MODULE 8 · STUDY NOTES

Introducing Containers and Container Services

How containers virtualize the OS, how Docker builds and runs images, microservices, and the AWS services — ECS, EKS, ECR, Fargate, and Elastic Beanstalk — that manage containers at scale

Containers took the idea that transformed global shipping — a standardized unit that hides the messy details of what is inside — and applied it to software. A container packages an application with everything it needs to run and shares the host machine's operating system, which makes it lighter and faster than a virtual machine and portable across every environment that runs the container platform. This module moves from that analogy through the evolution of deployment models (bare-metal → VMs → containers), into **Docker** — which builds immutable images from a Dockerfile and runs them as containers — and on to **microservices**, the architecture containers were born to support. It then covers the AWS services that manage containers at scale — **Amazon ECS** and **Amazon EKS** for orchestration, **Amazon ECR** for the image registry, **AWS Fargate** and **Amazon EC2** launch types, and **AWS Elastic Beanstalk** as the fastest path to a running multicontainer environment — and closes with two café labs: migrating the coffee-supplier application to Docker on an EC2 host, then moving to a managed setup with Elastic Beanstalk and Amazon Aurora Serverless.

LEARNING OBJECTIVES

- Describe the history, technology, and terminology behind containers
- Differentiate containers from bare-metal servers and virtual machines (VMs)
- Illustrate the components of Docker and how they interact
- Identify the characteristics of a microservices architecture
- Recognize the drivers for using container orchestration services and the AWS services that you can use for container management
- Host a dynamic website by using Docker containers
- Describe how AWS Elastic Beanstalk is used to deploy containers

8.1 Containers and the evolution of deployment models

In the physical world, a **container** is a standardized unit of storage. The term sounds generic, but its importance is clear from how it changed shipping. Before modern shipping containers, goods traveled in sacks, crates, drums, and boxes of every weight, shape, and size, loaded by hand into whatever vessel was carrying them — slow, inefficient, and costly. Uniformly sized containers made it far more efficient to load, unload, stack, and move cargo between ships, trucks, and railcars, which improved efficiency, increased productivity, and reduced costs. This is a classic use of **abstraction** to increase agility: abstraction hides the working details of a subsystem and separates concerns so that the people sending goods and the people shipping them no longer worry about how items fit into a container or how a container fits onto a vessel. The same idea makes software interoperable and platform independent.

Move that concept into computing and a container becomes a **standardized unit of software** designed to run quickly and reliably on any environment that runs the containerization platform. Containers provide **operating-system (OS)**

virtualization so you can run an application and its dependencies in **resource-isolated processes**. A container is a lightweight, standalone package that holds everything the application needs — the **application code, runtime engine, system tools, system libraries, and settings**. A single server can host several containers that all **share the host system's OS kernel**; those containers might be services within one larger enterprise application or entirely separate applications each running in its own isolated environment.

Increasing levels of abstraction track technical maturity, and containers are the latest step in the evolution of deployment models. With **bare-metal servers** you build every layer yourself — an OS on the hardware, shared libraries on the OS, then the applications — which is inefficient: hardware costs are identical at 0 or 100 percent utilization, all applications compete for the same resources, and every library version must stay in sync, so one application's incompatible update can break another. **Virtual machines (VMs)** added a virtualization platform over the host OS, giving each application its own isolated libraries inside a **full guest OS**. That improves utilization — you can stack more VMs on the same hardware and shrink your footprint — but the virtualization layer is heavy: three VMs mean three operating systems on one host, and therefore more patching, more updates, more disk, and redundant copies of the same OS and libraries.

Containers improve on virtualization by having the container runtime **share the host OS kernel** and build images from **file-system layers**. The result is lightweight, efficient, and fast: containers start up and shut down faster than VMs, which means better use of the underlying hardware. You can share libraries when it helps or isolate them per application, and because containers isolate software from the layers beneath them, their code runs **identically across environments** — from development and staging all the way to production. In short: bare-metal shares one OS and its libraries; a VM gives each app a full guest OS on a hypervisor (strong isolation, but heavy and redundant); a container isolates processes over a shared kernel (lightweight, fast, and portable).

> COMMON PITFALL
>
> A container is not a lightweight VM. A VM virtualizes the hardware and runs a **full guest OS**; a container virtualizes the **OS** and shares the host kernel. That single difference is why containers boot in a fraction of the time and take a fraction of the space — and why a host running many containers is far more efficient than the same host running many VMs.

