Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

SERVERLESS APPLICATION DEVELOPMENT

AWS LAMBDA WITH API GATEWAY – STUDY MATERIAL

INTRODUCTION TO AWS LAMBDA AND API GATEWAY

What is AWS Lambda?

AWS **Lambda** is a **serverless compute service** that allows you to **run code without provisioning or managing servers**. It executes functions **in response to events** and automatically scales as needed.

What is Amazon API Gateway?

Amazon **API Gateway** is a **fully managed service** that allows developers to create, publish, and manage secure APIs at scale. It acts as a **bridge between clients and backend services**, including **AWS Lambda**.

Why Use AWS Lambda with API Gateway?

✓ **Serverless and Cost-Effective** – No need to manage infrastructure.

✓ **Auto-Scalable** – Lambda scales automatically based on incoming requests.

✓ **Secure** – Supports authentication and authorization with **IAM,**

Cognito, and JWT tokens.

✓ **Multiple Integration Options** – Supports **REST, HTTP, and WebSocket APIs**.

✓ **Low Latency** – API Gateway provides caching to improve performance.

📌 **Example Use Case:**

A **serverless web application** where an API Gateway routes requests to an **AWS Lambda function**, which processes and retrieves **user data from DynamoDB**.

CHAPTER 1: KEY COMPONENTS OF AWS LAMBDA AND API GATEWAY

Component	**Description**
Lambda Function	Code execution unit that runs in response to an event.
API Gateway	Manages and routes API requests to backend services like Lambda.
IAM Role	Defines permissions for Lambda and API Gateway interactions.
Invocation Type	Supports synchronous (request/response) and asynchronous execution.
API Gateway Stages	Allows versioning and deployment of different API versions (e.g., Dev, Staging, Production).
API Keys & Throttling	Controls API usage and prevents overloading with rate limits.

CHAPTER 2: SETTING UP AWS LAMBDA WITH API GATEWAY

Step 1: Create an AWS Lambda Function

1. Open **AWS Console** → Navigate to **AWS Lambda**.
2. Click **"Create Function"** → Choose **"Author from Scratch"**.
3. Enter **Function Name**: UserDataHandler.
4. Select **Runtime**: Python 3.9.
5. Select **Execution Role**: Create a new role with **basic Lambda permissions**.
6. Click **"Create Function"**.

Step 2: Write Lambda Function Code

Modify the function code to **return user data**:

```
import json


def lambda_handler(event, context):
  # Sample response
  response = {
    "statusCode": 200,
    "body": json.dumps({"message": "User data retrieved successfully!", "user_id": 123})
  }
  return response
```

✓ Click **Deploy** to save changes.

✓ Test function execution using **"Test"**.

📌 **Expected Outcome:**

✓ Lambda **executes and returns a JSON response**.

Chapter 3: Creating an API with API Gateway

Step 1: Create a New API

1. Open **AWS API Gateway Console**.
2. Click **"Create API"** → Choose **"REST API"**.
3. Select **"New API"** → Name it UserAPI.
4. Choose **Endpoint Type:** Regional.
5. Click **Create API**.

Step 2: Create a New Resource & Method

1. Click **Actions** → Select **"Create Resource"**.
2. **Resource Name:** user.
3. Click **Create Resource**.
4. Click **"Create Method"** → Choose **"GET"**.
5. Select **Integration Type:** Lambda Function.
6. Enter **Lambda Function Name**: UserDataHandler.
7. Click **Save** → Confirm permissions to allow API Gateway to invoke Lambda.

📌 **Expected Outcome:**

✔ API Gateway **routes GET requests to AWS Lambda**.

Chapter 4: Deploy API and Test the Endpoint

Step 1: Deploy API

1. Click **Actions** → Select **"Deploy API"**.
2. **Stage Name:** dev.
3. Click **Deploy**.

Step 2: Get API Endpoint URL

1. After deployment, **copy the API Invoke URL**:
2. https://abcd1234.execute-api.us-east-1.amazonaws.com/dev/user
3. Open a **web browser** or use **Postman** to send a GET request:
4. curl -X GET https://abcd1234.execute-api.us-east-1.amazonaws.com/dev/user

📌 **Expected Outcome:**

✔ The API **invokes the Lambda function and returns the JSON response**.

Chapter 5: Securing the API with IAM and API Keys

1. Enable IAM Authentication

1. Open **API Gateway Console** → Select **UserAPI**.
2. Click **"Method Request"** under the **GET /user method**.

3. Set **Authorization** to AWS_IAM.
4. Attach an **IAM Policy** that grants permission to invoke the API.

📌 **Expected Outcome:**
✓ Only **authenticated IAM users** can call the API.

---

2. Enable API Key Authentication

1. Open **API Gateway Console** → Click **"Usage Plans"**.
2. Create a **Usage Plan** with request limits.
3. Click **"Create API Key"** → Associate it with the UserAPI.
4. Require **API Key** in the **Method Request Settings**.

📌 **Expected Outcome:**
✓ The API **requires an API key** for access.

CHAPTER 6: LOGGING & MONITORING API GATEWAY AND LAMBDA

✓ **Enable AWS CloudWatch Logs for API Gateway**:

1. Open **API Gateway Console** → Select UserAPI.
2. Click **Settings** → Enable **CloudWatch Logs**.
3. Monitor API logs in **CloudWatch Logs Console**.

✓ **Enable AWS CloudWatch Logs for Lambda**:

1. Open **AWS Lambda Console** → Select UserDataHandler.
2. Click **"Monitor"** → View **invocation logs, execution duration, and errors**.

📌 **Expected Outcome:**

✓ Logs **track API requests, Lambda execution, and debugging information**.

Chapter 7: Best Practices for AWS Lambda and API Gateway

✓ **Use Caching in API Gateway** – Enable caching to reduce redundant Lambda calls.

✓ **Optimize Lambda Memory and Execution Time** – Allocate appropriate memory to **reduce cold starts**.

✓ **Implement Throttling** – Prevent API abuse using **rate limiting**.

✓ **Secure API Endpoints** – Use **JWT, IAM roles, and API keys** for access control.

✓ **Use AWS X-Ray** – Enable tracing to analyze API latency and dependencies.

📌 **Example:**

A **banking application** secures its API with **IAM roles and OAuth-based authentication**.

Chapter 8: Real-World Use Cases for AWS Lambda & API Gateway

1. Serverless Web Applications

✓ API Gateway + Lambda handle **user authentication, payments, and real-time notifications**.

2. Data Processing & ETL Pipelines

✓ API triggers Lambda to process data from **S3, DynamoDB, or third-party sources**.

3. Chatbots & AI Assistants

✓ API Gateway routes messages to **Lambda functions powered by AI/ML models**.

4. IoT Device Communication

✓ IoT devices send **real-time data** to API Gateway, which triggers Lambda for processing.

Conclusion: Mastering AWS Lambda with API Gateway

By using **AWS Lambda and API Gateway**, businesses can:

✅ **Build fully serverless applications with no infrastructure management**.

✅ **Create secure, scalable, and cost-efficient APIs**.

✅ **Monitor and optimize API performance using CloudWatch and X-Ray**.

✅ **Secure endpoints with authentication and throttling**.

