Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

◇ BIG DATA FUNDAMENTALS: HADOOP, SPARK, NOSQL DATABASES

📌 CHAPTER 1: INTRODUCTION TO BIG DATA

🔸 1.1 What is Big Data?

Big Data refers to extremely large and complex datasets that cannot be processed using traditional data processing techniques. These datasets require specialized tools and frameworks to store, analyze, and extract insights efficiently.

Characteristics of Big Data (The 5 Vs):

✓ **Volume** – Large amounts of data generated every second.
✓ **Velocity** – Rapid data generation and processing.
✓ **Variety** – Different data types (structured, semi-structured, unstructured).
✓ **Veracity** – Ensuring data quality and reliability.
✓ **Value** – Extracting meaningful insights from data.

🔸 1.2 Why is Big Data Important?

✓ **Business Growth** – Helps companies make data-driven decisions.
✓ **Real-Time Analytics** – Used in fraud detection, predictive analytics, and customer segmentation.
✓ **Automation & AI** – Supports machine learning models with vast datasets.
✓ **Scalability** – Allows organizations to handle increasing data loads.

📌 **Example:**
✓ **Social Media:** Facebook processes over **500 terabytes of data daily**.
✓ **E-commerce:** Amazon uses **big data analytics** to recommend products.

💡 **Conclusion:**
Big Data is the **foundation of modern analytics**, enabling businesses to gain a competitive edge through data-driven insights.

📌 CHAPTER 2: HADOOP – THE BIG DATA FRAMEWORK

🔹 2.1 What is Hadoop?

Hadoop is an open-source framework for **storing and processing large-scale datasets** in a distributed computing environment. It enables parallel data processing across multiple servers.

Components of Hadoop:

1️⃣ **HDFS (Hadoop Distributed File System)** – Stores large files across multiple machines.
2️⃣ **MapReduce** – A programming model for processing large datasets in parallel.
3️⃣ **YARN (Yet Another Resource Negotiator)** – Manages cluster resources efficiently.

🔹 **HBase** – A NoSQL database that provides real-time access to big data.

🔹 2.2 How Does Hadoop Work?

✓ **Step 1:** Data is **split into blocks** and distributed across multiple nodes using HDFS.
✓ **Step 2:** MapReduce **processes data in parallel** on different machines.
✓ **Step 3:** YARN **allocates resources** for efficient execution.
✓ **Step 4:** Processed data is stored or sent to a data warehouse for analysis.

📌 **Example:**
A bank uses Hadoop to analyze **millions of daily transactions** to detect fraud patterns in real-time.

🔹 2.3 Implementing Hadoop in Practice

✓ **Hadoop Installation (Local Mode):**

```
# Install Hadoop

sudo apt update

sudo apt install hadoop
```

✓ **Check Hadoop Installation:**

```
hadoop version
```

✓ **Run a Simple MapReduce Job:**

```
hadoop jar /usr/local/hadoop/hadoop-examples.jar wordcount input output
```

💡 **Conclusion:**
Hadoop is a **powerful distributed computing framework** used for handling massive datasets in industries like **finance, healthcare, and e-commerce**.

📌 Chapter 3: Apache Spark – Fast Data Processing

🔹 3.1 What is Apache Spark?

Apache Spark is an open-source, distributed data processing engine **faster than Hadoop**. It performs data processing **in-memory**, making it suitable for real-time analytics.

Key Features of Spark:

✓ **Lightning-Fast Processing** – Processes data 100x faster than Hadoop's MapReduce.
✓ **In-Memory Computation** – Stores intermediate results in memory instead of disk.
✓ **Supports Multiple Languages** – Works with Python, Java, Scala, and R.
✓ **Fault-Tolerant** – Automatically recovers from failures.
✓ **Integrates with Hadoop** – Uses HDFS for data storage.

🔹 3.2 Spark vs. Hadoop

Feature	Hadoop	Apache Spark
Processing Speed	Slower (Disk-based)	Faster (In-memory)
Data Handling	Batch Processing	Batch & Streaming
Ease of Use	Java-based	Supports Python, Scala

Use Case	Large-scale storage	Real-time analytics

📌 **Example:**

A ride-sharing company (e.g., Uber) uses **Apache Spark for real-time trip analytics** to optimize pricing and wait times.

🔹 **3.3 Implementing Spark in Python (PySpark)**

✓ **Install PySpark:**

```
pip install pyspark
```

✓ **Run a Simple PySpark Job:**

```
from pyspark.sql import SparkSession

# Create Spark Session
spark = SparkSession.builder.appName("BigData").getOrCreate()

# Load Data
data = [(1, "Alice", 29), (2, "Bob", 35), (3, "Charlie", 40)]
columns = ["ID", "Name", "Age"]

df = spark.createDataFrame(data, columns)

# Show Data
df.show()
```

💡 **Conclusion:**
Apache Spark is **ideal for real-time analytics, big data streaming, and machine learning applications**.

📌 Chapter 4: NoSQL Databases – Scalable Data Storage

🔹 4.1 What are NoSQL Databases?

