ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

ADVANCED AWS SERVICES

AWS STEP FUNCTIONS FOR WORKFLOW AUTOMATION – STUDY MATERIAL

INTRODUCTION TO AWS STEP FUNCTIONS

What are AWS Step Functions?

AWS **Step Functions** is a **serverless orchestration service** that helps automate workflows by coordinating multiple AWS services. It allows users to define complex processes as **state machines** and execute them in a **visual workflow**.

Why Use AWS Step Functions?

✔ **Visual Workflow Automation** – Provides a graphical interface to design workflows.
✔ **Serverless & Scalable** – No infrastructure management required.
✔ **Error Handling & Retry Mechanisms** – Automatically retries failed steps.
✔ **Integration with AWS Services** – Works with **AWS Lambda, ECS, DynamoDB, S3, API Gateway, SNS, and more**.
✔ **Stateful Execution** – Keeps track of workflow execution status.

📌 **Example Use Case:**

A **document processing system** that uploads a file to S3 → extracts metadata using Lambda → stores metadata in DynamoDB.

CHAPTER 1: UNDERSTANDING AWS STEP FUNCTIONS

1. Key Components of AWS Step Functions

Component	Description
State Machine	Defines the workflow steps and execution order.
State	A step in the workflow (e.g., invoking a Lambda function, waiting, or branching logic).
Task State	Executes a task, such as an AWS Lambda function or an API call.
Choice State	Implements conditional branching (e.g., if condition A is met, go to step X).
Wait State	Introduces a delay before proceeding to the next step.
Parallel State	Runs multiple steps concurrently.
Pass State	Passes input data to the next step without modification.
Fail & Succeed State	Ends the workflow execution with a failure or success.
Execution	A running instance of a state machine.

---

2. Types of AWS Step Functions

Step Function Type	**Use Case**	**Execution Duration**
Standard Workflow	Long-running workflows like batch processing, human approvals, data pipelines	Up to **1 year**
Express Workflow	High-speed event-driven workloads like IoT, API requests, microservices	Up to **5 minutes**

CHAPTER 2: CREATING A BASIC AWS STEP FUNCTION WORKFLOW

Step 1: Open AWS Step Functions Console

1. Log in to **AWS Console**.
2. Navigate to **AWS Step Functions**.
3. Click **"Create State Machine"**.

Step 2: Define a Simple Workflow Using JSON

AWS Step Functions use **Amazon States Language (ASL)**, a JSON-based syntax.

Simple Example: A Step Function That Invokes a Lambda Function

```
{
 "Comment": "Basic AWS Step Function Example",
 "StartAt": "InvokeLambda",
 "States": {
  "InvokeLambda": {
    "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-
1:123456789012:function:MyLambdaFunction",
    "End": true
   }
  }
}
```

✔ **"StartAt"** defines the first state (InvokeLambda).
✔ **"Type": "Task"** executes an AWS Lambda function.
✔ **"End": true** indicates that this is the last step.

📌 **Expected Outcome:**
This Step Function will **trigger a Lambda function**.

Step 3: Deploy the State Machine in AWS Console

1. Open **AWS Step Functions Console**.
2. Click **"Create State Machine"**.
3. Choose **"Author with Code"**, then paste the JSON definition.
4. Choose **Standard** or **Express** workflow.
5. Click **"Next"** → Assign a **name** (MyStepFunction).
6. Choose an **IAM role** to allow Step Functions to invoke Lambda.
7. Click **"Create State Machine"**.

📌 **Expected Outcome:**
The **Step Function is created and ready for execution**.

Chapter 3: Running and Monitoring AWS Step Functions

Step 1: Start a New Execution

1. Open **AWS Step Functions Console**.
2. Select MyStepFunction.
3. Click **"Start Execution"**.
4. Enter input (if required) and click **"Start"**.

📌 **Expected Outcome:**
The workflow **executes each step sequentially**.

Step 2: Monitor Execution History

1. Go to **Execution History**.
2. Click on an execution to view **step-by-step status**.
3. Debug any errors or failures using CloudWatch logs.

📌 **Expected Outcome:**
Each step logs **input, output, and execution status**.

Chapter 4: Advanced Step Functions Features

1. Implementing Conditional Logic (Choice State)

Allows workflows to make **decisions** based on input.

```
{
 "StartAt": "CheckOrderStatus",
 "States": {
  "CheckOrderStatus": {
   "Type": "Choice",
   "Choices": [
    {
     "Variable": "$.orderStatus",
     "StringEquals": "Approved",
     "Next": "ProcessOrder"
    },
    {
     "Variable": "$.orderStatus",
     "StringEquals": "Rejected",
     "Next": "CancelOrder"
    }
   ],
   "Default": "EscalateIssue"
  },
  "ProcessOrder": {
   "Type": "Task",
   "Resource": "arn:aws:lambda:us-east-
1:123456789012:function:ProcessOrderFunction",
   "End": true
  },
  "CancelOrder": {
   "Type": "Task",
    "Resource": "arn:aws:lambda:us-east-1:123456789012:function:CancelOrderFunction",
    "End": true
   }
  }
}
```

📌 **Expected Outcome:**

✔ If orderStatus = Approved, process the order.

✔ If orderStatus = Rejected, cancel the order.

✔ Otherwise, escalate the issue.

---

2. Running Parallel States (Concurrent Execution)

```
{
 "StartAt": "ParallelProcessing",
 "States": {
  "ParallelProcessing": {
   "Type": "Parallel",
   "Branches": [
    {
     "StartAt": "ProcessA",
     "States": {
      "ProcessA": {
       "Type": "Task",
       "Resource": "arn:aws:lambda:us-east-1:123456789012:function:ProcessAFunction",
       "End": true
      }
     }
    },
    {
     "StartAt": "ProcessB",
     "States": {
      "ProcessB": {
       "Type": "Task",
       "Resource": "arn:aws:lambda:us-east-1:123456789012:function:ProcessBFunction",
       "End": true
      }
     }
    }
   ],
   "End": true
  }
 }
}
```

📌 **Expected Outcome:**

✓ Runs ProcessAFunction and ProcessBFunction **simultaneously**.

CHAPTER 5: BEST PRACTICES FOR AWS STEP FUNCTIONS

✓ **Use Standard Workflows** for long-running processes and **Express Workflows** for short-duration tasks.

✓ **Enable Logging in CloudWatch** for monitoring and debugging.

✓ **Implement Error Handling and Retry Mechanisms** to avoid failures.

✓ **Use Step Functions for Microservices Orchestration** instead of direct service-to-service calls.

📌 **Example:**

A **fraud detection system** uses **AWS Step Functions** to **analyze transaction patterns in real-time**.

CHAPTER 6: REAL-WORLD USE CASES FOR AWS STEP FUNCTIONS

1. Order Processing System

✓ **Trigger an order workflow** → Validate payment → Dispatch order → Notify customer.

2. Data Processing Pipelines

✓ Fetch data from **S3**, process using **Lambda**, store results in **DynamoDB**.

3. Serverless ETL Workflows

✓ Extract data → Transform using **Glue or Lambda** → Load into **Redshift or S3**.

4. Chatbot Workflow Automation

✓ Take user input → Process using **Lex** → Store responses in **DynamoDB**.

