zone in the world, in minutes, from a template called an **AMI**. Traffic to and from each instance is controlled with **security groups**.

The Launch Instance Wizard in the console can launch an instance in as few as six clicks if you accept the defaults, but real deployments tune **nine decisions** — and each decision teaches a core EC2 concept:

TABLE 6.2 The nine launch decisions

DECISION	WHAT IT DETERMINES
1 · AMI	The template the instance is created from — OS plus any pre-installed software
2 · Instance type	Memory, CPU, storage type and amount, network performance
3 · Network settings	Which VPC (and optionally subnet) the instance deploys into; whether it gets a public IP
4 · IAM role (optional)	Permissions for software on the instance to call other AWS services
5 · User data (optional)	A script that customizes the runtime environment at first start
6 · Storage	Root volume size and type; additional volumes; delete-on-termination; encryption
7 · Tags	Key/value metadata for filtering, automation, cost allocation, access control
8 · Security group	Firewall rules — which ports, protocols, and sources may reach the instance
9 · Key pair	Credentials for connecting securely to the instance after launch

AMIs in depth. An AMI contains a template for the **root volume** (OS plus everything installed in it), **launch permissions** controlling which accounts may use it, and a **block device mapping** of volumes to attach at launch. Four sources: **Quick Start** (Linux and Windows AMIs from AWS), **My AMIs** (ones you created), **AWS Marketplace** (third-party, pre-configured for specific use cases), and **Community AMIs** (shared by anyone, not checked by AWS — use at your own risk, never in production). Building your own follows a standard path: launch an unmodified instance → configure it into a **golden instance** with your OS and application settings → capture it as a new AMI (EC2 stops the instance, snapshots the root volume, registers the snapshot) → optionally **copy the AMI to other Regions**, because an AMI can only launch instances in the Region where it is registered.

Instance types. Each type is a fixed bundle of CPU, memory, storage, and network capability, named *family.generation.size* — in `t3.large`, T is the family, 3 the generation (higher generation = more powerful, better value), large the size. Sizes scale by doubling: a `t3.2xlarge` has twice the vCPU and memory of a `t3.xlarge`, which doubles a `t3.large`. **Network bandwidth is tied to instance size** — network-intensive jobs may force a larger instance. For interdependent instances, a **cluster placement group** delivers lower latency and higher throughput between them (by default EC2 spreads instances across hardware to minimize correlated failures). **Enhanced networking** raises packets-per-second and lowers jitter and latency: the **Elastic Network Adapter (ENA)** supports up to 100 Gbps; the Intel 82599 Virtual Function interface up to 10 Gbps.

TABLE 6.3 Instance type categories

CATEGORY	EXAMPLE FAMILY	TYPICAL USE CASES
General purpose	T3 (burstable)	Baseline CPU with bursts: websites and web apps, dev environments, build servers, code repos, microservices, test/staging, line-of-business apps
Compute optimized	C5	Compute-intensive: scientific modeling, batch processing, ad serving, scalable multiplayer gaming, video encoding
Memory optimized	R5	Memory-intensive: high-performance and in-memory databases, data mining, web-scale in-memory caches, real-time big-data processing, Hadoop/Spark clusters
Storage optimized	—	Workloads needing high, fast local storage throughput
Accelerated computing	—	GPU / hardware-accelerated workloads

NETWORK PLACEMENT, IAM ROLES, AND USER DATA

- The **Region** is chosen before the wizard starts — verify the console Region first. In a **default VPC** the instance gets a public IP by default; in a nondefault subnet, the subnet's attribute decides, and a launch-time setting can override it.
- **Never store AWS credentials on an instance.** Attach an **IAM role** instead — it is held in an *instance profile*, can be attached at launch **or to a running instance**, and changes to the role propagate to every instance that carries it. Classic example: a role granting an app on EC2 access to an S3 bucket.
- **User data** scripts run with root privileges during the final boot phase of the *first* start (cloud-init on Linux; EC2Config/EC2Launch on Windows) — patch the OS, install software, fetch license keys. Used strategically, user data reduces the number of custom AMIs you must build and maintain.