8.2 Docker containers

Containers became popular largely because of **Docker**, a lightweight container virtualization platform. Docker provides the tooling to **create, store, manage, and run** containers, and it integrates easily with automated build, test, and deployment pipelines. Its benefits all trace back to agility: Docker is a **portable runtime application environment**; you can package an application and its dependencies into a single **immutable artifact called an image** that runs anywhere Docker is supported; you can run different application versions with different dependencies **simultaneously**; and the result is faster development and deployment cycles with better resource utilization.

KEY TERMS

Dockerfile
A plain-text file that provides the instructions to create a container image. Each instruction creates a layer.

Container image
A read-only, immutable template used to create writable containers. Highly portable and composed of layers.

Container registry
A repository of public or private images that you can pull and base other images on.

Container layer
A thin read/write layer added on top of the image to make changes to the running container.

Container
A runnable instance of an image — you can create one or many containers from the same image.

These pieces interact in a clear pipeline. An image is **built from a Dockerfile**, whose instructions create the image's layers, and it is usually **based on another image** with some customization — for example, an image built on the Ubuntu Linux image that then installs a web server, your application, and the configuration needed to run it. A **container is a runnable instance** of an image; when it is instantiated it receives a thin **read/write layer** on top of the read-only image. That architecture is what makes launching containers fast — most of the work is read-only because of the file-system layers. The writable layer keeps a running application working, but it is **not designed for long-term storage**: persistent data should live in a **volume**. Think of a container as a discrete **compute** unit, not a storage unit.

Each Dockerfile line adds a layer. A real-world Dockerfile for an Apache web server builds up bottom to top: `FROM centos:7` sets the base layer; `RUN yum -y update && yum -y install httpd` adds the OS update and Apache; `EXPOSE 80` opens the web server's port; `ADD run-httpd.sh /run-httpd.sh` with `RUN chmod -v +x /run-httpd.sh` copies a startup script and makes it executable; and `CMD ["/run-httpd.sh"]` runs that script when the container starts. Every layer is **read-only**, which makes the finished image **immutable** — and changing the Dockerfile rebuilds only the changed layers, a key reason container images are so small, lightweight, and fast.

You manage images and containers by running **Docker CLI commands** from a Bash terminal. A typical session builds and names an image with `docker build --tag node_app .`, launches a container with `docker run -d --name node_app_1 -p 8000:80 node_app` — where `-d` runs it detached and prints the container ID, `--name` names it, and `-p 8000:80` publishes container port 8000 to host port 80 so outside services can reach it — verifies it with `docker ps`, opens an interactive shell with `docker exec -it node_app_1 sh` (`-i` keeps the session open, `-t` allocates a terminal), and finally stops and removes it with `docker stop node_app_1 && docker rm node_app_1`.

TABLE 8.1 Essential Docker CLI commands

COMMAND	WHAT IT DOES
docker build	Build an image from a Dockerfile
docker images	List images on the Docker host
docker run	Launch a container from an image
docker ps	List the running containers
docker exec	Run a command (such as a shell) in a running container
docker logs	View a container's log output
docker stop / docker start	Stop a running container / start a stopped one
docker tag	Tag an image (for example, for a registry)
docker push	Push an image to a registry
docker rm / docker rmi	Remove one or more containers / images

> **COMMON PITFALL**
>
> Immutable image, ephemeral container. Images are **read-only** — you never edit one in place; you rebuild it. The **read/write layer belongs to the running container**, and it disappears with the container, so never treat it as durable storage. Persistent data belongs in a volume, and shareable images belong in a registry.

8.3 Using containers for microservices

One of the strongest forces driving container growth is the rise of **microservice architectures**. Microservices are an architectural and organizational approach to software development aimed at speeding up deployment cycles; they foster innovation and ownership and improve the maintainability and scalability of applications. The contrast with the traditional **monolith** is stark. In a monolith all processes are tightly coupled, so if one process experiences a spike in demand you must scale the **entire** architecture; adding or improving features grows more complex as the code base grows; and because so many processes are dependent and tightly coupled, a single process failure has an outsized impact on availability.