CONCLUSION: MASTERING AWS STEP FUNCTIONS

By using **AWS Step Functions**, businesses can:

✅ **Automate workflows using serverless architecture**.
✅ **Reduce manual intervention in cloud processes**.
✅ **Enhance monitoring and execution tracking**.
✅ **Integrate multiple AWS services efficiently**.

FINAL EXERCISE:

1. **Create a Step Function that triggers Lambda to process customer data**.
2. **Use AWS Step Functions to automate EC2 instance scaling**.
3. **Implement an Express Workflow for API request handling**.

AWS SNS & SQS for Messaging – Study Material

Introduction to AWS SNS & SQS

What is AWS SNS (Simple Notification Service)?

AWS **Simple Notification Service (SNS)** is a fully managed **pub/sub messaging service** that allows applications to send **notifications or messages** to multiple subscribers (e.g., emails, SMS, Lambda, SQS, HTTP endpoints).

What is AWS SQS (Simple Queue Service)?

AWS **Simple Queue Service (SQS)** is a **fully managed message queuing service** that enables **asynchronous processing** by storing messages in a queue and allowing decoupled components to process them at different times.

Why Use AWS SNS & SQS?

✓ **Decoupling Microservices** – Ensures that services communicate without tight dependencies.
✓ **Scalability & Reliability** – Supports high-throughput message handling.
✓ **Fault Tolerance** – SQS stores messages for a defined retention period, ensuring they are processed.
✓ **Push & Pull Messaging** – SNS sends push notifications, while SQS allows for delayed, reliable message processing.
✓ **Integration with AWS Services** – Works with **Lambda, EC2, ECS, CloudWatch, API Gateway**, etc.

📌 **Example Use Case:**
An **e-commerce platform** uses SNS to notify users about order status and SQS to queue order processing tasks.

CHAPTER 1: AWS SNS (SIMPLE NOTIFICATION SERVICE)

1. Key Features of AWS SNS

Feature	**Description**
Topic-Based Pub/Sub	Messages are published to a topic and delivered to subscribers.
Multiple Protocols	Supports **Email, SMS, HTTP, Lambda, SQS** as subscribers.
Message Filtering	Allows subscribers to receive only relevant messages.
FIFO SNS (Preview)	Supports ordered message delivery (in development).
Security & Encryption	Supports AWS IAM policies and KMS encryption.

---

2. AWS SNS Architecture

✓ **Publisher (Producer)** – Publishes messages to an SNS topic.
✓ **SNS Topic** – Acts as a communication channel for message distribution.
✓ **Subscribers (Consumers)** – Receive messages via **Email, SMS, Lambda, SQS, HTTP/S endpoints**.

📌 **Example:**
A **monitoring system** publishes an alert message to an SNS topic → SNS forwards the message to **SMS, email, and Lambda subscribers**.

Chapter 2: Setting Up AWS SNS

Step 1: Create an SNS Topic

1. Open **AWS SNS Console** → Click **"Create Topic"**.
2. **Topic Name:** OrderUpdates.
3. **Type: Standard** (or FIFO for strict ordering).
4. **Access Policy:** Choose Allow all AWS accounts or specify IAM users.
5. Click **"Create Topic"**.

Step 2: Add Subscribers to SNS Topic

1. Open the **OrderUpdates topic**.
2. Click **"Create Subscription"**.
3. **Protocol:** Choose **Email, SMS, Lambda, HTTP, or SQS**.
4. **Endpoint:** Enter the recipient's email, phone number, or Lambda ARN.
5. Click **"Create Subscription"**.

📌 **Expected Outcome:**
The **subscriber will receive a confirmation request**. Email/SMS subscribers must **confirm the subscription**.

Step 3: Publish a Message to SNS Topic

1. Open **SNS Console** → Select **OrderUpdates**.
2. Click **"Publish Message"**.
3. Enter **Message Subject** and **Message Body**.

4. Click **"Publish Message"**.

📌 **Expected Outcome:**
All **subscribers receive the published message**.

CHAPTER 3: AWS SQS (SIMPLE QUEUE SERVICE)

1. Key Features of AWS SQS

Feature	**Description**
Standard Queues	Best-effort ordering, high throughput.
FIFO Queues	Guaranteed ordering, exactly-once processing.
Message Retention	Stores messages for up to **14 days**.
Dead-Letter Queues (DLQ)	Handles failed message processing.
Visibility Timeout	Prevents duplicate message processing.

📌 **Example:**
A **payment processing system** uses an **SQS queue** to handle asynchronous transactions.

CHAPTER 4: SETTING UP AWS SQS

Step 1: Create an SQS Queue

1. Open **AWS SQS Console** → Click **"Create Queue"**.
2. **Queue Name:** OrderProcessingQueue.
3. **Queue Type:** Choose **Standard** or **FIFO**.
4. Configure:

   - **Message Retention:** 4 days.
   - **Visibility Timeout:** 30 seconds.
   - **Delivery Delay:** 0 seconds.
5. Click **"Create Queue"**.

Step 2: Send a Message to SQS Queue

1. Select OrderProcessingQueue.
2. Click **"Send and Receive Messages"**.
3. Enter **a sample message** ({"order_id": "1234", "status": "pending"}) and click **"Send Message"**.

📌 **Expected Outcome:**
Message is **stored in the queue** and awaits processing.

Step 3: Retrieve Messages from SQS

1. Click **"Poll for Messages"** in SQS Console.
2. Select a message and **click "Delete"** after processing.

📌 **Expected Outcome:**
The message **is processed and removed from the queue**.

Chapter 5: SNS + SQS Integration (Fan-Out Pattern)

What is the SNS-SQS Fan-Out Pattern?

✔ **SNS can send messages to multiple SQS queues** simultaneously.
✔ Useful for **distributing tasks across multiple workers**.

Step 1: Subscribe SQS Queue to an SNS Topic

1. Open **SNS Console** → Select OrderUpdates.
2. Click **"Create Subscription"**.
3. **Protocol:** Choose **SQS**.
4. **Endpoint:** Select OrderProcessingQueue.
5. Click **"Create Subscription"**.

📌 **Expected Outcome:**
✔ When a message is **published to SNS**, it is **automatically delivered to the SQS queue**.

Step 2: Send a Message to SNS and Process It via SQS

1. Open **SNS Console** → Select OrderUpdates.
2. Click **"Publish Message"** and enter:
3. {
4.   "order_id": "5678",
5.   "status": "shipped"
6. }
7. Click **"Publish"**.
8. Go to **SQS Console** → Click **"Poll Messages"** to receive it.

📌 **Expected Outcome:**
✔ The **message is received in SQS** from SNS.

CHAPTER 6: MONITORING & BEST PRACTICES FOR SNS & SQS

1. Monitoring SNS & SQS with CloudWatch

✓ Use **Amazon CloudWatch Metrics** to track **message delivery, queue size, and errors**.
✓ Set up **CloudWatch Alarms** for failed messages.

---

2. Best Practices for AWS SNS

✓ **Use Message Filtering** to reduce unnecessary messages.
✓ **Enable SNS Encryption** (AWS KMS) for **secure message delivery**.
✓ **Use Dead-Letter Topics (DLT)** to capture undelivered messages.

---

3. Best Practices for AWS SQS

✓ **Use FIFO Queues** when **message order matters**.
✓ **Implement Dead-Letter Queues (DLQ)** to handle failed messages.
✓ **Use Long Polling** (WaitTimeSeconds > 0) to reduce empty responses.