AMAZON EBS – DURABLE	INSTANCE STORE – EPHEMERAL
block storage that outlives the instance state	*disks physically attached to the host*
▪ Durable, block-level volumes for throughput- and transaction-intensive workloads ▪ Stop and start the instance — the data is still there ▪ Four volume types (SSD and HDD); change type or grow size without disrupting the app ▪ Root-volume capable	▪ Temporary block storage; data **deleted** when the instance stops or terminates ▪ Data does survive a **reboot** ▪ Ideal for buffers, caches, scratch data, and data replicated across a fleet ▪ An instance-store-backed instance cannot be stopped by an API call — only terminated

Beyond block storage, an instance can mount **Amazon EFS** — a fully managed, elastic NFS file system that scales on demand to petabytes, growing and shrinking as files are added and removed — or connect to **Amazon S3**, object storage for websites, backups, archives, IoT, and big-data analytics. Neither can serve as the root volume.

WORKED EXAMPLE – TWO INSTANCES, TWO FATES

Instance 1 boots from an EBS root volume and has a 500-GB EBS data volume plus one instance-store volume. Stop it and start it again: the OS and both EBS volumes are intact; everything on the ephemeral volume is permanently lost. **Instance 2** boots from an instance-store root volume. It cannot be stopped through the EC2 API at all — it can only be terminated (or shut down from inside the OS, which terminates it) — and termination destroys everything, including the OS; the instance can never be started again. Moral: never keep valuable long-term data on instance store — use EBS, EFS, or S3.

Tags are labels of a key and an optional value that attach metadata to a resource — purpose, owner, environment. Both parts are **case-sensitive**: the `Name` tag shows in the console's instance list, but a lower-case `name` key will not. Consistent tagging strategies pay off in filtering, automation, cost allocation, and access control. **Security groups** are virtual firewalls: rules specify the protocol (TCP, UDP, ICMP), port, and allowed source — an IP address or range, another security group, or anywhere. All rules from all groups associated with an instance are evaluated together; rule changes apply automatically and immediately to every associated instance; the default group applies if you specify none. By default a security group allows all **outbound** traffic (removable), and network ACLs can additionally shield subnets. Example rule: SSH over TCP port 22 from *My IP*. **Key pairs** use public-key cryptography: AWS keeps the public key, you download the private `.pem` file — at creation, your **only** chance to save it. Windows: use the private key to decrypt the administrator password, then RDP in. Linux: the public key lands in `~/.ssh/authorized_keys` at boot, and you present the private key over SSH.

After launch, the console's description panel shows the instance's IP and DNS addresses, instance type, unique instance ID, AMI ID, VPC and subnet IDs, and more. Everything can also be done programmatically — via the AWS CLI or SDKs: `aws ec2 run-instances --image-id ami-1a2b3c4d --count 1 --instance-type c3.large --key-name MyKeyPair --security-groups MySecurityGroup --region us-east-1`. The command succeeds if it is well formed, the referenced resources exist, you have sufficient permissions, and the account has capacity; the API replies with the instance ID for use in later calls.

TABLE 6.4 Instance lifecycle states

STATE	WHAT IT MEANS
Pending	Instance is booting and being deployed to a host computer — on first launch or when a stopped instance starts
Running	Fully booted and reachable over the internet
Rebooting	Reboot via console/CLI/SDK (recommended over in-OS reboot): instance stays on the **same host**, keeps its public DNS and IP, keeps instance-store data
Stopping → Stopped	Only **EBS-backed** instances can stop; a stopped instance does not incur running-instance charges; starting it re-enters *pending* and moves it to a **new host**
Shutting down	Intermediate state between running and terminated
Terminated	Remains visible in the console for a while, but can never be connected to or recovered

HIBERNATION

- Hibernation saves the RAM contents to the **encrypted EBS root volume**; on restart, RAM is reloaded and previously running processes resume.
- Cost in hibernation is similar to a stopped instance.
- Prerequisites: certain EBS-backed Linux AMIs (e.g. Amazon Linux 2) and instance families, an **encrypted EBS root volume**, at most **150 GB RAM** — and hibernation must be **enabled at launch**; it cannot be turned on later.