Final Exercise:

1. **Create a new API that interacts with DynamoDB to store and retrieve user data.**
2. **Implement authentication using AWS Cognito.**
3. **Enable API rate limiting using API Gateway Usage Plans.**

SERVERLESS FRAMEWORK – STUDY MATERIAL

INTRODUCTION TO SERVERLESS FRAMEWORK

What is Serverless Framework?

The **Serverless Framework** is an **open-source development tool** that simplifies **deploying and managing serverless applications** across multiple cloud providers, such as **AWS, Azure, and Google Cloud**. It allows developers to define and deploy **serverless applications using a simple configuration file** instead of manually setting up resources.

Why Use Serverless Framework?

✔ **Simplifies Deployment** – Automates AWS Lambda, API Gateway, DynamoDB, and more.

✔ **Multi-Cloud Support** – Works with AWS, Google Cloud, Azure, and other providers.

✔ **Infrastructure as Code (IaC)** – Uses **YAML configuration** for defining resources.

✔ **Built-in Monitoring & Debugging** – Provides logs, metrics, and tracing.

✔ **Plugin Ecosystem** – Extensible with **community and enterprise plugins**.

📌 **Example Use Case:**

A **serverless e-commerce API** where API Gateway routes requests to **AWS Lambda**, which then retrieves product data from **DynamoDB**.

CHAPTER 1: KEY COMPONENTS OF SERVERLESS FRAMEWORK

Component	Description
serverless.yml	Configuration file that defines functions, resources, and event triggers.
AWS Lambda	Serverless compute function that executes business logic.
API Gateway	Routes HTTP requests to Lambda functions.
DynamoDB	NoSQL database that stores application data.
IAM Roles	Grants permissions to Lambda for accessing AWS services.
Plugins	Extends framework functionality (monitoring, deployment, testing).

CHAPTER 2: INSTALLING SERVERLESS FRAMEWORK

Step 1: Install Node.js and NPM

The Serverless Framework requires **Node.js**. Install it if you haven't already:

1. Download and install **Node.js** from Node.js Official Website.
2. Verify installation:
3. node -v
4. npm -v

Step 2: Install Serverless Framework Globally

1. Install via NPM:
2. npm install -g serverless

3. Verify installation:
4. serverless -v

📌 **Expected Outcome:**
✔ **Serverless CLI** is installed and ready to use.

CHAPTER 3: SETTING UP A SERVERLESS PROJECT

Step 1: Create a New Serverless Project

serverless create --template aws-nodejs --path my-serverless-app

cd my-serverless-app

📌 **Expected Outcome:**
✔ A **new Serverless project folder** is created with a serverless.yml file.

Step 2: Configure AWS Credentials

serverless config credentials --provider aws --key YOUR_AWS_ACCESS_KEY --secret YOUR_AWS_SECRET_KEY

📌 **Expected Outcome:**
✔ AWS credentials are **stored and configured for deployment**.

CHAPTER 4: WRITING A SIMPLE SERVERLESS FUNCTION

Step 1: Define Serverless Configuration (serverless.yml)

Modify serverless.yml to define a **Lambda function triggered by an HTTP request**:

service: my-serverless-app

```yaml
provider:
  name: aws
  runtime: nodejs14.x
  region: us-east-1

functions:
  hello:
    handler: handler.hello
    events:
      - http:
          path: hello
          method: get
```

📌 **Configuration Breakdown:**

✔ **Runtime:** Uses Node.js 14.x.

✔ **Function Name:** hello.

✔ **Event Trigger:** HTTP GET request at /hello.

Step 2: Write Lambda Function Code (handler.js)

Modify handler.js:

```javascript
module.exports.hello = async (event) => {
  return {
    statusCode: 200,
```

```
  body: JSON.stringify({ message: "Hello from Serverless Framework!" }),
 };
};
```

📌 **Expected Outcome:**
✓ Lambda function **returns a JSON response**.

Chapter 5: Deploying the Serverless Application

Step 1: Deploy to AWS

Run the following command to deploy the function:

```
serverless deploy
```

📌 **Expected Outcome:**
✓ Serverless Framework **deploys the Lambda function and API Gateway**.

Step 2: Get API Endpoint URL

After deployment, you'll see an output like this:

```
Service Information
endpoint: GET - https://abcd1234.execute-api.us-east-1.amazonaws.com/dev/hello
```

📌 **Test API:**
Use **cURL** or a web browser to test the endpoint:

```
curl -X GET https://abcd1234.execute-api.us-east-1.amazonaws.com/dev/hello
```

✓ The API **returns a JSON response**.

CHAPTER 6: MANAGING SERVERLESS APPLICATIONS

1. Invoke Lambda Function Locally

serverless invoke local --function hello

📌 **Expected Outcome:**

✓ Executes the Lambda function locally before deploying.

---

2. View Logs from CloudWatch

serverless logs -f hello --tail

📌 **Expected Outcome:**

✓ Displays **real-time logs** for debugging.

---

3. Remove Serverless Application

serverless remove

📌 **Expected Outcome:**

✓ **Deletes deployed resources** from AWS.

CHAPTER 7: EXTENDING SERVERLESS FRAMEWORK WITH PLUGINS

1. Install Serverless Plugin for Monitoring

serverless plugin install -n serverless-plugin-tracing

✓ Adds **AWS X-Ray tracing** to monitor function execution.

---

2. Use Serverless Offline Plugin for Local API Testing

serverless plugin install -n serverless-offline

serverless offline start

✔ Runs API Gateway **locally without deploying**.

Chapter 8: Best Practices for Serverless Applications

✔ **Use Environment Variables** – Store sensitive data securely using AWS Secrets Manager.

✔ **Optimize Cold Starts** – Use **Provisioned Concurrency** for lower latency.

✔ **Implement Security** – Use **IAM Roles, API Gateway authentication, and VPC**.

✔ **Enable Logging and Monitoring** – Use AWS **CloudWatch and X-Ray**.

✔ **Minimize Deployment Size** – Use **Webpack or Serverless Bundle** for code optimization.

📌 **Example:**
A **fintech company** secures its **API with IAM and logs all transactions using AWS X-Ray**.

Chapter 9: Real-World Use Cases for Serverless Framework

1. RESTful API for Web Apps

✔ API Gateway + Lambda handles **user authentication and CRUD operations**.

2. Data Processing Pipelines

✔ Lambda **processes and stores data in DynamoDB**.

3. IoT Event Processing

✔ AWS IoT triggers **Lambda functions** to analyze real-time device data.

4. AI/ML Model Deployment

✔ SageMaker + Lambda provide **AI-powered recommendations** via API Gateway.

Conclusion: Mastering Serverless Framework

By using **Serverless Framework**, businesses can:

✅ **Build serverless applications quickly and efficiently**.

✅ **Deploy and manage Lambda, API Gateway, and DynamoDB effortlessly**.

✅ **Secure and monitor serverless workloads with IAM, CloudWatch, and X-Ray**.

✅ **Optimize performance and cost by reducing infrastructure overhead**.

Final Exercise:

1. **Create a serverless API to store and retrieve customer orders from DynamoDB.**
2. **Use the Serverless Framework to deploy a Lambda function triggered by S3 uploads.**
3. **Implement API authentication using AWS Cognito.**

MICROSERVICES ON AWS

AWS ECS (ELASTIC CONTAINER SERVICE) – STUDY MATERIAL

INTRODUCTION TO AWS ECS

What is AWS ECS?