MONOLITHIC ARCHITECTURE	MICROSERVICE ARCHITECTURE
tightly coupled — one unit	*loosely coupled — independent services*
▪ All processes are packaged and deployed together ▪ A spike in one function forces scaling of the whole app ▪ Adding features gets harder as the code base grows ▪ One process failure can jeopardize the entire application	▪ Each service is an independent component ▪ Services communicate over lightweight **API operations** ▪ Each performs a single function that can support many apps ▪ Services are updated, deployed, and scaled independently

Because a well-designed microservice matches the characteristics of a container so closely, containers are the underlying technology that powers many modern microservice architectures — and, in turn, microservices let developers take full

advantage of containers. The table maps common microservices design principles to the container features that deliver them.

TABLE 8.2 Microservices design principles ↔ container characteristics

MICROSERVICES PRINCIPLE	HOW CONTAINERS DELIVER IT
Decentralized, evolutionary design; smart endpoints, dumb pipes	Each container uses the language and technology best suited to its service, and each component can be isolated and evolve separately rather than as one monolithic update
Independent products, not projects	Containers package all of a service's dependencies and libraries into a single, immutable object
Designed for failure; disposable	Gracefully shut down a failing container and create a new instance — start fast, fail fast, release file handlers; like a circuit breaker, resources are temporary and constantly change
Development and production parity	Containers keep development, testing, and production environments consistent — what works on a developer's system works the same in production, which facilitates DevOps

8.4 AWS container services

Running one or two containers on a single host is simple; managing containers **at scale** is not. Recall the shipping analogy — tracking one ship and a half-dozen containers is manageable, but scale up the number and type of transports and the logistics explode. In software, once you have tens of hosts with hundreds of containers, or hundreds of hosts with thousands of containers, you are managing an enterprise-scale, clustered environment. You must know the **state** of everything (which containers are failing, starting, or stopping), where you have room and memory to place new containers, and how to place them intelligently to maximize availability, resilience, and performance.

This is where **container orchestration platforms** earn their keep. They handle **scheduling** (when containers start and stop) and **placement** (which host has the room, memory, and resources for each container) based on the underlying hardware and the needs of the application, and they provide **service integration** with networking, persistent storage, security, monitoring, and logging. Options include native tools such as **Docker Swarm**, the open-source platform **Kubernetes**, and **Amazon ECS** — and the orchestration platform is arguably the single most important choice you make when you architect a container-based workload.

Amazon Elastic Container Service (Amazon ECS) is a highly scalable, high-performing container orchestration service that supports Docker. You use it to run and scale containerized applications on AWS, scheduling the placement of containers across a **managed cluster of EC2 instances** and scaling rapidly to thousands of containers. ECS provides its own schedulers but can also integrate with third-party schedulers, and it is tightly integrated with **IAM, Amazon CloudWatch, and Amazon Route 53**.

Amazon Elastic Container Registry (Amazon ECR) is a fully managed, cloud-based Docker image registry for storing, managing, and deploying container images. It integrates with Amazon ECS and the Docker CLI: you push images from your development machine, and ECS pulls them for production deployments, without operating your own repositories or scaling the infrastructure. It is secure — images transfer over **HTTPS**, are **encrypted at rest**, and are protected by **IAM** repository policies that restrict access to specific users, roles, or accounts.

At a high level, Amazon ECS works in four steps: **pull images** from a registry (Amazon ECR, or a third-party or private registry); **define your application** by customizing the images, adding configuration, and defining containers as short-running **tasks** or long-running **services**; **choose a launch type** — AWS Fargate or Amazon EC2, which you can mix and

match within one application; and then **let ECS manage** the containers, scaling your application and maintaining availability.

TABLE 8.3 ECS launch types – Fargate or Amazon EC2

	AWS FARGATE	AMAZON EC2
Who manages infra	AWS completely manages it — a near-serverless experience; it places tasks and sizes CPU/memory	You create and manage clusters of EC2 instances and define container placement
Choose it when	Services with wide demand swings; large workloads optimized for low overhead; small test environments; scheduled batch jobs	More predictable resource needs (or reserved instances); price-optimized large workloads; org security/compliance; excess EC2 capacity
You focus on	The tasks and application architecture, not the infrastructure	Granular control of resources, isolation, and availability without running your own cluster management

Create an ECR repository and push an image. ECR integrates with the Docker CLI for pushing, pulling, and tagging. To publish a new image: create the repository with `aws ecr create-repository --repository-name hello-world --region us-east-1`; build and tag it with `docker build -t hello-world .` then `docker tag hello-world:latest aws_account_id.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest`; authenticate Docker with `aws ecr get-login-password --region region | docker login --username AWS --password-stdin aws_account_id.dkr.ecr.region.amazonaws.com` (skippable with the ECR credential helper); and push with `docker push aws_account_id.dkr.ecr.us-east-1.amazonaws.com/hello-world:latest`.