Addresses, metadata, and monitoring. A reboot changes no addresses, but stop + start releases the public IPv4 back to the AWS pool — the restarted instance gets a **new public IP and external DNS hostname**, while the private IPv4 and internal DNS persist. For a persistent public address, allocate an **Elastic IP address** in the Region and associate it with the instance; it remains allocated to your account until released (soft limit: five per Region, since public IPv4 addresses are scarce). From inside the instance, **instance metadata** is readable at `http://169.254.169.254/latest/meta-data/` (a link-local address, valid only from the instance; use a browser or `curl`): public and private IP, hostname, instance ID, security groups, Region, Availability Zone — and any launch-time user data at `.../latest/user-data/`. Configuration scripts can read the metadata to configure applications or OS settings. **Amazon CloudWatch** monitors instances with near-real-time metrics and 15 months of history: **basic monitoring** (default, free) sends data every 5 minutes; **detailed monitoring** (fixed monthly rate, seven pre-selected metrics) delivers every 1 minute. RAM metrics are not collected by default, though they can be configured.

COMMON PITFALL

Two lifecycle facts the knowledge check loves: (1) only EBS-backed instances can be stopped or hibernated — instance-store-backed instances can only be terminated; (2) stop + start moves the instance to a new host and changes the public IP; reboot keeps the same host and all addresses.

LAB 3 & THE EC2 VS. RDS ACTIVITY

Lab 3 (≈35 min) builds the section's muscle memory: launch a web-server instance, monitor it, view the system log, open the security group to reach the web server, resize the instance type and its EBS root volume, explore EC2 service limits, and test termination protection. The **activity** compares deploying Microsoft SQL Server on EC2 versus Amazon RDS: RDS is the *managed* option (provisioning, install and patching, automated backups, point-in-time restore, high availability, monitoring), while EC2 offers complete control over every configuration, the OS, and the software stack; the Quick Start adds a reference architecture with proven best practices. Neither is best for every case, and cost "depends": an RDS instance costs more per hour than a similar EC2 instance, but once labor is counted, RDS is often cheaper unless you have skilled DBAs and very specific control requirements.

6.3 Amazon EC2 cost optimization

AWS offers several ways to pay for the same instance, and choosing well is one of the largest cost levers in the cloud. **Per-second billing** applies to On-Demand, Reserved, and Spot Instances that run Amazon Linux or Ubuntu. Each model earns its keep in a specific situation: On-Demand for flexibility, Reserved for predictability, Spot for interruptible scale, Dedicated Hosts and Dedicated Instances for licensing and compliance isolation.

TABLE 6.5 Pricing models, benefits, and use cases

MODEL	COMMITMENT	WHY CHOOSE IT
On-Demand	None — pay by the hour	Lowest upfront cost, most flexibility; Free Tier eligible; short-term, spiky, unpredictable workloads; application development and testing
Reserved	1- or 3-year term; full / partial / no upfront	Discounted hourly rate fixed for as long as you own the RI; substantial savings for consistent, heavy use — servers you will run for months or years
Scheduled Reserved	1-year term, recurring daily / weekly / monthly window	Capacity always available on the schedule you specify; you pay for scheduled time even if unused
Spot	None — bid on spare capacity	Runs while your bid exceeds the fluctuating Spot price; interrupted with a **2-minute notification** (terminate by default; stop or hibernate configurable); big discounts for fault-tolerant work: web servers, API backends, big data — especially apps that persist state to S3
Dedicated Hosts	Physical server dedicated to you	Bring existing per-socket, per-core, or per-VM licenses (Windows, SQL Server); satisfy compliance or regulatory requirements
Dedicated Instances	Hardware dedicated to a single customer	Physical isolation at the host hardware level from other AWS accounts, inside your VPC

Cost optimization then rests on **four pillars**, applied continuously rather than once:

THE FOUR PILLARS

- **Right size** — from ~60 instance types, provision to match the need in CPU, memory, storage, and network. Use CloudWatch metrics (including custom metrics and load testing) to find idle or oversized instances and downsize. Best practice: **right size first, then reserve**.
- **Increase elasticity** — stop or hibernate EBS-backed instances that are not in use, starting with non-production development and test environments; use automatic (and time-based) scaling to meet peaks without paying for peak capacity all day. Rule of thumb: target **20–30%** of your fleet as On-Demand or Spot.
- **Optimal pricing model** — analyze usage patterns and combine purchase types: On-Demand + Spot for variable workloads, Reserved for predictable ones. Also question the architecture itself: functionality that could run in **Lambda** may cost far less than a 24/7 VM.
- **Optimize storage choices** — resize over-provisioned EBS volumes; drop to a cheaper volume type when performance allows (Throughput Optimized HDD `st1` typically costs half as much as General Purpose SSD `gp2`); delete EBS snapshots no longer needed; and move data whose home need not be EBS to S3, with lifecycle policies aging it into cheaper tiers such as S3 Glacier.