NoSQL (Not Only SQL) databases store data in **non-tabular formats** such as key-value pairs, documents, graphs, or columns. They are designed for **scalability and high availability**.

Types of NoSQL Databases:

✓ **Document Stores** – MongoDB (stores data in JSON format).
✓ **Key-Value Stores** – Redis, DynamoDB (fast data retrieval).
✓ **Column Stores** – Apache Cassandra, HBase (distributed storage).
✓ **Graph Databases** – Neo4j (relationship-based data).

🔹 4.2 NoSQL vs. SQL Databases

Feature	SQL (Relational)	NoSQL (Non-Relational)
Schema	Fixed schema	Dynamic schema
Scalability	Vertical scaling	Horizontal scaling
Data Model	Tables (Rows & Columns)	Documents, Key-Value, Graphs
Best For	Structured data	Big Data, Unstructured data

📌 **Example:**
✓ **Netflix** uses **NoSQL (Cassandra) to store billions of user watch histories** for personalized recommendations.

🔹 4.3 Implementing MongoDB (Document-Based NoSQL)

✓ Install MongoDB:

sudo apt update

sudo apt install -y mongodb

✓ Start MongoDB Service:

sudo systemctl start mongod

✓ Insert & Query Data in Python:

```
from pymongo import MongoClient

# Connect to MongoDB
client = MongoClient("mongodb://localhost:27017/")

# Create Database & Collection
db = client["MovieDB"]
collection = db["Reviews"]

# Insert Data
collection.insert_one({"movie": "Inception", "rating": 9, "review": "Excellent!"})

# Fetch Data
```

```
for doc in collection.find():
    print(doc)
```

💡 **Conclusion:**
NoSQL databases **handle large-scale, unstructured data efficiently**, making them ideal for **real-time applications and analytics**.

📌 Summary & Next Steps

✅ **Key Takeaways:**

✓ **Hadoop** enables **distributed data storage and batch processing**.

✓ **Apache Spark** processes big data **faster with in-memory computations**.

✓ **NoSQL Databases** provide **scalable solutions for large-scale, unstructured data**.

✓ **Big Data technologies** power **AI, machine learning, and real-time analytics**.

📌 **Next Steps:**

- **Learn Spark Streaming for real-time data processing.**
- **Practice NoSQL queries in MongoDB and Cassandra.**
- **Build a big data pipeline combining Hadoop, Spark, and NoSQL.** 🚀

◇ Cloud Computing for Data Science: AWS, Google Cloud, Azure

📌 Chapter 1: Introduction to Cloud Computing for Data Science

🔹 1.1 What is Cloud Computing?

Cloud Computing is the **on-demand delivery of computing resources** such as servers, databases, storage, and AI services over the **internet**. Instead of maintaining **physical infrastructure**, users can leverage cloud platforms to **process, analyze, and store data efficiently**.

Key Benefits of Cloud Computing for Data Science:

✓ **Scalability** – Instantly increase or decrease resources based on workload.
✓ **Cost Efficiency** – Pay for what you use, avoiding upfront infrastructure costs.
✓ **Data Storage & Security** – Secure cloud storage solutions for large datasets.
✓ **Access to AI & ML Tools** – Pre-built machine learning models and analytics tools.
✓ **Collaboration & Integration** – Teams can work remotely with shared cloud environments.

📌 **Example:**
A **data scientist** working on a **predictive analytics model** can use **Google Cloud AI** to train machine learning models **without requiring expensive hardware**.

💡 **Conclusion:**
Cloud computing enables **faster, cost-effective, and scalable data science operations**.

📌 Chapter 2: Cloud Service Models for Data Science

Cloud services are classified into three major models:

Cloud Model	Description	Example Services
IaaS (Infrastructure as a Service)	Provides virtual machines, networking, and storage. Users manage OS, applications, and data.	AWS EC2, Google Compute Engine, Azure Virtual Machines
PaaS (Platform as a Service)	Provides a ready-made environment to deploy and run applications.	AWS Elastic Beanstalk, Google App Engine, Azure App Services
SaaS (Software as a Service)	Pre-built cloud applications that users can access over the internet.	Google Drive, AWS SageMaker, Azure ML Studio

📌 **Example:**
A **data analytics team** can use **Google BigQuery (PaaS)** for real-time data processing without setting up physical infrastructure.

💡 **Conclusion:**
Choosing the **right cloud model** depends on **project needs, budget, and scalability requirements**.

📌 Chapter 3: AWS for Data Science

🔹 3.1 Overview of AWS for Data Science

Amazon Web Services (AWS) is one of the most widely used cloud platforms for **data science, machine learning, and big data processing**.

Key AWS Services for Data Science:

AWS Service	Use Case
Amazon S3	Cloud storage for structured and unstructured data
AWS EC2	Virtual machines for running data science workloads
AWS SageMaker	Fully managed machine learning service
AWS Lambda	Serverless computing for event-driven data processing
Amazon Redshift	Cloud-based data warehouse for big data analytics
AWS Glue	ETL (Extract, Transform, Load) service for data pipelines

📌 Example:

A **data scientist** can use **AWS SageMaker** to **train, deploy, and manage machine learning models without managing infrastructure**.