Kubernetes is an open-source container management platform for deploying and managing containerized applications at scale, using the same toolset on premises and in the cloud. **Amazon Elastic Kubernetes Service (Amazon EKS)** is a managed service that runs the Kubernetes management infrastructure **across multiple Availability Zones** to reduce the chance of a single point of failure. EKS is **certified Kubernetes conformant**, so existing tools and plugins keep working and standard Kubernetes applications can be migrated to it, and it sets up secure, encrypted channels to your worker nodes to be **secure by default**. To use it, you provision an EKS cluster, deploy **Amazon EC2 or Fargate worker nodes**, then connect and run your Kubernetes applications.

> **COMMON PITFALL**
>
> Keep the roles straight: a **registry stores**, an **orchestrator runs**. Amazon ECR only holds images. Amazon ECS (AWS-native) and Amazon EKS (managed open-source Kubernetes) are what actually schedule, place, and run containers — and both can run tasks on either the Fargate or the EC2 launch type.

8.5 Deploying applications with Elastic Beanstalk

AWS Elastic Beanstalk is a service for deploying and scaling web applications and services. It automatically handles the deployment details — **capacity provisioning, load balancing, automatic scaling, and application health monitoring** — and provides a variety of platforms on which to build. With Beanstalk you manage all of the AWS resources that run your application as a single unit called an **environment**.

TABLE 8.4 Elastic Beanstalk components

COMPONENT	DESCRIPTION
Application	A logical collection of Elastic Beanstalk components — conceptually similar to a folder
Application version	A specific, labeled iteration of deployable code for a web application
Environment	A collection of AWS resources that runs one application version
Environment tier	Designation of the application type an environment runs; determines what resources Beanstalk provisions
Environment configuration	The parameters and settings that define how an environment and its resources behave
Saved configuration	A template used as a starting point for creating unique environment configurations
Platform	An OS, language runtime, web server, application server, and Beanstalk components you target your app to
Elastic Beanstalk CLI	A command-line interface with interactive commands to create, update, and monitor environments

The Elastic Beanstalk permissions model requires **two IAM roles** assigned at environment creation. The **service role** lets Beanstalk use other AWS services on your behalf; if you do not specify one with `--service-role`, Beanstalk creates or reuses the default `aws-elasticbeanstalk-service-role`. The **instance profile** is a container for an IAM role applied to the instances so they can retrieve application versions from Amazon S3, upload logs, and — in multicontainer Docker environments — coordinate deployments with Amazon ECS; its default is `aws-elasticbeanstalk-ec2-role`. Optional **user policies** grant people the ability to create and manage environments, with managed policies for full access (`AdministratorAccess-AWSElasticBeanstalk`) or read-only access (`AWSElasticBeanstalkReadOnly`).

Elastic Beanstalk's biggest convenience for containers is that it **coordinates deployment through Amazon ECS for you**. Going straight to ECS, you would create a task definition, create and configure a cluster (EC2 instances, VPC settings, and IAM roles), and create a service to run and maintain the tasks. With Beanstalk you write a **`Dockerrun.aws.json`** file, provide your zipped code, select the platform for your language, and launch — Beanstalk takes care of the ECS cluster creation, task definition, and running tasks, and it uses **AWS CloudFormation** to launch and update the environment's resources.

GETTING STARTED WITH AMAZON ECS	GETTING STARTED WITH ELASTIC BEANSTALK
more steps, more control ▪ Create a **task definition** ▪ Create and configure a cluster: EC2 instances, VPC settings, IAM roles ▪ Create a **service** to run and maintain the tasks	*fewer steps, fastest start* ▪ Write a **`Dockerrun.aws.json`** file and provide your zipped code ▪ Select the platform for your language ▪ Launch your application

With the **multicontainer Docker platform**, Beanstalk runs multiple Docker containers side by side on each EC2 instance in an **Auto Scaling group**; the images must be **prebuilt and stored in a public or private repository**. Beanstalk automatically creates the Amazon ECS resources — an **ECS cluster** (one per environment), an **ECS task**

definition generated from your `Dockerrun.aws.json`, an **ECS task** on every instance, the **ECS container agent**, and **ECS data volumes** for per-container logs. The `Dockerrun.aws.json` file has three sections: `AWSEBDockerrunVersion` (value `2` for multicontainer environments), `volumes` (from folders on the instance or your source bundle, mounted via `mountPoints`), and `containerDefinitions` (an array using the same formatting as an ECS task-definition file).