WORKED EXAMPLE – ELASTICITY IN ONE TIME ZONE

Development and test workloads used only during business hours in a single time zone can simply be turned off nights and weekends — cutting their runtime costs by roughly **65 percent**. It is the light-switch principle: the office encourages you to flip it off on the way out. Pair it with a right-sizing pass (a 500-GB volume provisioned where 20 GB is used should shrink) and the savings compound.

Measure, monitor, improve. Done correctly, cost optimization is an ongoing process: define and enforce **cost allocation tagging** (activate tags in the Billing and Cost Management console and AWS groups usage and cost by tag — cost centers, application names, owners); define metrics, set targets, and review regularly; encourage teams to **architect for cost** using **AWS Cost Explorer** (a free tool graphing spend patterns and trends) and **AWS Trusted Advisor** (real-time best-practice guidance); and assign the responsibility for optimization to a specific individual or team — efforts with an owner succeed more often.

COMMON PITFALL

Spot Instances are not On-Demand-with-a-discount: they run only while your bid exceeds the fluctuating Spot price and can be interrupted with just a 2-minute warning. Use them where interruption is tolerable — and remember the default interruption action is *terminate*, not stop.

6.4 Container services

Containers are a method of operating-system virtualization: an application and its dependencies run in **resource-isolated processes** that share a virtualized OS, rather than each carrying a full operating system. A container packages the code, configurations, and dependencies into a single object — everything the software needs: libraries, system tools, code, runtime — delivering environmental consistency (the software runs the same on the developer's laptop, in test, and in production), operational efficiency, developer productivity, and version control. Container images are typically an **order of magnitude smaller** than VMs, and a container spins up in **hundreds of milliseconds**.

KEY TERMS

Docker
A software platform that packages software into containers, installed on each host that will run them; provides simple commands to build, start, and stop containers. Best for standardizing environments, reducing language-stack conflicts, microservices, and portable data processing.

Image
The template from which a container is created.

Kubernetes
Open source container-orchestration software: Docker runs containers on one host; Kubernetes orchestrates **many Docker hosts (nodes)**, automating provisioning, networking, load distribution, and scaling. Containers run in logical groups called **pods**, each with its own IP address and DNS name — and the same toolset works on premises and in the cloud.

Task definition
A text file — the blueprint of an ECS application — describing up to **10 containers**, the ports to open, and the data volumes to use.

Task
The instantiation of a task definition on a cluster; the ECS task scheduler places tasks across the cluster.

Cluster
The compute the tasks run on — with the EC2 launch type, a group of EC2 instances that each run the ECS **container agent**.

CONTAINERS	VIRTUAL MACHINES
isolated processes on a shared kernel ▪ All containers on a host share the Linux guest OS; the Docker engine manages their lifecycle ▪ Processes talk directly to the shared kernel, largely unaware of their container silo ▪ Portable: laptop, VM, EC2 instance, or bare metal can host them ▪ A large EC2 instance can run **hundreds** of containers	*full OS per workload* ▪ Each VM runs directly on a hypervisor with its own complete OS ▪ Process isolation comes from separate machines — e.g. three apps on three EC2 instances ▪ Heavier images, slower start-up ▪ The familiar model for on-premises virtualization

You *could* install Docker on your own fleet of EC2 instances and manage everything yourself — AWS's container services exist so you don't have to. **Amazon ECS** is a highly scalable, high-performance container management service that orchestrates Docker containers on a managed cluster: it launches up to tens of thousands of containers in seconds, monitors deployments, manages cluster state, and schedules containers (built-in or third-party schedulers such as Apache Mesos or Blox) against your CPU, memory, and availability requirements. It integrates the features EC2 users already know — Elastic Load Balancing, security groups, EBS volumes, IAM roles — and ECS clusters can run on Spot or Reserved Instances.