AWS **Elastic Container Service (ECS)** is a **fully managed container orchestration service** that allows developers to deploy, manage, and scale containerized applications. It supports **Docker containers** and integrates with **AWS services like EC2, Fargate, Load Balancer, IAM, and CloudWatch**.

Why Use AWS ECS?

✓ **Serverless & Managed Service** – AWS manages container orchestration.

✓ **Supports Both EC2 and Fargate** – Choose between **self-managed EC2 instances** or **AWS-managed Fargate for serverless deployments**.

✓ **Highly Scalable** – Automatically scales container workloads.

✓ **Integration with AWS Services** – Works with **CloudWatch, IAM, ALB, and DynamoDB**.

✓ **Secure and Reliable** – Supports **IAM roles, VPC networking, and encryption**.

📌 **Example Use Case:**

A **web application** uses AWS ECS with **Fargate** to **run microservices** that scale automatically.

CHAPTER 1: KEY COMPONENTS OF AWS ECS

Component	Description
Cluster	Logical grouping of ECS container instances.
Task Definition	Blueprint that defines container configuration (CPU, memory, ports, image).
Task	Running instance of a container defined in a Task Definition.
Service	Manages tasks and maintains a desired count.
Launch Type	Defines whether containers run on **EC2** or **Fargate**.
Container Agent	Runs on EC2 instances to connect them with ECS.
ECS Fargate	Serverless container execution without managing EC2.
ECS EC2	Deploys containers on self-managed EC2 instances.

CHAPTER 2: SETTING UP AWS ECS CLUSTER

Step 1: Create an ECS Cluster

1. Open **AWS ECS Console** → Click **"Create Cluster"**.
2. Choose **"Networking only"** for **Fargate** or **"EC2 Linux + Networking"** for **EC2-based deployment**.
3. **Cluster Name:** MyECSCluster.
4. Click **"Create"**.

📌 **Expected Outcome:**
✔ A **new ECS cluster is created**.

Step 2: Create an ECS Task Definition

1. Open **AWS ECS Console** → Click **"Task Definitions"** → **Create New Task Definition**.
2. **Launch Type:** Choose **Fargate** (or EC2).
3. **Task Role:** Select an **IAM role** with permissions for ECS.
4. Define **Container Configuration**:
   - **Container Name:** my-web-app.
   - **Image:** nginx:latest.
   - **CPU & Memory:** 512 MB, 0.25 vCPU.
   - **Port Mapping:** 80:80.
5. Click **"Create"**.

📌 **Expected Outcome:**
✔ A **task definition is created**, defining how containers will run.

Chapter 3: Running Containers with AWS ECS Service

Step 1: Deploy a Service on ECS

1. Open **AWS ECS Console** → Select **"Create Service"**.
2. Choose **Launch Type:** Fargate (or EC2).
3. **Cluster:** MyECSCluster.
4. **Task Definition:** Select my-web-app.

5. **Service Name:** MyService.
6. Set **Number of Tasks:** 2 (for high availability).
7. **Load Balancer Integration (Optional)**:
   - Create an **Application Load Balancer (ALB)**.
   - Set **target group** to route traffic to ECS tasks.
8. Click **"Create Service"**.

📌 **Expected Outcome:**
✔ ECS **launches two container instances** using the task definition.

Step 2: Verify Running Tasks

1. Open **AWS ECS Console** → Navigate to **Clusters** → **MyECSCluster** → **Tasks**.
2. Confirm that **two running tasks** are displayed.
3. Open the **ALB URL** to check the application.

📌 **Expected Outcome:**
✔ The application **runs successfully on ECS and is accessible via ALB**.

CHAPTER 4: MANAGING ECS SERVICES

1. Scaling ECS Services

✔ To **scale up or down**, go to **ECS Console** → **MyService** → **Edit**.
✔ Increase or decrease **the desired task count**.
✔ ECS will automatically launch or stop tasks.

📌 **Expected Outcome:**
✓ ECS **adjusts container count dynamically**.

---

2. Deploying New Versions of the Application

✓ Update the **task definition** with a **new container image version** (e.g., nginx:latest → nginx:1.20).
✓ **Restart ECS service** to deploy the new version.

📌 **Expected Outcome:**
✓ ECS **pulls the latest image and replaces running containers**.

---

3. Logging and Monitoring ECS Tasks

✓ Enable **CloudWatch Logs** to track container logs:

```
logConfiguration:
 logDriver: awslogs
 options:
  awslogs-group: my-log-group
  awslogs-region: us-east-1
  awslogs-stream-prefix: ecs
```

✓ Monitor container **CPU and memory usage** in **ECS Metrics**.

📌 **Expected Outcome:**
✓ Logs and metrics help **debug errors and monitor performance**.

Chapter 5: Securing ECS Workloads

✓ **IAM Roles** – Assign least privilege permissions to ECS tasks.

✓ **VPC and Security Groups** – Restrict container access using private subnets.

✓ **Encryption** – Store sensitive data in **AWS Secrets Manager** or **Parameter Store**.

📌 **Example:**

A **finance company** secures its ECS cluster with **private subnets and IAM role-based access**.

Chapter 6: Comparing AWS ECS Launch Types

Feature	ECS EC2	ECS Fargate
Infrastructure Management	Self-managed EC2 instances	Fully managed by AWS
Scaling	Manual or Auto Scaling Groups	Auto-scaling based on demand
Networking	Uses EC2 VPC networking	Runs in AWS-managed VPC
Cost Model	Pay for EC2 instances	Pay per vCPU & memory usage

📌 **Use Fargate** for **serverless, auto-scaling applications**.

📌 **Use EC2** for **full control over infrastructure**.

Chapter 7: Real-World Use Cases for AWS ECS

1. Microservices Architecture

✓ Deploy multiple **independent services** with ECS and API Gateway.

2. Machine Learning Model Deployment

✔ Use ECS to **serve ML models** via REST API (e.g., TensorFlow, PyTorch).

3. Batch Processing Workloads

✔ Schedule ECS tasks for **data transformation and analytics**.

4. CI/CD Pipelines for Containers

✔ Automate ECS deployments with **AWS CodePipeline and CodeDeploy**.

CONCLUSION: MASTERING AWS ECS FOR CONTAINER ORCHESTRATION

By using **AWS ECS**, businesses can:

✅ **Deploy and manage containers easily with EC2 or Fargate**.
✅ **Scale applications dynamically based on workload**.
✅ **Integrate seamlessly with AWS services like ALB, CloudWatch, and IAM**.
✅ **Improve security with VPC, IAM roles, and encryption**.

FINAL EXERCISE:

1. **Deploy a Node.js microservice on AWS ECS with Fargate.**
2. **Configure an ALB to distribute traffic across multiple ECS tasks.**
3. **Use AWS CloudWatch to monitor and debug container logs.**

AWS Fargate for Serverless Containers – Study Material

Introduction to AWS Fargate

What is AWS Fargate?

AWS **Fargate** is a **serverless compute engine** for containers that allows developers to run containerized applications **without managing EC2 instances**. It works with both **Amazon ECS (Elastic Container Service) and Amazon EKS (Elastic Kubernetes Service)** to deploy and scale containerized workloads seamlessly.

Why Use AWS Fargate?