TABLE 8.5 Elastic Beanstalk deployment policies

POLICY	HOW IT WORKS	TRADE-OFF
All at once	Deploys the new version to every instance simultaneously	Fastest, but requires some downtime (the default policy)
Rolling	Deploys to a batch of instances at a time	Avoids downtime; runs at reduced capacity; longer
Rolling with additional batch	Launches an extra batch of instances first, then rolls	Avoids reduced availability; longer than rolling
Immutable	Launches a second Auto Scaling group; the new version goes only on new instances that must pass health checks	Safest; quick, safe rollback; longer deployment
Traffic splitting	Launches a full set of new instances and routes a portion of traffic to them	Supports canary testing; no interruption on rollback
Blue/green	Deploy to a separate environment, then swap CNAMEs to shift traffic instantly	Required for an incompatible platform version

Deployment behavior is configured through **configuration options organized into namespaces**, set in `.ebextensions` configuration files. The `aws:elasticbeanstalk:command` namespace chooses the deployment policy, timeout, batch size and type, and whether to cancel on a failed health check; the `aws:elasticbeanstalk:trafficsplitting` namespace adds the percentage of traffic to send to new instances and how long to wait before shifting more.

WORKED EXAMPLE – TRAFFIC SPLITTING FOR CANARY TESTING

A canary configuration sets `DeploymentPolicy: TrafficSplitting` under `aws:elasticbeanstalk:command`, and `NewVersionPercent: "15"` with `EvaluationTime: "10"` under `aws:elasticbeanstalk:trafficsplitting`. During the deployment, Beanstalk creates a new set of instances in a **separate temporary Auto Scaling group** and instructs the load balancer to send 15 percent of incoming traffic to them. For the 10-minute evaluation window it tracks the new instances' health; if all is well, it shifts the remaining traffic over, attaches the new instances to the original Auto Scaling group, terminates the old instances, and removes the temporary group. **Blue/green** deployment is the related pattern for a bigger change: clone the environment, deploy and test the new version there, then use the **swap-URL** option to swap CNAMEs and redirect all traffic instantly — the required approach when moving to an incompatible platform version.

COMMON PITFALL

Elastic Beanstalk does not replace Amazon ECS — it **drives** it. Beanstalk uses ECS to coordinate multicontainer Docker deployments and CloudFormation to provision resources, so you never hand-configure the cluster, task definition, or tasks. Remember two constraints: the platform version is `2` in `Dockerrun.aws.json`, and the container images must already exist in a repository before you deploy.

8.6 The café labs: Docker on EC2, then a managed service

The café owners acquired a favorite coffee supplier whose **inventory tracking application** runs on its own AWS account, and Sofía is asked to learn how it works and fold it into the café's infrastructure. In **Lab 8.1**, she migrates that application to run on Docker containers. The tasks are to prepare the development environment, analyze the existing application infrastructure, migrate the application to a Docker container, migrate the **MySQL database** to a Docker container, test the MySQL container with the node application, and add the Docker images to **Amazon ECR**. The final product: both the application and its backend data run as Docker containers on a single **Amazon EC2 instance used as a container host**.

In **Lab 8.2**, Sofía reduces the effort to maintain the application and improves its scalability by moving to managed services: she deploys the application website with **AWS Elastic Beanstalk** and retires the container-based MySQL database in favor of **Amazon Aurora Serverless**. The work includes configuring subnets for Amazon RDS and Elastic Beanstalk, setting up the Aurora Serverless database, reviewing the container image, configuring communication between the container and the database, creating the database objects and seeding supplier data, reviewing the IAM policy and role for Elastic Beanstalk, creating the Elastic Beanstalk application, and configuring an Amazon API Gateway proxy.

Amazon Aurora is a fully managed relational database engine **compatible with MySQL and PostgreSQL** and part of **Amazon RDS**, combining the performance and availability of high-end commercial databases with the simplicity and cost-effectiveness of open-source databases. **Aurora Serverless** is an on-demand configuration that **automatically scales compute up and down** based on traffic and **shuts down when not in use** — a simple, cost-effective fit for **infrequent, intermittent, or unpredictable** workloads, and a better choice than a self-managed database container.