The cluster question: who manages the servers? Creating an ECS cluster offers three options — *Networking only (AWS Fargate)*, *EC2 Linux + Networking*, or *EC2 Windows + Networking*. With an **EC2 launch type** you specify the instances in detail (including On-Demand vs. Spot) and keep granular control of the infrastructure while ECS tracks cluster resources and picks the best server for each container. With **Fargate**, AWS manages the cluster: you package the application in containers, specify CPU and memory, define networking and IAM policies, and launch — no provisioning,

scaling, server-type selection, or cluster-packing optimization. Fargate lets you focus on designing and building the application.

Amazon EKS brings the same managed-cluster convenience to **Kubernetes**: it is certified Kubernetes conformant (existing upstream Kubernetes applications migrate without change), supports Linux and Windows containers plus community tools and popular add-ons, and automatically manages the availability and scalability of cluster nodes — detecting and replacing unhealthy control-plane nodes. AWS offers *both* ECS and EKS deliberately, to give customers flexible orchestration options. **Amazon ECR** completes the picture as a fully managed Docker container registry for storing, managing, and deploying images: it is integrated with ECS (name the ECR repository in your task definition and ECS pulls the right images), usable with EKS, and supports the Docker Registry HTTP API v2 — so ordinary Docker CLI commands work from cloud, on-premises, or local environments. Images transfer over HTTPS and are encrypted at rest with S3 server-side encryption.

> **COMMON PITFALL**
>
> Keep the roles straight: **Docker** runs containers on a single host; **Kubernetes** (and ECS) orchestrate containers across many hosts; **Fargate** is not an orchestrator at all — it is the serverless *launch type* that removes cluster management from ECS; **ECR** only stores and serves images. Exam stems test exactly these one-line jobs.

6.5 Introduction to AWS Lambda

The third approach to compute requires no servers to provision or manage at all. **AWS Lambda** is an event-driven, serverless compute service: you create a **Lambda function** — the AWS resource that contains your uploaded code — and set it to be triggered on a schedule or in response to an event. The code runs *only* when triggered, and you pay only for the compute time consumed — nothing while the code is idle.

> **WHY LAMBDA**
>
> - **Multiple languages** — Java, Go, PowerShell, Node.js, C#, Python, and Ruby, with any native or third-party library; no new frameworks to learn.
> - **Completely automated administration** — Lambda deploys the code, handles maintenance and security patches, and provides built-in logging and monitoring through CloudWatch.
> - **Built-in fault tolerance** — compute capacity is maintained across multiple Availability Zones in each Region; no maintenance windows or scheduled downtime.
> - **Orchestration of multiple functions** — AWS Step Functions builds workflows that trigger collections of Lambda functions with sequential, parallel, branching, and error-handling steps — stateful, long-running processes from stateless pieces.
> - **Pay-per-use** — billed for requests served and compute time in **100-ms increments**; scales automatically from a few requests per day to thousands per second.

Event sources are AWS services or developer applications that produce the events that run a function. Some services invoke Lambda **directly and asynchronously** — Amazon S3, Amazon SNS, CloudWatch Events. Others are **polled**: Lambda pulls records from an SQS queue or reads events from DynamoDB and runs the function per fetched message. Application Load Balancer and Amazon API Gateway invoke functions **synchronously**, and you can always invoke directly from the Lambda console, the Lambda API, an SDK, the CLI, or AWS toolkits — handy when a mobile app should call a function. Lambda automatically monitors functions through CloudWatch and stores all request logs in CloudWatch Logs for troubleshooting.

Configuring a function starts with a name, a **runtime** (e.g., a version of Python or Node.js), and an **execution role** — the IAM role the function assumes so it can touch other AWS services. Then you add a trigger (one of the event sources), add code in the built-in editor or by upload, allocate memory (**128 MB to 10,240 MB**), and optionally set environment variables, a description, the timeout, a VPC to run in, and tags. Everything lands in a **deployment package** — a ZIP archive of code and dependencies that the console manages for you (you build it yourself when using the Lambda API). **Layers** — ZIP archives of libraries, custom runtimes, or other dependencies — keep the deployment package under its size limit and share code between functions; for truly large dependencies (machine learning, data-intensive work), deploy the function as a **container image up to 10 GB**.