📌 **Example:**
A **log processing system** uses an **SQS DLQ** to **store unprocessed messages** for later debugging.

CONCLUSION: MASTERING AWS SNS & SQS

By using **AWS SNS & SQS**, businesses can:
✅ **Decouple services and improve scalability**.
✅ **Enhance reliability and fault tolerance**.

✅ **Enable real-time notifications and asynchronous processing.**

✅ **Secure message delivery with encryption and IAM policies.**

FINAL EXERCISE:

1. **Create an SNS topic that notifies multiple subscribers via SMS and Lambda.**
2. **Set up an SQS queue that automatically processes incoming messages.**
3. **Implement SNS-SQS integration to fan out messages to multiple queues.**

AI/ML ON AWS

AWS SAGEMAKER OVERVIEW – STUDY MATERIAL

INTRODUCTION TO AWS SAGEMAKER

What is AWS SageMaker?

AWS **SageMaker** is a **fully managed machine learning (ML) service** that enables data scientists and developers to build, train, and deploy machine learning models quickly and efficiently. It eliminates the heavy lifting of managing ML infrastructure, allowing users to focus on developing and fine-tuning their models.

Why Use AWS SageMaker?

✓ **Fully Managed ML Environment** – Automates data preparation, training, and model deployment.
✓ **Integrated Development Environment (IDE)** – Provides Jupyter notebooks for ML model development.
✓ **AutoML Capabilities** – Uses SageMaker **Autopilot** to automatically train and optimize models.
✓ **Scalable Model Training** – Supports distributed training across multiple GPUs and TPUs.
✓ **One-Click Deployment** – Deploy trained models via **SageMaker Endpoints** for real-time inference.
✓ **Security & Compliance** – Supports **IAM, VPC isolation, encryption, and audit logs**.

📌 **Example Use Case:**
A **retail company** uses **AWS SageMaker** to build an ML model for **predicting customer demand**.

CHAPTER 1: KEY COMPONENTS OF AWS SAGEMAKER

Component	Description
SageMaker Studio	An IDE for building, training, and deploying ML models.
SageMaker Notebooks	Jupyter notebooks with pre-configured ML environments.
SageMaker Data Wrangler	Automates data preprocessing and feature engineering.
SageMaker Autopilot	Automated Machine Learning (AutoML) for model selection and tuning.
SageMaker JumpStart	Pre-trained models and ML solutions for easy deployment.
SageMaker Training	Distributed training for ML models on AWS infrastructure.
SageMaker Inference	Deploy trained models for real-time or batch predictions.
SageMaker Pipelines	CI/CD for ML models, automating the ML workflow.

CHAPTER 2: SETTING UP AWS SAGEMAKER

Step 1: Open SageMaker Studio

1. Open **AWS Console** → Navigate to **SageMaker**.
2. Click **"SageMaker Studio"**.
3. Select **"Create Domain"** and configure:

- **IAM Role:** SageMakerExecutionRole (with S3 and CloudWatch permissions).
- **VPC & Subnets:** Select a secure network configuration.
- **Storage Configuration:** Choose default or custom S3 buckets.

4. Click **"Create"**.

📌 **Expected Outcome:**
AWS **creates a SageMaker environment** with an integrated Jupyter notebook.

Step 2: Create a Jupyter Notebook in SageMaker

1. Open **SageMaker Studio** → Click **"Notebooks"**.
2. Click **"Create a New Notebook"**.
3. Choose a **Kernel** (e.g., Python 3, TensorFlow, PyTorch, Scikit-Learn).
4. Click **"Create"**.

📌 **Expected Outcome:**
A **Jupyter notebook is ready for ML model development**.

CHAPTER 3: PREPARING DATA WITH SAGEMAKER DATA WRANGLER

What is Data Wrangler?

AWS **SageMaker Data Wrangler** simplifies **data preprocessing, feature engineering, and visualization**.

Step 1: Import Data

1. Open **SageMaker Studio** → Click **"Data Wrangler"**.

2. Choose **"Import Dataset"**.
3. Select **S3, Redshift, or an external data source**.

Step 2: Perform Data Preprocessing

1. Use **Data Wrangler's UI** to apply:
    - **Filtering & Cleaning** (handle missing values, outliers).
    - **Feature Engineering** (scaling, one-hot encoding).
    - **Data Transformations** (log transformations, normalization).
2. Click **"Export to SageMaker Notebook"** for training.

📌 **Expected Outcome:**
Data is **cleaned and transformed**, ready for model training.

CHAPTER 4: TRAINING AN ML MODEL IN AWS SAGEMAKER

Step 1: Define Training Script

Create a Python script (train.py) for training an ML model using **Scikit-Learn**:

```
from sklearn.ensemble import RandomForestClassifier

from sklearn.model_selection import train_test_split

from sklearn.datasets import load_iris

import joblib


# Load dataset

iris = load_iris()
```

```
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target,
test_size=0.2)


# Train model

model = RandomForestClassifier(n_estimators=100)

model.fit(X_train, y_train)


# Save model

joblib.dump(model, "model.joblib")
```

Step 2: Train Model in SageMaker

Run the following **Python script** in a Jupyter notebook to train a model using SageMaker:

```
import boto3

import sagemaker

from sagemaker.sklearn.estimator import SKLearn


# Set up SageMaker session

sagemaker_session = sagemaker.Session()

role = sagemaker.get_execution_role()


# Define model training

sklearn_estimator = SKLearn(
    entry_point="train.py",
    framework_version="0.23-1",
    instance_type="ml.m5.large",
    role=role,
)

# Start training
sklearn_estimator.fit()
```

📌 **Expected Outcome:**
SageMaker **trains the ML model** using the specified compute instance.

CHAPTER 5: DEPLOYING AN ML MODEL WITH AWS SAGEMAKER

Step 1: Deploy Model Using SageMaker Endpoints

Once the model is trained, deploy it as a **real-time endpoint**:

```
predictor = sklearn_estimator.deploy(
    instance_type="ml.m5.large",
    initial_instance_count=1
)
```

Step 2: Test Model Deployment

Send test data to the deployed model:

```
response = predictor.predict([[5.1, 3.5, 1.4, 0.2]])
print("Predicted Class:", response)
```

📌 **Expected Outcome:**

The model **predicts the class label** for the given input data.

CHAPTER 6: AUTOMATING ML WORKFLOWS WITH SAGEMAKER PIPELINES

What is SageMaker Pipelines?

AWS **SageMaker Pipelines** automates the ML lifecycle, including **data processing, training, validation, and deployment**.

Step 1: Define an ML Pipeline

```
from sagemaker.workflow.pipeline import Pipeline

from sagemaker.workflow.steps import ProcessingStep, TrainingStep


# Define pipeline steps

step_process = ProcessingStep(...)

step_train = TrainingStep(...)


# Create pipeline

pipeline = Pipeline(

    name="MyMLPipeline",

    steps=[step_process, step_train]

)


# Execute pipeline
```

pipeline.start()

📌 **Expected Outcome:**
A **CI/CD pipeline for ML models** is executed.