✓ **No Server Management** – No need to provision or manage EC2 instances.

✓ **Auto-Scaling** – Automatically scales containers based on demand.

✓ **Pay-Per-Use Pricing** – Only pay for vCPU and memory usage.

✓ **Seamless Integration** – Works with **ECS, EKS, ALB, IAM, and CloudWatch**.

✓ **Enhanced Security** – Containers run in an **isolated environment**, improving security.

📌 **Example Use Case:**
A **serverless web application** where Fargate hosts **microservices running in containers** without requiring EC2 instances.

Chapter 1: Key Components of AWS Fargate

Component	Description
Cluster	Logical grouping of ECS services and tasks.

Task Definition	Blueprint defining the container settings (CPU, memory, networking, image, etc.).
Task	A running instance of a container.
Service	Manages and maintains a specified number of running tasks.
Load Balancer (ALB/NLB)	Routes traffic to containers.
Security Groups & IAM Roles	Define security and access control for Fargate tasks.

Chapter 2: Setting Up AWS Fargate for Container Deployment

Step 1: Create an ECS Cluster for Fargate

1. Open **AWS ECS Console** → Click **"Create Cluster"**.
2. Choose **"Networking Only (Fargate)"**.
3. **Cluster Name:** FargateCluster.
4. Click **"Create"**.

📌 **Expected Outcome:**
✔ A **new ECS cluster** is created to run Fargate tasks.

Step 2: Create a Task Definition for Fargate

1. Open **AWS ECS Console** → Click **"Task Definitions"** → **Create New Task Definition**.
2. Choose **"Fargate"** as the launch type.

3. **Task Name:** FargateTask.
4. Define **Container Configuration**:
   - **Container Name:** web-container.
   - **Image:** nginx:latest.
   - **CPU & Memory:** 512 MB, 0.25 vCPU.
   - **Port Mapping:** 80:80.
5. Click **"Create"**.

📌 **Expected Outcome:**
✔ The **task definition specifies how the container runs on Fargate**.

Step 3: Deploy a Service on AWS Fargate

1. Open **AWS ECS Console** → Select **"Create Service"**.
2. Choose **Launch Type:** Fargate.
3. **Cluster:** FargateCluster.
4. **Task Definition:** Select FargateTask.
5. **Service Name:** FargateService.
6. Set **Number of Tasks:** 2 (for high availability).
7. **Load Balancer Integration (Optional)**:
   - Create an **Application Load Balancer (ALB)**.
   - Set **target group** to route traffic to ECS tasks.
8. Click **"Create Service"**.

📌 **Expected Outcome:**

✓ **Fargate launches two container instances** using the defined task definition.

Step 4: Verify Running Tasks

1. Open **AWS ECS Console** → Navigate to **Clusters** → **FargateCluster** → **Tasks**.
2. Confirm that **two running tasks** are displayed.
3. Open the **ALB URL** to check the application.

📌 **Expected Outcome:**

✓ The application is **accessible via the Load Balancer**, running on **AWS Fargate**.

CHAPTER 3: MANAGING AWS FARGATE SERVICES

1. Scaling Fargate Services

✓ To **scale up or down**, go to **ECS Console** → **FargateService** → **Edit**.
✓ Increase or decrease **the desired task count**.
✓ ECS **automatically launches or stops Fargate tasks** based on scaling policies.

📌 **Expected Outcome:**

✓ AWS Fargate **scales container workloads dynamically**.

---

2. Deploying New Versions of a Containerized Application

✓ Update the **task definition** with a **new container image version** (e.g., nginx:latest → nginx:1.20).

✓ **Restart ECS service** to deploy the new version.

📌 **Expected Outcome:**

✓ AWS Fargate **pulls the latest image and replaces running containers**.

---

3. Logging and Monitoring with AWS CloudWatch

✓ Enable **CloudWatch Logs** to track container logs:

```
logConfiguration:
  logDriver: awslogs
  options:
    awslogs-group: fargate-log-group
    awslogs-region: us-east-1
    awslogs-stream-prefix: ecs
```

✓ Monitor **CPU and memory usage** in **ECS Metrics**.

📌 **Expected Outcome:**

✓ Logs and metrics help **debug errors and monitor performance**.

CHAPTER 4: SECURITY BEST PRACTICES FOR AWS FARGATE

✓ **Use IAM Roles** – Assign least privilege permissions to Fargate tasks.

✓ **Implement VPC Security Groups** – Restrict access to Fargate tasks.

✓ **Enable AWS Secrets Manager** – Securely store credentials for containerized applications.

✓ **Monitor Network Traffic** – Use **AWS VPC Flow Logs** to analyze network activity.

📌 **Example:**

A **financial application** isolates Fargate containers in a **private VPC subnet** for security.

Chapter 5: AWS Fargate vs AWS ECS EC2

Feature	**AWS Fargate**	**AWS ECS EC2**
Infrastructure Management	Fully managed, no EC2 instances	Requires EC2 instance management
Scaling	Automatic scaling of tasks	Manual scaling or Auto Scaling Groups
Cost Model	Pay per vCPU and memory usage	Pay for EC2 instances (fixed cost)
Security	Runs in an isolated environment	Shared EC2 instances

📌 **Use Fargate** for **serverless, auto-scaling applications**.

📌 **Use ECS EC2** when **full control over infrastructure is needed**.

Chapter 6: Real-World Use Cases for AWS Fargate

1. Microservices Architecture

✓ Deploy multiple **independent services** with **Fargate + API Gateway**.

2. Machine Learning Model Deployment

✓ Run **ML inference workloads** on AWS Fargate using TensorFlow or PyTorch.

3. Event-Driven Processing

✓ Use AWS Lambda to trigger **Fargate tasks for background jobs**.

4. Web Application Hosting

✓ Deploy **serverless web applications** using ECS + Fargate.

CONCLUSION: MASTERING AWS FARGATE FOR SERVERLESS CONTAINERS

By using **AWS Fargate**, businesses can:

✅ **Deploy and manage containers without managing servers**.

✅ **Auto-scale applications dynamically**.

✅ **Integrate seamlessly with AWS services like ALB, IAM, CloudWatch, and DynamoDB**.

✅ **Improve security with VPC, IAM roles, and encryption**.

FINAL EXERCISE:

1. **Deploy a Node.js microservice on AWS Fargate with an API Gateway.**
2. **Configure an ALB to distribute traffic across multiple Fargate tasks.**
3. **Use AWS CloudWatch to monitor and debug Fargate container logs.**

Real-world Serverless Applications

Developing & Deploying Serverless APIs – Study Material

Introduction to Serverless APIs

What is a Serverless API?

A **serverless API** is an API that runs on a **serverless architecture**, where the backend logic is executed by **AWS Lambda, Azure Functions, or Google Cloud Functions**, without the need to manage or provision servers. Serverless APIs typically integrate with **API Gateway** to handle HTTP requests and route them to the appropriate backend functions.

Why Use Serverless APIs?

✔ **No Server Management** – AWS manages infrastructure, scaling, and availability.
✔ **Cost-Effective** – You pay only for the compute time used, eliminating idle costs.
✔ **Auto-Scaling** – APIs scale automatically based on incoming requests.
✔ **Security and Compliance** – Built-in **IAM authentication, API keys, and JWT token support**.
✔ **Seamless AWS Integration** – Works with **DynamoDB, S3, RDS, SNS, SQS, and Step Functions**.