SCHEDULE-BASED EXAMPLE – THE "STOPINATOR"

Goal: cut EC2 costs by stopping instances at night. A CloudWatch event scheduled at 22:00 GMT triggers a Lambda function; the function runs with an IAM role that permits stopping the instances, and they enter the stopped state. At 05:00 GMT a second scheduled event triggers a start function with the matching role, and the instances return to running. The module's hands-on activity has you build exactly this: a basic Lambda function that stops an EC2 instance.

EVENT-BASED EXAMPLE – S3 THUMBNAILS

Goal: create a thumbnail for every `.jpg` or `.png` uploaded to a bucket. A user uploads an object to the source bucket → S3 detects the *object-created* event → S3 invokes the Lambda function, passing the event data → Lambda runs the function under its execution role → from the event data the function learns the source bucket and object key, reads the image, builds the thumbnail with graphics libraries, and saves it to the target bucket.

TABLE 6.6 Lambda quotas

QUOTA	VALUE	SOFT OR HARD
Concurrent executions (per Region)	1,000	Soft
Function and layer storage (per Region)	75 GB	Soft
Memory allocation (per function)	128 MB – 10,240 MB	Hard
Function timeout	15 minutes (settable 1 s – 15 min)	Hard
Deployment package size	250 MB unzipped, including layers	Hard
Container image code package	10 GB	Hard

COMMON PITFALL

Soft limits can potentially be relaxed with a justified support ticket; **hard limits cannot be increased** — no ticket makes a Lambda function run past 15 minutes or above 10,240 MB. Keep both numbers in mind when troubleshooting a deployment.

6.6 Introduction to AWS Elastic Beanstalk

AWS Elastic Beanstalk is platform as a service: the fastest way to get a web application up and running on AWS. You upload your code — the only thing you must create — and Elastic Beanstalk automatically handles **infrastructure provisioning and configuration, deployment, load balancing, automatic scaling, health monitoring, analysis and debugging, and logging**. You still remain in control: choose the instance type and database, set and adjust automatic scaling, update the application, access server log files, and enable HTTPS on the load balancer — with full access to the

underlying resources at any time. There is **no additional charge for Elastic Beanstalk itself**; you pay only for the AWS resources it creates (EC2 instances, S3 buckets), with no minimum fees and no upfront commitments.

PLATFORMS AND DEPLOYMENT PATHS

- Supported platforms: **Java, .NET, PHP, Node.js, Python, Ruby, Go, and Docker**.
- Code deploys onto the matching server: Apache Tomcat for Java, Apache HTTP Server for PHP and Python, NGINX or Apache for Node.js, Passenger or Puma for Ruby, Microsoft IIS for .NET (also Java SE, Docker, Go).
- Upload from the AWS Management Console, the AWS CLI, a Git repository, or an IDE such as Eclipse or Visual Studio.

Four benefits summarize the pitch: it is **fast and simple to start** (upload and go); it buys **developer productivity** — you write code while AWS patches and updates the underlying platform and manages the servers, databases, load balancers, firewalls, and networks; it is **difficult to outgrow**, scaling the application up and down on adjustable settings (CPU utilization metrics can trigger scaling) so peaks are handled while cost is minimized; and it grants **complete resource control** — you can seamlessly take over some or all of the infrastructure whenever you choose. The module's hands-on activity deploys a small web app with Elastic Beanstalk so you can feel this workflow end to end.

COMMON PITFALL

Exam cue: the phrases *developers quickly deploy* plus *multiple programming languages such as .NET and Java* point straight at **Elastic Beanstalk** — the module's sample exam question uses exactly those keywords, with CloudFormation, SQS, and EC2 as distractors.