CHAPTER 7: MONITORING & BEST PRACTICES FOR AWS SAGEMAKER

1. Monitor Training & Inference Jobs

✔ Use **Amazon CloudWatch** to track **training logs, model accuracy, and failure rates**.
✔ Enable **SageMaker Model Monitor** for **drift detection** in deployed models.

---

2. Best Practices for AWS SageMaker

✔ **Use AutoML (SageMaker Autopilot)** for automatic model selection.
✔ **Enable Model Explainability** using **SageMaker Clarify**.
✔ **Implement Hyperparameter Tuning** with **SageMaker Automatic Model Tuning**.
✔ **Secure ML Workloads** using **IAM roles, VPC isolation, and encryption**.

📌 **Example:**
A **finance company** uses **SageMaker Model Monitor** to detect model drift in fraud detection models.

CHAPTER 8: REAL-WORLD USE CASES FOR AWS SAGEMAKER

1. Predictive Maintenance

✓ Uses SageMaker to analyze IoT sensor data and predict **equipment failures**.

2. Customer Recommendation Systems

✓ Trains ML models to recommend **personalized product suggestions**.

3. Fraud Detection

✓ Monitors **transaction patterns** and detects anomalies using SageMaker.

4. Autonomous Vehicles

✓ Uses SageMaker **Reinforcement Learning (RL)** to train self-driving car models.

CONCLUSION: MASTERING AWS SAGEMAKER

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

✅ **Streamline machine learning workflows.**
✅ **Automate data preparation, training, and deployment.**
✅ **Scale ML workloads efficiently with cloud infrastructure.**
✅ **Enhance security and compliance for AI models.**

FINAL EXERCISE:

1. **Train a deep learning model in SageMaker using TensorFlow or PyTorch.**
2. **Deploy a real-time inference endpoint for an image classification model.**

3. **Implement model monitoring using SageMaker Model Monitor**.

AWS Rekognition for Image Processing – Study Material

Introduction to AWS Rekognition

What is AWS Rekognition?

AWS **Rekognition** is a **fully managed AI-powered image and video analysis service** that enables developers to add powerful **image recognition, object detection, facial analysis, and text extraction** capabilities to their applications.

Why Use AWS Rekognition?

✓ **Pre-trained AI Models** – Detect faces, objects, scenes, and text without training a custom model.
✓ **Face Recognition & Emotion Analysis** – Identify individuals, emotions, and facial attributes.
✓ **Text Detection (OCR)** – Extract text from images and videos.
✓ **Moderation & Content Safety** – Detect inappropriate or unsafe content.
✓ **Scalable & Cost-Effective** – Pay-as-you-go pricing with high availability.
✓ **Integration with AWS Services** – Works with **S3, Lambda, DynamoDB, and more**.

📌 **Example Use Case:**
A **security application** uses AWS Rekognition to **detect unauthorized individuals in surveillance footage**.

Chapter 1: Key Features of AWS Rekognition

Feature	**Description**

Object & Scene Detection	Identifies objects (cars, animals, furniture) and scenes (beach, city, office).
Facial Analysis	Detects age range, emotions, gender, and facial attributes.
Face Comparison & Recognition	Matches faces against a database of known individuals.
Text Detection (OCR)	Extracts text from images (license plates, billboards, documents).
Unsafe Content Moderation	Flags explicit or inappropriate content.
Celebrity Recognition	Identifies well-known personalities in images and videos.
Custom Labels (Custom AI Training)	Train a custom Rekognition model for specific object detection needs.

CHAPTER 2: SETTING UP AWS REKOGNITION

Step 1: Create an AWS S3 Bucket for Image Storage

1. Open **AWS S3 Console** → Click **"Create Bucket"**.
2. **Bucket Name:** rekognition-image-bucket-12345.
3. Choose **Region** (e.g., us-east-1).
4. Enable **Public Access Blocking** for security.
5. Click **"Create Bucket"**.

📌 **Expected Outcome:**

An **S3 bucket** is created to store images for analysis.

Step 2: Upload an Image to S3

1. Open **S3 Console** → Click on rekognition-image-bucket-12345.
2. Click **"Upload"** → Select an image file (e.g., face.jpg).
3. Click **"Upload"** to store the image in S3.

📌 **Expected Outcome:**
The **image is uploaded and ready for analysis**.

Chapter 3: Detecting Objects & Scenes in an Image

Step 1: Use AWS Rekognition to Detect Labels

Run the following **Python script** using **Boto3** (AWS SDK for Python):

```
import boto3


# Initialize AWS Rekognition client

rekognition = boto3.client('rekognition', region_name='us-east-1')


# Analyze image stored in S3

response = rekognition.detect_labels(

    Image={'S3Object': {'Bucket': 'rekognition-image-bucket-12345', 'Name': 'face.jpg'}},

    MaxLabels=10

)
```

```
# Print detected labels

for label in response['Labels']:

    print(f"Detected: {label['Name']} - Confidence: {label['Confidence']:.2f}%")
```

📌 **Expected Outcome:**
✔ The script detects **objects and scenes** (e.g., **"Person", "Car", "Building"**).
✔ Returns confidence scores (e.g., **"Person: 98.5%"**).

Chapter 4: Face Detection & Analysis

Step 1: Detect Faces in an Image

Modify the script to **detect facial attributes**:

```
response = rekognition.detect_faces(

    Image={'S3Object': {'Bucket': 'rekognition-image-bucket-12345', 'Name': 'face.jpg'}},

    Attributes=['ALL']

)


# Print detected face attributes

for face in response['FaceDetails']:

    print(f"Gender: {face['Gender']['Value']}")

    print(f"Age Range: {face['AgeRange']['Low']} - {face['AgeRange']['High']}")
    print(f"Emotion: {face['Emotions'][0]['Type']} (Confidence: {face['Emotions'][0]['Confidence']:.2f}%)")
```

📌 **Expected Outcome:**

✓ Detects **gender, age range, and emotions** (e.g., **Happy, Sad, Angry**).

✓ Useful for **customer sentiment analysis**.

Chapter 5: Face Comparison & Recognition

Step 1: Compare Two Faces

Upload two images (face1.jpg and face2.jpg) and compare them:

```
response = rekognition.compare_faces(
    SourceImage={'S3Object': {'Bucket': 'rekognition-image-bucket-12345', 'Name': 'face1.jpg'}},
    TargetImage={'S3Object': {'Bucket': 'rekognition-image-bucket-12345', 'Name': 'face2.jpg'}}
)


for match in response['FaceMatches']:
    print(f"Match Confidence: {match['Similarity']:.2f}%")
```

📌 **Expected Outcome:**

✓ Determines **whether the faces in two images belong to the same person**.

Chapter 6: Text Detection (OCR) with AWS Rekognition

Step 1: Detect Text in an Image

```
response = rekognition.detect_text(
    Image={'S3Object': {'Bucket': 'rekognition-image-bucket-12345', 'Name': 'text.jpg'}}
)


for text in response['TextDetections']:
    print(f"Detected Text: {text['DetectedText']} (Confidence: {text['Confidence']:.2f}%)")
```

📌 **Expected Outcome:**

✔ Extracts **text from an image** (e.g., **"STOP", "Caution", "License Plate 1234"**).
✔ Useful for **document scanning, license plate recognition, and signboard analysis**.