📌 **Example Use Case:**
A **serverless REST API** that allows users to **register, log in, and fetch user profiles** using AWS Lambda, API Gateway, and DynamoDB.

Chapter 1: Key Components of a Serverless API

Component	Description
AWS Lambda	Executes backend logic in response to API requests.
API Gateway	Routes HTTP requests to Lambda functions.
DynamoDB	Stores structured data for the API (NoSQL database).
IAM Roles	Defines permissions for API Gateway and Lambda.
JWT Authentication	Secures API endpoints using Cognito or Autho.
AWS CloudWatch	Monitors API performance and logs errors.

Chapter 2: Setting Up a Serverless API Using AWS Lambda and API Gateway

Step 1: Create an AWS Lambda Function

1. Open **AWS Console** → Navigate to **AWS Lambda**.
2. Click **"Create Function"** → Choose **"Author from Scratch"**.
3. **Function Name:** UserAPIHandler.
4. **Runtime:** Python 3.9 (or Node.js, Go, Java, etc.).
5. **Execution Role:** Create a new role with **basic Lambda permissions**.
6. Click **"Create Function"**.

Step 2: Write the Lambda Function Code

Modify the function code to handle API requests:

```
import json


def lambda_handler(event, context):
   response = {
      "statusCode": 200,
      "body": json.dumps({"message": "User data retrieved successfully!", "user_id": 123})
   }
   return response
```

✔ Click **Deploy** to save changes.
✔ Test function execution using **"Test"**.

📌 **Expected Outcome:**
✔ Lambda executes and returns a **JSON response**.

Step 3: Create an API Gateway for Lambda

1. Open **AWS API Gateway Console**.
2. Click **"Create API"** → Choose **"REST API"**.
3. Select **"New API"** → Name it UserAPI.
4. Choose **Endpoint Type:** Regional.
5. Click **Create API**.

Step 4: Create a Resource & Method in API Gateway

1. Click **Actions** → Select **"Create Resource"**.
2. **Resource Name:** user.
3. Click **Create Resource**.
4. Click **"Create Method"** → Choose **"GET"**.
5. Select **Integration Type:** Lambda Function.
6. Enter **Lambda Function Name**: UserAPIHandler.
7. Click **Save** → Confirm permissions.

📌 **Expected Outcome:**

✔ API Gateway **routes GET requests to AWS Lambda**.

Chapter 3: Deploying and Testing the Serverless API

Step 1: Deploy API Gateway

1. Click **Actions** → Select **"Deploy API"**.
2. **Stage Name:** dev.
3. Click **Deploy**.

Step 2: Get API Endpoint URL

1. After deployment, **copy the API Invoke URL**:
2. https://abcd1234.execute-api.us-east-1.amazonaws.com/dev/user
3. Open a **web browser** or use **Postman** to send a GET request:

4. curl -X GET https://abcd1234.execute-api.us-east-1.amazonaws.com/dev/user

📌 **Expected Outcome:**

✓ API Gateway invokes **Lambda and returns a JSON response**.

Chapter 4: Enhancing API Security and Performance

1. Enable Authentication Using AWS Cognito

1. Open **AWS Cognito Console** → Create a new **User Pool**.
2. Enable **JWT Authentication** and configure **OAuth2 flows**.
3. Attach Cognito Authorizer to API Gateway.

📌 **Expected Outcome:**

✓ API requests require **user authentication via Cognito tokens**.

---

2. Enable API Keys for Secure Access

1. Open **API Gateway Console** → Select UserAPI.
2. Click **"Create API Key"** → Generate a key.
3. Attach the key to **Usage Plans** and require it in API Gateway.

📌 **Expected Outcome:**

✓ API requests must include a **valid API key**.

---

3. Enable CORS for Cross-Origin Requests

✓ Modify API Gateway settings to allow CORS:

{

```
 "statusCode": 200,
 "headers": { "Access-Control-Allow-Origin": "*" },
 "body": "Response Data"
}
```

📌 **Expected Outcome:**

✔ Web applications can now **access the API from different domains**.

Chapter 5: Logging & Monitoring Serverless APIs

1. Enable AWS CloudWatch Logs for API Gateway

1. Open **API Gateway Console** → Select UserAPI.
2. Click **Settings** → Enable **CloudWatch Logs**.
3. View API logs in **CloudWatch Console**.

---

2. Enable AWS CloudWatch Logs for Lambda

1. Open **AWS Lambda Console** → Select UserAPIHandler.
2. Click **Monitor** → View execution logs.

📌 **Expected Outcome:**

✔ Logs **track API requests, Lambda execution, and errors**.

Chapter 6: Scaling and Performance Optimization

✔ **Enable API Gateway Caching** – Reduce Lambda invocations by caching API responses.

✔ **Use AWS Lambda Provisioned Concurrency** – Reduce cold starts for frequently accessed APIs.

✓ **Optimize Lambda Memory Allocation** – Adjust CPU and memory settings based on performance metrics.

✓ **Use AWS CloudFront** – Distribute API requests across global edge locations.

📌 **Example:**

A **finance company** caches API responses to reduce latency and improve performance.

Chapter 7: Comparing Serverless APIs with Traditional APIs

Feature	Serverless APIs (AWS Lambda + API Gateway)	Traditional APIs (EC2/Containers)
Infrastructure Management	Fully managed, no servers	Requires provisioning & scaling
Auto-Scaling	Scales automatically	Requires manual scaling
Cost Efficiency	Pay only for execution time	Pay for always-on servers
Security	IAM-based permissions, API keys, JWT	Requires custom security setup

📌 **Use Serverless APIs** for **event-driven, cost-efficient applications**.

Chapter 8: Real-World Use Cases for Serverless APIs

1. Mobile App Backend

✓ AWS Lambda + API Gateway handle **user authentication, payments, and real-time notifications**.

2. IoT Data Processing

✓ AWS IoT Core triggers **serverless APIs for sensor data storage and analysis**.

3. Chatbots & AI Assistants

✓ AWS Lambda processes **user queries and integrates with AI services**.

4. Data Processing Pipelines

✓ API triggers **Lambda functions for ETL tasks** in **S3 and DynamoDB**.

CONCLUSION: MASTERING SERVERLESS API DEVELOPMENT

By using **AWS Lambda and API Gateway**, businesses can:

✅ **Develop scalable, serverless applications without managing infrastructure**.

✅ **Secure APIs with IAM, Cognito, and API keys**.

✅ **Monitor and optimize API performance using CloudWatch and caching**.

✅ **Reduce costs by paying only for execution time**.

FINAL EXERCISE:

1. **Create a RESTful API to store and retrieve user profiles from DynamoDB.**
2. **Implement authentication using AWS Cognito.**

3. **Use AWS CloudWatch to monitor API performance.**

MONITORING & DEBUGGING SERVERLESS APPS – STUDY MATERIAL

INTRODUCTION TO MONITORING & DEBUGGING SERVERLESS APPLICATIONS

Why is Monitoring and Debugging Important for Serverless Apps?

In a serverless architecture, applications run across multiple **AWS services** (such as Lambda, API Gateway, DynamoDB, S3, and Step Functions), making it crucial to **monitor performance, detect failures, and debug errors** efficiently.