Module summary

- Four compute categories: virtual machines/IaaS (EC2), serverless (Lambda), container-based (ECS, EKS, Fargate, ECR), and PaaS (Elastic Beanstalk) — the optimal choice depends on the use case, and picking wrong costs performance efficiency.
- Launching an EC2 instance means nine decisions: AMI, instance type, network settings, IAM role, user data, storage, tags, security group, key pair.
- AMIs (Regional) bundle a root-volume template, launch permissions, and block device mapping; instance types are named family.generation.size and set CPU, RAM, storage, and network performance.
- EBS volumes persist across stop/start; instance store is ephemeral (survives only a reboot); only EBS-backed instances can stop or hibernate; stop + start moves hosts and changes the public IP — use an Elastic IP for a persistent public address.
- Pricing models: On-Demand, Reserved, Scheduled Reserved, Spot (2-minute interruption notice), Dedicated Hosts, Dedicated Instances — per-second billing on Amazon Linux/Ubuntu for On-Demand, Reserved, and Spot.
- Cost optimization has four pillars — right size, increase elasticity, optimal pricing model, optimize storage — supported by cost allocation tags, Cost Explorer, and Trusted Advisor, and owned as an ongoing process.
- Containers share one OS kernel and start in hundreds of milliseconds; Docker packages them, ECS orchestrates them (EC2 launch type for control, Fargate for zero cluster management), EKS runs conformant Kubernetes, and ECR stores the images.
- Lambda runs code only when triggered — 10,240 MB and 15 minutes are hard limits, billing is per 100 ms; Elastic Beanstalk deploys code on eight platforms with no service charge beyond the underlying resources.

Review questions

1. A steady 24/7 production server, a nightly interruptible batch job, and a web app the developers just want deployed — pick a compute/pricing answer for each.

Answer: Steady server: EC2 with Reserved Instances (predictable long-term usage earns the discount). Interruptible batch: Spot Instances (2-minute-notice interruptions are tolerable, deep discounts) or Lambda if runs finish in 15 minutes. Quick deployment: Elastic Beanstalk — upload code and the platform handles provisioning, balancing, scaling, and monitoring.

2. What is the difference between stopping and terminating an instance, and which instances can be stopped at all?

Answer: Only EBS-backed instances can be stopped: a stopped instance stops incurring running charges, keeps its EBS data, and restarts (on a new host, with a new public IP). Termination is permanent — the instance shows in the console a while, but can never be connected to or recovered, and instance-store data is gone. Instance-store-backed instances cannot be stopped, only terminated.

3. Why should an application on EC2 get its AWS permissions from an IAM role rather than stored access keys?

Answer: Storing credentials on an instance is highly insecure. A role attached via an instance profile grants API permissions without any long-lived secret on the box, can be attached at launch or to a running instance, and any change to the role propagates automatically to every instance that carries it.

4. What does a user data script do, when does it run, and what is it good for strategically?

Answer: It customizes the runtime environment — patching, installing software, fetching license keys — running with root privileges during the first boot (cloud-init on Linux, EC2Config/EC2Launch on Windows). Strategically it reduces the number of custom AMIs you must build and maintain, since one base AMI plus different scripts yields different servers.

5. Why do containers start so much faster than virtual machines, and what limits does that difference imply?

Answer: A VM boots a full operating system on a hypervisor; a container is just a resource-isolated process sharing the host's OS kernel, so it starts in hundreds of milliseconds and its image is roughly an order of magnitude smaller. The flip side: all containers on a host share that one kernel, and the Docker engine must be present to manage them.

6. You are creating an Amazon ECS cluster. What is the key question that decides between the EC2 launch type and Fargate?

Answer: Do you want to manage the cluster's servers? EC2 launch type: you specify and manage the instances (even choosing On-Demand vs. Spot) for granular infrastructure control. Fargate: AWS manages the cluster — you just package containers, set CPU and memory, define networking and IAM, and launch.

7. Walk through what happens, step by step, when an image lands in the S3 source bucket of the thumbnail example.

Answer: S3 detects the object-created event → publishes it to Lambda by invoking the function with the event data → Lambda runs the function under its execution role → the function reads the source bucket and key from the event data, fetches the object, generates the thumbnail with graphics libraries, and writes it to the target bucket.

8. Which Lambda quotas are soft and which are hard — and what does that distinction mean in practice?

Answer: Soft (raisable via a justified support ticket): 1,000 concurrent executions and 75 GB of function-plus-layer storage per Region. Hard (never raisable): 10,240 MB maximum memory, 15-minute timeout, 250 MB unzipped deployment package (layers can help stay under it), 10 GB container image. If a workload needs more than a hard limit allows, it needs a different compute service.