Chapter 7: Content Moderation with AWS Rekognition

Step 1: Detect Unsafe Content

```
response = rekognition.detect_moderation_labels(
    Image={'S3Object': {'Bucket': 'rekognition-image-bucket-12345', 'Name': 'image.jpg'}}
)


for label in response['ModerationLabels']:
    print(f"Moderation Label: {label['Name']} (Confidence: {label['Confidence']:.2f}%)")
```

📌 **Expected Outcome:**

✔ Detects **inappropriate content** (e.g., **violence, nudity, explicit text**).

✔ Useful for **social media content moderation**.

CHAPTER 8: AWS REKOGNITION VIDEO ANALYSIS

AWS Rekognition also supports **real-time video analysis** to detect **faces, objects, and scenes in live video streams**.

Use Case: Detect Objects in a Video

```
response = rekognition.start_label_detection(
    Video={'S3Object': {'Bucket': 'rekognition-video-bucket', 'Name': 'video.mp4'}}
)


job_id = response['JobId']

print(f"Started Video Processing Job: {job_id}")
```

📌 **Expected Outcome:**

✔ Processes **videos stored in S3** and detects objects like **cars, people, animals, etc.**.

✔ Useful for **traffic monitoring and security applications**.

CHAPTER 9: BEST PRACTICES FOR AWS REKOGNITION

✔ **Use S3 for Image Storage** – Store images in **Amazon S3** for scalability.

✔ **Optimize Cost by Using Rekognition API Limits** – Analyze only

necessary images.
✓ **Secure Image Data with IAM Roles** – Restrict access to Rekognition API.
✓ **Use Custom Labels for Industry-Specific Models** – Train Rekognition for specialized object detection.

📌 **Example:**
A **healthcare company** trains **AWS Rekognition** to **identify medical conditions in X-ray images**.

CONCLUSION: MASTERING AWS REKOGNITION

By using **AWS Rekognition**, businesses can:
✅ **Automate image and video analysis.**
✅ **Enhance security and identity verification.**
✅ **Improve content moderation and compliance.**
✅ **Scale AI-powered applications with minimal effort.**

FINAL EXERCISE:

1. **Train a custom Rekognition model using AWS Custom Labels.**
2. **Develop a real-time face recognition system using AWS Lambda + API Gateway.**
3. **Integrate AWS Rekognition with SNS to trigger alerts based on detected objects.**

AWS IoT & Edge Computing

AWS IoT Core – Study Material

Introduction to AWS IoT Core

What is AWS IoT Core?

AWS **IoT Core** is a **managed cloud service** that enables connected devices to interact with cloud applications securely and efficiently. It provides **low-latency messaging, data processing, and real-time analytics** for IoT applications.

Why Use AWS IoT Core?

✓ **Scalability** – Handles millions of connected devices.

✓ **Secure Device Communication** – Uses **MQTT, HTTP, WebSockets** with authentication via **X.509 certificates**.

✓ **Data Processing & Storage** – Integrates with **AWS Lambda, S3, DynamoDB, and Kinesis**.

✓ **Device Shadowing** – Maintains device state even when offline.

✓ **AI & ML Integration** – Enhances IoT data insights using **AWS SageMaker and Rekognition**.

📌 **Example Use Case:**
A **smart home system** uses AWS IoT Core to **control devices (lights, thermostats, cameras) via MQTT messages**.

Chapter 1: Key Features of AWS IoT Core

Feature	Description

MQTT Protocol Support	Uses lightweight **MQTT (Message Queuing Telemetry Transport)** for low-power devices.
Device Shadow	Stores last known state of a device for offline access.
Rules Engine	Processes IoT data and triggers AWS services (e.g., Lambda, S3, DynamoDB).
AWS IoT Analytics	Analyzes IoT data for trends and insights.
IoT Device Defender	Provides security monitoring for IoT devices.
IoT Greengrass	Enables local computing on edge devices.

CHAPTER 2: SETTING UP AWS IOT CORE

Step 1: Create an AWS IoT Thing (Device)

1. Open **AWS IoT Core Console** → Click **"Manage"** → **"Things"**.
2. Click **"Create Thing"** → Name it MyIoTDevice.
3. Click **"Next"** → Select **"Create a Certificate"**.
4. Download:
   - **Device Certificate**
   - **Private Key**
   - **Root CA Certificate**
5. Click **"Activate"** → **Attach a Policy** allowing IoT permissions.

📌 **Expected Outcome:**
A **virtual IoT device is created** with authentication credentials.

CHAPTER 3: CONNECTING A DEVICE TO AWS IOT CORE

Step 1: Install MQTT Client (Python Boto3)

```
pip install paho-mqtt boto3
```

Step 2: Publish a Message to AWS IoT Core

```
import paho.mqtt.client as mqtt


# AWS IoT Endpoint

BROKER = "your-aws-iot-endpoint.amazonaws.com"

TOPIC = "my/iot/topic"


# MQTT Client

client = mqtt.Client()

client.connect(BROKER, 8883)


# Publish a test message

client.publish(TOPIC, "Hello from AWS IoT!")

print("Message sent to IoT Core")
```

📌 **Expected Outcome:**

✔ The device **publishes a message to AWS IoT Core** using MQTT.

Step 3: Subscribe to IoT Messages

Modify the script to **subscribe and listen** for messages:

```
def on_message(client, userdata, message):
    print(f"Received message: {message.payload.decode()}")


client.subscribe(TOPIC)
client.on_message = on_message
client.loop_forever()
```

📌 **Expected Outcome:**
✔ The device **receives messages** sent to the IoT topic.

CHAPTER 4: IMPLEMENTING AWS IOT DEVICE SHADOW

What is an IoT Device Shadow?

AWS **Device Shadow** keeps a **persistent virtual copy of a device's state** in the cloud. It allows applications to query and update device states even when the device is offline.

Step 1: Update Device Shadow

```
import boto3


iot_client = boto3.client('iot-data', region_name='us-east-1')


shadow_payload = {
    "state": {
        "desired": {
            "temperature": 25
    }
  }
}


response = iot_client.update_thing_shadow(
  thingName="MyIoTDevice",
  payload=str(shadow_payload)
)


print("Device shadow updated")
```

📌 **Expected Outcome:**
✔ The device **updates its shadow state to 25°C**.

Step 2: Retrieve Device Shadow

```
response =
iot_client.get_thing_shadow(thingName="MyIoTDevice")
shadow_data = response["payload"].read()
print("Device Shadow Data:", shadow_data)
```

📌 **Expected Outcome:**
✔ Retrieves the **latest device state**, even if the device is offline.

Chapter 5: Processing IoT Data Using the Rules Engine

AWS **IoT Rules Engine** routes messages to AWS services **(Lambda, S3, DynamoDB, etc.)**.

Step 1: Create an IoT Rule to Store Data in DynamoDB

1. Open **AWS IoT Console** → Go to **"Message Routing"** → **"Rules"**.
2. Click **"Create Rule"** → Name it StoreIoTData.
3. SQL Query:
4. SELECT temperature, humidity FROM 'iot/sensor/data'
5. Choose **"DynamoDB Action"** → Select iot-sensor-data-table.
6. Click **"Create Rule"**.

📌 **Expected Outcome:**
✔ IoT sensor data is **automatically stored in DynamoDB**.