Challenges in Monitoring Serverless Apps

✔ **Cold Starts** – Lambda functions may have higher latency when idle for long periods.
✔ **Distributed Architecture** – Multiple AWS services interact asynchronously.
✔ **Limited Debugging Tools** – Traditional debugging methods (e.g., SSH access) are unavailable.
✔ **Scalability Issues** – Unexpected traffic spikes may cause performance bottlenecks.

📌 **Example Use Case:**
A **serverless e-commerce API** using **AWS Lambda, API Gateway, DynamoDB, and S3** must track latency, error rates, and request tracing to ensure smooth operation.

CHAPTER 1: KEY MONITORING & DEBUGGING TOOLS FOR SERVERLESS APPLICATIONS

AWS Service/Tool	Description

AWS CloudWatch	Monitors logs, metrics, and dashboards for serverless apps.
AWS X-Ray	Provides distributed tracing for Lambda and API Gateway requests.
AWS Lambda Insights	Monitors Lambda performance, cold starts, and memory usage.
Amazon CloudTrail	Tracks API activity and security-related events.
AWS Config	Ensures compliance by tracking resource configurations and changes.
AWS Step Functions Logs	Debugs workflow execution failures in serverless orchestration.

CHAPTER 2: SETTING UP MONITORING WITH AWS CLOUDWATCH

Step 1: Enable CloudWatch Logs for AWS Lambda

1. Open **AWS Lambda Console** → Select MyLambdaFunction.
2. Click **"Configuration"** → Select **"Monitoring and Operations Tools"**.
3. Enable **"Send logs to Amazon CloudWatch"**.
4. Click **Save Changes**.

📌 **Expected Outcome:**

✔ Lambda execution logs will be available in **CloudWatch Logs**.

Step 2: View Lambda Logs in CloudWatch

Run the following AWS CLI command to check Lambda logs:

aws logs tail /aws/lambda/MyLambdaFunction --follow

Alternatively, navigate to **AWS CloudWatch Console** → **Log Groups** → **/aws/lambda/MyLambdaFunction**.

📌 **Expected Outcome:**
✔ Displays **Lambda execution details, errors, and duration metrics**.

Step 3: Create a CloudWatch Alarm for Lambda Errors

1. Open **AWS CloudWatch Console** → Click **"Alarms"**.
2. Click **"Create Alarm"** → Select **Lambda Function Metrics**.
3. Choose **"Errors"** → Set threshold (e.g., > 5 errors in 5 minutes).
4. Set **Action** to send an **SNS notification**.
5. Click **Create Alarm**.

📌 **Expected Outcome:**
✔ Receives alerts when **Lambda errors exceed the threshold**.

CHAPTER 3: TRACING SERVERLESS APPLICATIONS WITH AWS X-RAY

AWS **X-Ray** helps trace API calls, Lambda executions, and DynamoDB interactions for better **performance debugging**.

Step 1: Enable X-Ray for AWS Lambda

1. Open **AWS Lambda Console** → Select MyLambdaFunction.
2. Click **"Configuration"** → Select **"Monitoring and Operations Tools"**.
3. Enable **"Active Tracing"**.

4. Click **Save Changes**.

Step 2: View X-Ray Traces in Console

1. Open **AWS X-Ray Console** → Click **"Traces"**.
2. Filter requests with **"Lambda"**, **"API Gateway"**, or **"DynamoDB"**.
3. Analyze **latency, failed requests, and request flow**.

📌 **Expected Outcome:**
✔ Helps **debug slow API responses and identify performance bottlenecks**.

Chapter 4: Using AWS Lambda Insights for Performance Monitoring

Step 1: Enable AWS Lambda Insights

1. Open **AWS Lambda Console** → Select MyLambdaFunction.
2. Click **"Configuration"** → Select **"Monitoring and Operations Tools"**.
3. Enable **"Enhanced Monitoring"**.

Step 2: View Performance Metrics

1. Open **AWS CloudWatch Console** → Click **"Metrics"**.
2. Navigate to **Lambda Insights** → Select metrics like:
    - **Duration** (execution time).
    - **Cold Start Count** (number of slow starts).

- **Memory Utilization**.

📌 **Expected Outcome:**
✔ Helps **optimize memory allocation and reduce cold start times**.

CHAPTER 5: DEBUGGING SERVERLESS APPLICATIONS

1. Debugging AWS Lambda with Custom Logs

Modify Lambda function to log execution details:

```
import json

import logging


logger = logging.getLogger()

logger.setLevel(logging.INFO)


def lambda_handler(event, context):

   logger.info(f"Event received: {json.dumps(event)}")


   try:

      user_id = event.get("user_id", "Unknown")

      response = {"message": "User data retrieved", "user_id": user_id}

      logger.info(f"Response: {json.dumps(response)}")

      return {"statusCode": 200, "body": json.dumps(response)}

   except Exception as e:

      logger.error(f"Error occurred: {str(e)}")
```

return {"statusCode": 500, "body": "Internal Server Error"}

📌 **Expected Outcome:**

✓ **Logs provide detailed execution flow for debugging**.

---

2. Debugging API Gateway Errors

Use API Gateway **Execution Logs** to analyze errors:

1. Open **API Gateway Console** → Select MyAPI.
2. Click **"Stages"** → Select dev.
3. Enable **"CloudWatch Logs"**.
4. Check **CloudWatch Log Streams** for API execution details.

📌 **Expected Outcome:**

✓ Detects **authentication failures, request timeouts, and missing parameters**.

CHAPTER 6: AUTOMATED ALERTS & ERROR HANDLING

1. Use AWS SNS for Failure Notifications

Create an **SNS Topic** to notify developers about failures:

1. Open **SNS Console** → Click **"Create Topic"**.
2. Name it ServerlessAlerts.
3. Subscribe team emails for notifications.
4. Attach SNS topic to **CloudWatch Alarms** for errors.

📌 **Expected Outcome:**

✓ Receives **real-time alerts** when Lambda or API Gateway encounters errors.

---

2. Implement Dead Letter Queues (DLQ) for AWS Lambda

If Lambda fails, send unprocessed messages to an **SQS Dead Letter Queue (DLQ)**.

1. Open **AWS Lambda Console** → Select MyLambdaFunction.
2. Click **"Configuration"** → Select **"Destinations"**.
3. Set **"On Failure"** destination as an **SQS Queue**.
4. Click **Save**.

📌 **Expected Outcome:**
✓ Unprocessed Lambda events **are stored in SQS for later debugging**.

CHAPTER 7: BEST PRACTICES FOR MONITORING & DEBUGGING SERVERLESS APPS

✓ **Use Structured Logging** – Log important events in JSON format for easy parsing.
✓ **Implement Timeouts & Retries** – Set execution time limits and retry failed requests.
✓ **Reduce Cold Starts** – Use **Provisioned Concurrency** for Lambda functions.
✓ **Enable CloudWatch Alarms** – Set up alerts for API Gateway, DynamoDB, and Lambda failures.
✓ **Use AWS X-Ray** – Monitor distributed traces across multiple services.

📌 **Example:**
A **fintech app** uses **AWS X-Ray to trace latency issues in API requests**.

Conclusion: Mastering Serverless Monitoring & Debugging

By using AWS **CloudWatch, X-Ray, and Lambda Insights**, businesses can:

✅ **Monitor serverless app performance in real-time**.