The module's sample exam question. *A cloud architect wants to migrate a web application to containers. The team does not have much experience with AWS or containers, but the architect wants to get them started quickly so they can experiment. Which solution is best?* Isolate the key phrases first: **not much experience**, **started quickly**, **experiment**. The answer is to **use Elastic Beanstalk to launch a multicontainer Docker environment** — Beanstalk sets up a working environment without the team having to configure instances, networking, or IAM permissions. Using Amazon ECR (hosting images built from scratch) still expects the team to build images and would be more useful starting from existing images; configuring EC2 instances with Docker, or standing up an ECS cluster of EC2 instances, both demand more AWS and container expertise than a team just getting started should take on.

Module summary

- A container is a standardized, lightweight unit of software that packages an application with its code, runtime, tools, libraries, and settings and shares the host OS kernel — an example of abstraction that increases agility and portability.
- Deployment models evolved bare-metal → VMs → containers. Unlike a VM's full guest OS on a hypervisor, a container shares the host kernel, so it is lighter, starts and stops faster, and runs identically from development to production.
- Docker is the platform that popularized containers; it builds an immutable, layered image from a Dockerfile (each instruction adds a read-only layer; only changed layers rebuild) and runs it as a container with a thin read/write layer — a compute unit, not storage, so persist data in a volume and store images in a registry.
- Microservices — small, independent, single-function components communicating over lightweight APIs — are a primary driver of container adoption; containers deliver their goals of technology freedom, packaged dependencies, disposability, and dev/prod parity.

- Orchestration handles scheduling, placement, and service integration for containers at scale; AWS container services are Amazon ECS (managed orchestration), Amazon EKS (managed Kubernetes), and Amazon ECR (managed registry), and ECS/EKS tasks run on the AWS Fargate (serverless) or Amazon EC2 (self-managed) launch type.
- AWS Elastic Beanstalk deploys and scales web applications as environments (capacity, load balancing, auto scaling, health monitoring) and, for containers, stands up a multicontainer Docker environment via ECS and CloudFormation from a Dockerrun.aws.json file; its deployment policies — all at once, rolling, rolling with additional batch, immutable, traffic splitting (canary), and blue/green — trade speed for safety.

Review questions

1. Why is a container more lightweight than a virtual machine?

Answer: A VM runs a full guest operating system on a hypervisor, so a host with three VMs runs three OSes — heavy, redundant, and slow to boot. A container shares the host's OS kernel and isolates only the application's processes, so it starts and stops faster and takes far less space.

2. Trace how a Dockerfile becomes a running container, and name the role of a registry.

Answer: Each Dockerfile instruction adds a read-only layer, producing an immutable image. A container is a runnable instance of that image with a thin read/write layer added at launch. A registry (such as Amazon ECR) stores images so they can be pulled to any host that runs Docker.

3. How do a monolithic and a microservice architecture differ in how they scale and fail?

Answer: In a monolith the processes are tightly coupled, so a spike in one function forces you to scale the whole application, and one process failure can take down the app. Microservices are independent components communicating over lightweight APIs; each can be updated, deployed, and scaled on its own, and a single failure is contained.

4. Distinguish Amazon ECS, Amazon EKS, and Amazon ECR, and name the two ECS launch types.

Answer: ECS is AWS's fully managed orchestration service for Docker containers; EKS is a managed service that runs open-source Kubernetes on AWS across multiple Availability Zones; ECR is a fully managed registry that stores Docker images. ECS and EKS run containers on either the AWS Fargate (near-serverless) or Amazon EC2 (self-managed) launch type; ECR stores the images they pull.

5. When would you choose the AWS Fargate launch type over Amazon EC2?

Answer: Choose Fargate for a near-serverless experience — services with wide demand swings, low-overhead or small test environments, and scheduled batch jobs — when you would rather not manage infrastructure. Choose EC2 when you need control over the instances: predictable resource needs, reserved-instance or price optimization, compliance requirements, or excess EC2 capacity.

6. How does Elastic Beanstalk simplify deploying a multicontainer Docker application, and what does it use under the hood?

Answer: You provide a Dockerrun.aws.json file and your zipped code, choose a platform, and launch — Beanstalk provisions capacity, load balancing, auto scaling, and health monitoring automatically. Under the hood it uses Amazon ECS to coordinate the containers (creating the cluster, task definition, and tasks) and AWS CloudFormation to launch the resources; the images must be prebuilt in a repository. In Lab 8.2 the café then moves its database from a container to Amazon Aurora Serverless — a managed MySQL/PostgreSQL-compatible engine that auto-scales and shuts down when idle — to cut maintenance and improve scalability.