Chapter 6: Real-Time IoT Data Analysis Using AWS IoT Analytics

Step 1: Enable IoT Analytics

1. Open **AWS IoT Analytics Console**.
2. Create a **"Data Store"** to store IoT sensor data.
3. Create a **"Pipeline"** to process incoming data.
4. Run **queries** to analyze device trends.

📌 **Expected Outcome:**
✔ Insights about **temperature trends, device performance, and alerts**.

Chapter 7: Securing IoT Devices Using AWS IoT Device Defender

✓ **Detect security threats** with **AWS IoT Device Defender**.
✓ **Monitor device behavior** (unusual traffic, unauthorized access).
✓ **Set up security alerts** to notify administrators.

```
iot_client.create_security_profile(
   securityProfileName='IoT-Security-Profile',
   behaviors=[
      {
         'name': 'UnusualTraffic',
         'metric': 'aws:message-byte-size',
         'criteria': {'comparisonOperator': 'greater-than', 'value': 5000}
      }
   ]
)
```

📌 **Expected Outcome:**
✓ IoT Device Defender **flags unusual device behavior**.

Chapter 8: Best Practices for AWS IoT Core

✓ **Use MQTT for Low Power Consumption** – MQTT reduces network latency and power usage.
✓ **Enable Encryption & Authentication** – Use **X.509 certificates for secure device authentication**.
✓ **Implement IoT Device Shadows** – Maintain **device states for disconnected devices**.

✓ **Process IoT Data with AWS Lambda** – Automate event-driven workflows.
✓ **Use AWS Greengrass for Edge Computing** – Run **local ML models** without cloud dependency.

📌 **Example:**
A **smart factory** uses AWS **IoT Core and Lambda** to **trigger maintenance alerts based on machine vibration data**.

Chapter 9: Real-World Use Cases of AWS IoT Core

1. Smart Agriculture

✓ IoT **sensors monitor soil moisture** and **automatically trigger irrigation**.

2. Smart Cities

✓ AWS IoT **controls streetlights, traffic signals, and public transport**.

3. Industrial IoT (IIoT)

✓ **Factories monitor machinery health** and detect failures.

4. Healthcare Monitoring

✓ **Wearable devices track patient vitals** and send alerts to hospitals.

Conclusion: Mastering AWS IoT Core

By using **AWS IoT Core**, businesses can:
✅ **Connect and manage millions of IoT devices securely**.
✅ **Automate real-time data processing with AWS Lambda**.

✅ **Enhance security using IoT Device Defender**.

✅ **Scale IoT applications efficiently with the cloud**.

FINAL EXERCISE:

1. **Create an AWS IoT device and publish messages via MQTT.**
2. **Use the IoT Rules Engine to store sensor data in S3 or DynamoDB.**
3. **Enable security monitoring with AWS IoT Device Defender.**

AWS GREENGRASS – STUDY MATERIAL

INTRODUCTION TO AWS GREENGRASS

What is AWS Greengrass?

AWS **Greengrass** is an **edge computing service** that allows IoT devices to process, analyze, and respond to data locally while still being connected to AWS cloud services. It helps businesses **reduce latency, improve security, and ensure real-time processing at the edge**.

Why Use AWS Greengrass?

✓ **Edge Computing** – Runs applications on local devices instead of in the cloud.

✓ **Offline Operation** – Devices continue working even without an internet connection.

✓ **Machine Learning at the Edge** – Runs AI/ML models locally using AWS SageMaker.

✓ **Secure Device Communication** – Uses **MQTT, HTTPS, and local message queues**.

✓ **Seamless Integration** – Works with AWS IoT Core, Lambda, S3, DynamoDB, and more.

📌 **Example Use Case:**
A **smart factory** uses AWS Greengrass to **process machine sensor data in real-time and trigger automated maintenance alerts**.

CHAPTER 1: KEY FEATURES OF AWS GREENGRASS

Feature	Description

Local Processing & Machine Learning	Runs Lambda functions, containers, and AI models on edge devices.
Device Shadow	Stores device states in the cloud for disconnected devices.
Secure Communication	Uses **X.509 certificates and AWS IAM** for secure messaging.
ML Inference at the Edge	Deploys and runs **trained ML models** on IoT devices.
AWS IoT Greengrass Connectors	Integrates with third-party applications like **Splunk, Twilio, and ServiceNow**.
Greengrass Stream Manager	Handles data streams locally and syncs with AWS cloud.

CHAPTER 2: SETTING UP AWS GREENGRASS

Step 1: Install AWS Greengrass on an IoT Device

1. **Select a Greengrass-compatible device** (e.g., **Raspberry Pi, NVIDIA Jetson, or an EC2 instance**).
2. Connect to the device via SSH:
3. ssh pi@your-device-ip
4. Download Greengrass Core software:
5. wget https://d1onfpft10uf50.cloudfront.net/greengrass-core-latest.tar.gz

📌 **Expected Outcome:**
AWS Greengrass **installation files are downloaded to the device**.

Step 2: Configure the Greengrass Device in AWS IoT

1. Open **AWS IoT Console** → Navigate to **Greengrass**.
2. Click **"Create a Greengrass Group"** → Name it SmartFactoryGroup.
3. **Create a Core Device** → Download generated security certificates.
4. Transfer the certificates to the IoT device:
5. scp -r certs/ pi@your-device-ip:/greengrass/certs/

📌 **Expected Outcome:**
Greengrass Core **is securely registered with AWS IoT**.

Step 3: Start the Greengrass Core Service

Run the following command on the IoT device:

sudo ./greengrassd start

Verify that Greengrass Core is running:

ps aux | grep greengrass

📌 **Expected Outcome:**
The **Greengrass Core service is running** on the IoT device.

Chapter 3: Deploying an AWS Lambda Function to Greengrass

Step 1: Create a Lambda Function for Edge Processing

1. Open **AWS Lambda Console** → Click **"Create Function"**.

2. Select **"Author from Scratch"** → Name it EdgeProcessingFunction.
3. Choose **Python 3.9** as the runtime.
4. Modify the function to process local sensor data:

```
import greengrasssdk


client = greengrasssdk.client('iot-data')


def lambda_handler(event, context):
   print("Processing data at the edge...")
   message = {"temperature": event['temperature'], "status": "Processed"}
   client.publish(topic="iot/edge/data", payload=str(message))
```

5. Click **"Deploy to Greengrass"**.

📌 **Expected Outcome:**
✔ AWS Lambda **runs on the Greengrass device** and processes data locally.

Chapter 4: Running Machine Learning Models on AWS Greengrass

AWS Greengrass **supports deploying trained ML models** from AWS SageMaker.

Step 1: Train an ML Model in AWS SageMaker

1. Open **SageMaker Studio** → Train an image recognition model.

2. Convert the model to **TensorFlow Lite (.tflite)** or **ONNX** for edge inference.
3. Store the model in an S3 bucket.

Step 2: Deploy ML Model to Greengrass Device

1. Open **Greengrass Console** → Select SmartFactoryGroup.
2. Click **"Add Machine Learning Resource"** → Choose the trained model from S3.
3. Deploy the model to the Greengrass device.

📌 **Expected Outcome:**
✓ The **AI model runs locally on the IoT device**, reducing latency.