✅ **Quickly debug errors in AWS Lambda, API Gateway, and DynamoDB**.

✅ **Set up automated alerts for faster issue resolution**.

✅ **Optimize serverless functions for better scalability and cost efficiency**.

Final Exercise:

1. **Enable CloudWatch logs for your Lambda function and analyze execution details.**
2. **Use AWS X-Ray to trace API Gateway requests to Lambda.**
3. **Set up an SNS topic for error notifications when Lambda fails.**

ASSIGNMENT

DEVELOP A MICROSERVICES-BASED ARCHITECTURE USING ECS

SOLUTION: DEVELOP A MICROSERVICES-BASED ARCHITECTURE USING AWS ECS (STEP-BY-STEP GUIDE)

This guide will walk you through **developing a microservices-based architecture using AWS ECS (Elastic Container Service)**. The architecture will include **multiple microservices**, managed using **AWS ECS with Fargate**, and connected via an **Application Load Balancer (ALB)**.

Step 1: Understanding Microservices on AWS ECS

Why Use AWS ECS for Microservices?

✓ **Containerized Deployment** – Runs each microservice in a separate container.
✓ **Auto-Scaling** – Scales microservices dynamically based on demand.
✓ **AWS Fargate Support** – Allows serverless container execution.
✓ **Service Discovery** – Uses Amazon ECS Service Discovery to manage microservices.
✓ **Integrated Load Balancing** – Distributes traffic using **AWS ALB**.

📌 **Use Case Example:**
A **retail e-commerce platform** with the following microservices:

1. **User Service** (Handles user authentication and profiles).
2. **Product Service** (Manages product catalog).
3. **Order Service** (Handles order processing and payments).

Step 2: Set Up ECS Cluster for Microservices

1. Create an ECS Cluster

1. Open **AWS ECS Console** → Click **"Clusters"**.
2. Click **"Create Cluster"**.
3. Choose **"Networking only (AWS Fargate)"**.
4. **Cluster Name:** ECS-Microservices-Cluster.
5. Click **"Create"**.

📌 **Expected Outcome:**
✓ A **new ECS cluster** is created for running microservices.

Step 3: Deploy User Microservice on AWS ECS

1. Create a Docker Image for User Service

Create a new folder user-service and create app.py:

```
from flask import Flask, jsonify


app = Flask(__name__)


@app.route("/user", methods=["GET"])
def get_user():
    return jsonify({"user_id": 1, "name": "John Doe"})


if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000)
```

---

2. Create a Dockerfile for the Service

Create a Dockerfile inside user-service folder:

```
FROM python:3.9

WORKDIR /app

COPY requirements.txt .

RUN pip install -r requirements.txt

COPY . .

CMD ["python", "app.py"]
```

---

3. Build & Push Docker Image to Amazon Elastic Container Registry (ECR)

1. Create an **ECR Repository**:

```
aws ecr create-repository --repository-name user-service
```

2. Authenticate Docker with AWS ECR:

```
aws ecr get-login-password | docker login --username AWS --password-stdin <AWS_ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com
```

3. Build and push the Docker image:

```
docker build -t user-service .

docker tag user-service:latest <AWS_ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/user-service:latest

docker push <AWS_ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/user-service:latest
```

📌 **Expected Outcome:**
✔ **User microservice image is stored in ECR**.

Step 4: Define ECS Task Definition for User Microservice

1. Open **AWS ECS Console** → Click **"Task Definitions"**.
2. Click **"Create new Task Definition"**.
3. Select **Fargate** as the launch type.
4. **Task Definition Name:** UserServiceTask.
5. **Execution Role:** Select **ecsTaskExecutionRole**.
6. Add **Container Definition**:
    - **Container Name:** user-service-container
    - **Image:** <AWS_ACCOUNT_ID>.dkr.ecr.<REGION>.amazonaws.com/user-service:latest
    - **Memory:** 512 MB
    - **CPU:** 0.25 vCPU
    - **Port Mapping:** 5000:5000

📌 **Expected Outcome:**
✔ The **ECS Task Definition for User Microservice** is created.

Step 5: Deploy User Microservice as an ECS Service

1. Open **AWS ECS Console** → Navigate to **Clusters** → **ECS-Microservices-Cluster**.
2. Click **"Create Service"**.

3. Select **Launch Type:** Fargate.
4. Choose **Task Definition:** UserServiceTask.
5. Enter **Service Name:** UserService.
6. Set **Desired Task Count:** 2 (for high availability).
7. **Configure Load Balancer:**
   - Choose **Application Load Balancer (ALB)**.
   - Create a **Target Group** named UserServiceTG.
   - Register 5000 as the listener port.
8. Click **"Create Service"**.

📌 **Expected Outcome:**

✔ **User Service is running on AWS ECS with ALB integration**.

Step 6: Deploy Additional Microservices (Product & Order Services)

Repeat **Steps 3 to 5** for product-service and order-service.

- **Product Service API:** /product (port **5001**)
- **Order Service API:** /order (port **5002**)

📌 **Expected Outcome:**

✔ ECS **manages multiple microservices**, each with **separate container images, task definitions, and services**.

Step 7: Configure Service Discovery for Microservices Communication

1. Open **AWS ECS Console** → Select **ECS-Microservices-Cluster**.
2. Click **"Service Discovery"** → Create **Namespace** (microservices.local).
3. Attach each service (UserService, ProductService, OrderService).
4. Each microservice can now **access other services using DNS names**:
   - user-service.microservices.local
   - product-service.microservices.local
   - order-service.microservices.local

📌 **Expected Outcome:**

✔ Microservices can **discover and communicate** without hardcoding IP addresses.

Step 8: Monitor Microservices Performance

1. Enable CloudWatch Logs for ECS Tasks

Add logging to serverless.yml:

```
logConfiguration:
 logDriver: awslogs
 options:
  awslogs-group: ecs-logs
  awslogs-region: us-east-1
  awslogs-stream-prefix: ecs
```

---

2. Enable Auto Scaling for Microservices

1. Open **AWS ECS Console** → Select **ECS-Microservices-Cluster**.
2. Click **"UserService"** → Select **Auto Scaling**.
3. Set **Scaling Policy:**
    - **Scale up when CPU > 70%**.
    - **Minimum tasks:** 2
    - **Maximum tasks:** 10

📌 **Expected Outcome:**
✓ ECS **automatically scales microservices** based on demand.

Step 9: Test Microservices API Endpoints

1. Retrieve the **Application Load Balancer URL** from **AWS ALB Console**.
2. Test each microservice endpoint using **Postman or cURL**:

curl -X GET http://ALB-URL/user

curl -X GET http://ALB-URL/product

curl -X GET http://ALB-URL/order

📌 **Expected Outcome:**
✓ Each microservice **responds with JSON data**.

Conclusion: Mastering Microservices Deployment on AWS ECS

By using **AWS ECS with Fargate**, businesses can:

✅ **Deploy scalable microservices without managing EC2 instances**.

✅ **Enable secure service discovery and inter-service communication**.

✅ **Auto-scale services based on CPU usage and demand**.

✅ **Use ALB to distribute traffic across multiple microservices**.

Final Exercise:

1. **Deploy a new "Payment Service" microservice and integrate it with Order Service.**
2. **Use AWS CloudWatch to monitor API response times and set up alerts.**
3. **Implement a CI/CD pipeline to automate ECS deployments using AWS CodePipeline.**

Deploy a Serverless Application using AWS Lambda

Solution: Deploy a Serverless Application Using AWS Lambda (Step-by-Step Guide)

This guide walks you through **deploying a serverless application using AWS Lambda**, integrated with **Amazon API Gateway and DynamoDB** for a fully functional **RESTful API**.

Step 1: Understanding AWS Lambda for Serverless Applications

Why Use AWS Lambda for Serverless Apps?

✔ **No Server Management** – AWS manages the infrastructure, scaling, and availability.
✔ **Auto-Scaling** – Functions scale automatically with demand.
✔ **Cost-Efficient** – Pay only for execution time.
✔ **Easy Integration** – Works with **API Gateway, DynamoDB, S3, SNS, and other AWS services.**
✔ **Security & IAM Control** – Uses **IAM roles** to control permissions.

📌 **Use Case Example:**
A **user management system** with the following functionalities:

1. **Create User** (POST /users)
2. **Get User** (GET /users/{id})

Step 2: Set Up AWS Lambda for the Serverless Application

1. Create an AWS Lambda Function

1. Open **AWS Lambda Console** → Click **"Create Function"**.
2. Choose **"Author from Scratch"**.

3. **Function Name:** UserManagementFunction.
4. **Runtime:** Python 3.9.
5. **Execution Role:** Select **"Create a new role with basic Lambda permissions"**.
6. Click **"Create Function"**.

---

2. Write Lambda Function Code

Modify the Lambda function to **handle user creation and retrieval**:

```
import json

import boto3

import uuid


# Initialize DynamoDB client

dynamodb = boto3.resource("dynamodb")

table = dynamodb.Table("UsersTable")


def lambda_handler(event, context):

   http_method = event["httpMethod"]


   if http_method == "POST":

      return create_user(event)

   elif http_method == "GET":

      return get_user(event)
```

```
    else:
        return {"statusCode": 400, "body": json.dumps({"error": "Unsupported method"})}


def create_user(event):
    body = json.loads(event["body"])
    user_id = str(uuid.uuid4())
    table.put_item(Item={"user_id": user_id, "name": body["name"], "email": body["email"]})

    return {"statusCode": 201, "body": json.dumps({"message": "User created", "user_id": user_id})}


def get_user(event):
    user_id = event["pathParameters"]["id"]
    response = table.get_item(Key={"user_id": user_id})

    if "Item" in response:
        return {"statusCode": 200, "body": json.dumps(response["Item"])}
    else:
        return {"statusCode": 404, "body": json.dumps({"error": "User not found"})}
```

✓ Click **Deploy** to save changes.

📌 **Expected Outcome:**

✓ The Lambda function **handles user creation and retrieval**.

Step 3: Create a DynamoDB Table for Storing Users

1. Open **AWS DynamoDB Console** → Click **"Create Table"**.
2. **Table Name:** UsersTable.
3. **Partition Key:** user_id (String).
4. Click **"Create Table"**.

📌 **Expected Outcome:**

✓ A DynamoDB table is created to **store user details**.

Step 4: Grant Lambda Permission to Access DynamoDB

1. Open **AWS IAM Console** → Select **Lambda Execution Role**.
2. Click **"Add Permissions"** → Attach Policy.
3. Select **"AmazonDynamoDBFullAccess"** → Click **Attach Policy**.

📌 **Expected Outcome:**

✓ Lambda function **gains permission to read/write DynamoDB data**.

Step 5: Set Up API Gateway to Trigger Lambda

1. Create a New API in API Gateway

1. Open **AWS API Gateway Console** → Click **"Create API"**.
2. Select **"REST API"** → Click **Build**.

3. **API Name:** UserAPI.
4. Click **"Create API"**.

---

2. Create Resources and Methods

Create User Endpoint (POST /users)

1. Click **Actions** → **Create Resource**.
2. **Resource Name:** users.
3. Click **Create Resource**.
4. Click **Actions** → **Create Method** → Select **POST**.
5. **Integration Type:** Lambda Function.
6. **Lambda Function Name:** UserManagementFunction.
7. Click **Save** → Confirm permissions.

Get User Endpoint (GET /users/{id})

1. Click **Actions** → **Create Resource**.
2. **Resource Name:** {id} (as a path parameter).
3. Click **Create Resource**.
4. Click **Actions** → **Create Method** → Select **GET**.
5. **Integration Type:** Lambda Function.
6. **Lambda Function Name:** UserManagementFunction.
7. Click **Save** → Confirm permissions.

📌 **Expected Outcome:**

✓ API Gateway **routes HTTP requests to AWS Lambda**.

Step 6: Deploy the API and Test the Application

1. Deploy API Gateway

1. Click **Actions** → **Deploy API**.
2. **Stage Name:** dev.
3. Click **Deploy**.

---

2. Get API Endpoint URL

1. Copy the **Invoke URL**:
2. https://abcd1234.execute-api.us-east-1.amazonaws.com/dev
3. Test the API using **cURL or Postman**:

Create User (POST Request)

curl -X POST https://abcd1234.execute-api.us-east-1.amazonaws.com/dev/users \

-H "Content-Type: application/json" \

-d '{"name": "Alice", "email": "alice@example.com"}'

Get User by ID (GET Request)

curl -X GET https://abcd1234.execute-api.us-east-1.amazonaws.com/dev/users/{user_id}

📌 **Expected Outcome:**

✓ User is **created and retrieved successfully**.

Step 7: Enable Logging and Monitoring

1. Enable CloudWatch Logs for Lambda

1. Open **AWS Lambda Console** → Select UserManagementFunction.
2. Click **"Configuration"** → Select **"Monitoring and Operations Tools"**.
3. Enable **"Send logs to CloudWatch"**.
4. Click **Save**.

2. Enable API Gateway Logs

1. Open **API Gateway Console** → Select UserAPI.
2. Click **Stages** → Select dev.
3. Enable **CloudWatch Logs** for API execution monitoring.

📌 **Expected Outcome:**

✓ Logs **track API requests, Lambda execution, and errors**.

Step 8: Optimize and Secure the Serverless API

✓ Enable API Keys for Secure Access

- Open **API Gateway Console** → Create API Key.
- Attach the key to API **Usage Plans**.
- Require API Key in **Method Request Settings**.

✓ Use IAM Roles for Lambda

- Limit Lambda **IAM Role permissions** to only required services.

✓ Enable CORS

- Modify API Gateway settings to allow **Cross-Origin Requests**.

📌 **Expected Outcome:**

✔ The API is **secure and optimized for production**.

Conclusion: Mastering Serverless Application Deployment on AWS Lambda

By using **AWS Lambda, API Gateway, and DynamoDB**, businesses can:

✅ **Deploy scalable serverless applications without managing infrastructure.**

✅ **Secure APIs with IAM roles and API keys.**

✅ **Monitor and debug applications with CloudWatch Logs.**

✅ **Optimize performance with auto-scaling and caching.**

Final Exercise:

1. **Extend the API with an "Update User" (PUT) and "Delete User" (DELETE) endpoint.**
2. **Integrate AWS Cognito for user authentication.**
3. **Deploy a CI/CD pipeline for Lambda using AWS CodePipeline.**