Chapter 5: AWS Greengrass Security Best Practices

✓ **Use IAM Roles** – Assign minimal required permissions for Greengrass devices.
✓ **Enable Secure Communication** – Use **TLS encryption** and **X.509 certificates**.
✓ **Implement Local Authentication** – Use **AWS Secrets Manager** for credentials.
✓ **Monitor Device Logs** – Enable **CloudWatch Logs** to track anomalies.

📌 **Example:**
A **smart traffic system** encrypts all **IoT device messages with TLS** to prevent data leaks.

Chapter 6: AWS Greengrass Use Cases

1. Smart Agriculture

✓ Uses **IoT sensors** to monitor soil conditions and control irrigation.

2. Industrial Automation (IIoT)

✓ Analyzes **machine sensor data** and predicts failures in real-time.

3. Autonomous Vehicles

✓ Runs **AI-based object detection models** for self-driving cars.

4. Healthcare Wearables

✓ Processes **heart rate and ECG data locally** before sending alerts.

CHAPTER 7: REAL-TIME EVENT PROCESSING WITH AWS GREENGRASS STREAM MANAGER

AWS **Greengrass Stream Manager** manages large data streams locally before syncing with AWS cloud.

Example: Stream Sensor Data to AWS Kinesis

```
import greengrasssdk


client = greengrasssdk.client('iot-data')


def lambda_handler(event, context):
   client.publish(topic="iot/sensor/stream", payload=str(event))
```

📌 **Expected Outcome:**

✓ **Processes high-speed IoT data streams** before syncing with AWS.

CHAPTER 8: AWS GREENGRASS VS AWS IOT CORE

Feature	**AWS Greengrass**	**AWS IoT Core**
Processing Location	Runs workloads **locally** on IoT devices	Runs workloads **in the cloud**
Offline Capability	Works without internet	Requires an active internet connection
Use Case	AI inference, local automation	Cloud-based IoT data processing

---

CONCLUSION: MASTERING AWS GREENGRASS

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

✅ **Process and analyze IoT data locally**.

✅ **Reduce latency and improve response times**.

✅ **Enhance security with local authentication**.

✅ **Integrate with AWS ML services for AI-powered IoT applications**.

FINAL EXERCISE:

1. **Install AWS Greengrass on a Raspberry Pi or EC2 instance**.
2. **Deploy an AWS Lambda function to process sensor data locally**.
3. **Run an ML model on an edge device using AWS Greengrass**.

ASSIGNMENT

BUILD A WORKFLOW AUTOMATION WITH AWS STEP FUNCTIONS

SOLUTION: BUILD A WORKFLOW AUTOMATION WITH AWS STEP FUNCTIONS (STEP-BY-STEP GUIDE)

This guide will walk you through **building a workflow automation with AWS Step Functions**, integrating multiple AWS services using a **serverless approach**.

Step 1: Understanding AWS Step Functions

What are AWS Step Functions?

AWS **Step Functions** is a **serverless orchestration service** that allows you to **automate workflows** by coordinating multiple AWS services (such as **Lambda, S3, DynamoDB, SNS, and SQS**) into a **state machine**.

📌 **Example Use Case:**
A workflow that **processes customer orders**:

1. **Validates the order** using AWS Lambda.
2. **Stores order details** in DynamoDB.
3. **Sends a confirmation email** using SNS.
4. **Triggers an SQS message** to initiate shipping.

Step 2: Create an AWS Step Functions State Machine

1. Open **AWS Console** → Search for **"Step Functions"** → Click **"Create State Machine"**.
2. Choose **"Author with Code"** → Select **"Standard Workflow"**.

3. Name the state machine: OrderProcessingWorkflow.

Step 3: Define the Workflow in JSON (Amazon States Language - ASL)

Paste the following JSON **workflow definition**:

```
{
 "Comment": "Order Processing Workflow",
 "StartAt": "ValidateOrder",
 "States": {
  "ValidateOrder": {
   "Type": "Task",
   "Resource": "arn:aws:lambda:us-east-
1:123456789012:function:ValidateOrderFunction",
   "Next": "StoreOrder"
  },
  "StoreOrder": {
   "Type": "Task",
   "Resource": "arn:aws:lambda:us-east-
1:123456789012:function:StoreOrderFunction",
   "Next": "SendOrderConfirmation"
  },
  "SendOrderConfirmation": {
   "Type": "Task",
   "Resource": "arn:aws:sns:us-east-1:123456789012:OrderConfirmationTopic",
   "Next": "ShipOrder"
  },
  "ShipOrder": {
   "Type": "Task",
   "Resource": "arn:aws:sqs:us-east-1:123456789012:OrderShippingQueue",
   "End": true
  }
 }
}
```

📌 **Workflow Breakdown:**
✔ ValidateOrder – Calls a Lambda function to validate the order.
✔ StoreOrder – Saves order details in **DynamoDB**.
✔ SendOrderConfirmation – Sends an **SNS notification** to the customer.
✔ ShipOrder – Triggers an **SQS message** to the shipping system.

Step 4: Create AWS Lambda Functions for Workflow Steps

1. Validate Order (Lambda Function)

```
import json


def lambda_handler(event, context):
  if "order_id" in event and event["order_id"]:
    return {"order_id": event["order_id"], "status": "Validated"}
  else:
    raise ValueError("Invalid Order")
```

---

2. Store Order in DynamoDB (Lambda Function)

```
import json
import boto3


dynamodb = boto3.resource("dynamodb")
table = dynamodb.Table("OrdersTable")


def lambda_handler(event, context):
  table.put_item(Item={"order_id": event["order_id"], "status": "Stored"})
  return {"order_id": event["order_id"], "status": "Stored"}
```

Step 5: Create an SNS Topic for Order Confirmation

1. Open **AWS SNS Console** → Click **"Create Topic"**.
2. **Topic Name:** OrderConfirmationTopic.
3. **Protocol:** Email (or SMS, Lambda).
4. Subscribe an email address for notifications.

📌 **Expected Outcome:**
✓ Customers receive **order confirmation emails** when an order is placed.

Step 6: Create an SQS Queue for Shipping Requests

1. Open **AWS SQS Console** → Click **"Create Queue"**.
2. **Queue Name:** OrderShippingQueue.
3. Choose **Standard Queue**.
4. Configure the queue policy to allow Step Functions access.

📌 **Expected Outcome:**
✓ Orders are **queued for shipping** once confirmed.

Step 7: Deploy the State Machine & Execute Workflow

1. Open **Step Functions Console** → Click **"Deploy"**.
2. Click **"Start Execution"**.
3. Enter input JSON:

```
{
 "order_id": "12345",
 "customer_email": "customer@example.com"
}
```

📌 **Expected Outcome:**
✓ The workflow executes **step-by-step**, invoking Lambda, DynamoDB, SNS, and SQS.
✓ Customers receive an **email confirmation** via SNS.

✓ Order details are **stored in DynamoDB**.
✓ The shipping process is **queued in SQS**.

Step 8: Monitor & Debug Workflow Execution

1. Open **Step Functions Console** → Click on **Executions**.
2. View **Execution History** for detailed logs.
3. Click **each step** to inspect input/output JSON.

📌 **Expected Outcome:**
✓ Easily **debug errors** and **re-run failed steps**.

Step 9: Optimize & Secure Your Workflow

✓ **Enable AWS IAM Policies** – Restrict access to Step Functions & Lambda.
✓ **Use CloudWatch Logs** – Monitor execution performance.
✓ **Add Error Handling in Step Functions** – Implement retries on failures.
✓ **Integrate AI/ML** – Use AWS SageMaker for **predictive order processing**.

📌 **Example:**
A **retail company** enhances order processing by **detecting fraudulent orders using AI**.

CONCLUSION: MASTERING AWS STEP FUNCTIONS FOR WORKFLOW AUTOMATION

By using **AWS Step Functions**, businesses can:
✅ **Automate multi-step processes with serverless orchestration**.

✅ **Integrate multiple AWS services (Lambda, DynamoDB, SNS, SQS, etc.).**
✅ **Monitor and debug workflows in real-time.**
✅ **Optimize operations with retry mechanisms and event-driven automation.**

FINAL EXERCISE:

1. **Modify the workflow to include a payment processing step using Stripe API.**
2. **Use Amazon S3 to store order invoices after processing.**
3. **Extend the workflow by triggering AWS Lambda for fraud detection.**

CREATE AN AI/ML MODEL USING AWS SAGEMAKER

Solution: Create an AI/ML Model Using AWS SageMaker (Step-by-Step Guide)

This guide will walk you through **building, training, and deploying an AI/ML model using AWS SageMaker**.

Step 1: Understanding AWS SageMaker

AWS **SageMaker** is a **fully managed machine learning service** that provides tools to **prepare data, train models, and deploy ML solutions** at scale.

📌 **Use Case Example:**
A **retail business** wants to use **AWS SageMaker** to **predict customer purchase behavior** based on past shopping data.

Step 2: Set Up AWS SageMaker

1. Open **AWS Console** → Search for **SageMaker**.
2. Click **"SageMaker Studio"** → **Create a New Domain** (if not already set up).
3. **Select an IAM Role** that has access to **S3, SageMaker, and CloudWatch**.
4. Click **"Create Domain"** → Launch **SageMaker Studio**.

📌 **Expected Outcome:**
A **Jupyter-based development environment** is ready.

Step 3: Prepare Your Data

1. Upload Data to an S3 Bucket

1. Open **AWS S3 Console** → Click **"Create Bucket"**.
2. **Bucket Name:** sagemaker-ml-dataset.
3. Click **"Create"** → Upload a CSV dataset (customer_data.csv).

2. Load Data into SageMaker Notebook

```
import boto3

import pandas as pd


# Define S3 bucket and file

bucket_name = "sagemaker-ml-dataset"

file_name = "customer_data.csv"


# Load data

s3_client = boto3.client("s3")

s3_client.download_file(bucket_name, file_name, file_name)


df = pd.read_csv(file_name)

print(df.head())
```

📌 **Expected Outcome:**

✔ Data is **loaded into SageMaker for model training**.

Step 4: Data Preprocessing & Feature Engineering

```
from sklearn.model_selection import train_test_split

from sklearn.preprocessing import StandardScaler


# Split dataset into training and testing sets

train, test = train_test_split(df, test_size=0.2, random_state=42)


# Normalize numeric features

scaler = StandardScaler()

train_scaled = scaler.fit_transform(train.drop("target", axis=1))

test_scaled = scaler.transform(test.drop("target", axis=1))


# Convert to DataFrame

train_scaled = pd.DataFrame(train_scaled,
columns=train.columns[:-1])

test_scaled = pd.DataFrame(test_scaled, columns=test.columns[:-
1])
```

📌 **Expected Outcome:**
✓ The data is **cleaned and normalized** for training.

Step 5: Train a Machine Learning Model in AWS SageMaker

1. Upload Processed Data to S3

```
import sagemaker
```

```
s3_train_path = f"s3://{bucket_name}/train.csv"
s3_test_path = f"s3://{bucket_name}/test.csv"

train.to_csv("train.csv", index=False)
test.to_csv("test.csv", index=False)

s3_client.upload_file("train.csv", bucket_name, "train.csv")
s3_client.upload_file("test.csv", bucket_name, "test.csv")
```

📌 **Expected Outcome:**

✔ Preprocessed data is **uploaded to S3**.

---

2. Define SageMaker Training Job

```
from sagemaker.sklearn.estimator import SKLearn

# Initialize SageMaker session
sagemaker_session = sagemaker.Session()
role = sagemaker.get_execution_role()

# Create an SKLearn estimator
sklearn_estimator = SKLearn(
    entry_point="train.py",
    framework_version="0.23-1",
    instance_type="ml.m5.large",
    role=role,
)


# Start training
sklearn_estimator.fit({"train": s3_train_path})
```

📌 **Expected Outcome:**
✔ AWS SageMaker **trains the model** using the provided dataset.

Step 6: Deploy the Trained Model

1. Deploy Model as a SageMaker Endpoint

```
predictor = sklearn_estimator.deploy(
    instance_type="ml.m5.large",
    initial_instance_count=1
)
```

📌 **Expected Outcome:**
✔ The trained model is **deployed as a REST API endpoint**.

Step 7: Test the Deployed Model

```
import json


# Example customer data input
input_data = [[45, 60000, 3]]  # Age, Income, Purchases
```

```
# Send data to SageMaker endpoint

response = predictor.predict(input_data)

print("Predicted Class:", response)
```

📌 **Expected Outcome:**
✓ The **model predicts the customer's purchase behavior**.

Step 8: Monitor Model Performance Using AWS CloudWatch

1. Open **AWS CloudWatch Console** → Navigate to **Logs**.
2. Select the SageMaker **endpoint logs**.
3. Monitor **API requests, model accuracy, and errors**.

📌 **Expected Outcome:**
✓ You can **track model performance** and troubleshoot issues.

Step 9: Automate Model Retraining Using SageMaker Pipelines

```
from sagemaker.workflow.pipeline import Pipeline

from sagemaker.workflow.steps import TrainingStep


# Define pipeline steps

step_train = TrainingStep(

    name="TrainMLModel",

    estimator=sklearn_estimator,

    inputs={"train": s3_train_path}

)
```

```
# Create pipeline
pipeline = Pipeline(
    name="CustomerBehaviorPipeline",
    steps=[step_train]
)

# Execute pipeline
pipeline.start()
```

📌 **Expected Outcome:**

✔ The model **automatically retrains on new data**.

Step 10: Optimize Model Performance

✔ **Use Hyperparameter Tuning** to improve accuracy:

```
from sagemaker.tuner import HyperparameterTuner, IntegerParameter

tuner = HyperparameterTuner(
    estimator=sklearn_estimator,
    objective_metric_name="validation:accuracy",
    hyperparameter_ranges={"max_depth": IntegerParameter(3, 10)},
    max_jobs=5,
    max_parallel_jobs=2
)


tuner.fit({"train": s3_train_path})
```

📌 **Expected Outcome:**

✔ The model **finds optimal hyperparameters** to improve accuracy.

CONCLUSION: MASTERING AWS SAGEMAKER FOR AI/ML MODEL DEPLOYMENT

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

✅ **Build machine learning models efficiently**.

✅ **Automate data preprocessing and training workflows**.

✅ **Deploy models as scalable REST API endpoints**.

✅ **Monitor and optimize ML models in real-time**.

FINAL EXERCISE:

1. **Train a deep learning model using SageMaker TensorFlow or PyTorch.**
2. **Deploy an image classification model for real-time predictions.**
3. **Enable SageMaker Model Monitor to track data drift in production.**