🔹 3.2 Implementing a Machine Learning Model in AWS SageMaker

Step 1: Set Up SageMaker Environment

1. **Sign in to AWS Console**
2. Navigate to **Amazon SageMaker**
3. Click **Notebook Instances** → Create a new instance
4. Choose an **instance type (e.g., ml.t2.medium)**

Step 2: Load and Train a Model

```
import sagemaker

from sagemaker import get_execution_role

from sagemaker.sklearn import SKLearn


role = get_execution_role()


sklearn_estimator = SKLearn(
   entry_point='train.py',
   role=role,
   instance_type='ml.m4.xlarge',
   framework_version='0.23-1'
)


sklearn_estimator.fit({'train': 's3://your-bucket-name/dataset.csv'})
```

📌 **Example Output:**

✓ Model trained using **SageMaker's managed infrastructure**.

💡 **Conclusion:**
AWS provides **powerful cloud-based machine learning tools** that eliminate the need for **expensive on-premise hardware**.

📌 Chapter 4: Google Cloud Platform (GCP) for Data Science

🔹 4.1 Overview of GCP for Data Science

Google Cloud Platform (GCP) offers **scalable AI and big data services** that integrate seamlessly with popular data science tools.

Key GCP Services for Data Science:

GCP Service	Use Case
Google Colab	Cloud-based Jupyter notebooks for deep learning
BigQuery	Serverless, scalable data warehouse for large-scale queries
AI Platform	Machine learning model training and deployment
Cloud Storage	Storing datasets and machine learning artifacts
TensorFlow Cloud	Runs TensorFlow models on GCP

📌 **Example:**
A **data scientist** uses **BigQuery** to analyze **terabytes of customer transaction data** in seconds.

🔹 4.2 Running a Machine Learning Model on Google Colab

Google Colab allows running **Python and deep learning models on free GPUs**.

Step 1: Enable GPU in Google Colab

1. Open Google Colab
2. Click **Runtime → Change runtime type**
3. Select **GPU**

Step 2: Train a Deep Learning Model

```
import tensorflow as tf
from tensorflow.keras.models import Sequential
from tensorflow.keras.layers import Dense

# Create a simple neural network
model = Sequential([
    Dense(64, activation='relu', input_shape=(10,)),
    Dense(32, activation='relu'),
    Dense(1, activation='sigmoid')
])

# Compile the model
model.compile(optimizer='adam', loss='binary_crossentropy', metrics=['accuracy'])

# Display model summary
```

```
model.summary()
```

📌 **Example Output:**

✓ The **model runs on free Colab GPU**, reducing training time.

💡 **Conclusion:**

GCP provides **free and paid cloud services** for **data analysis, storage, and machine learning**.

📌 Chapter 5: Microsoft Azure for Data Science

🔹 5.1 Overview of Azure for Data Science

Microsoft Azure provides a variety of cloud services for **data science, AI, and analytics**.

Key Azure Services for Data Science:

Azure Service	Use Case
Azure Machine Learning	Automated machine learning (AutoML)
Azure Databricks	Big data analytics and AI workflows
Azure Storage	Secure cloud-based data storage
Azure SQL Database	Managed cloud database for data analysis
Azure AI Cognitive Services	Pre-trained AI models for NLP and Vision

📌 **Example:**

A **data engineer** can use **Azure Databricks** to process **big data pipelines with Apache Spark**.

🔹 5.2 Training a Model using Azure Machine Learning Studio

Step 1: Set Up Azure ML Workspace

1. Sign in to Azure Portal
2. Navigate to **Azure Machine Learning Studio**
3. Create a new **workspace**
4. Set up **Compute Instance** for model training

Step 2: Train a Machine Learning Model on Azure ML

```
from azureml.core import Workspace, Experiment
from azureml.train.automl import AutoMLConfig

ws = Workspace.from_config()
experiment = Experiment(ws, 'automl-classification')

automl_config = AutoMLConfig(
    task='classification',
    training_data=my_data,
    label_column_name='target',
    primary_metric='accuracy'
)

experiment.submit(automl_config)
```

📌 **Example Output:**
✓ Model is **automatically trained, optimized, and deployed** on Azure.

💡 **Conclusion:**

Azure offers **powerful AI & ML automation tools** for **quick model training**.

📌 Summary & Next Steps

✅ **Key Takeaways:**

✓ **AWS** is best for **end-to-end machine learning** and **enterprise AI services**.

✓ **Google Cloud** offers **free GPU computing and scalable big data analytics**.

✓ **Azure** provides **automated machine learning and enterprise-grade AI solutions**.

📌 **Next Steps:**

🔹 **Deploy machine learning models on AWS, Google Cloud, and Azure.**

🔹 **Use cloud GPUs (Google Colab, AWS SageMaker) for deep learning projects.**

🔹 **Explore serverless computing for data processing pipelines.**

🚀

◇ Deploying Machine Learning Models on Cloud

📌 Chapter 1: Introduction to Cloud Deployment

🔹 1.1 What is Cloud Deployment in Machine Learning?

Cloud deployment refers to **hosting and serving machine learning models on cloud platforms,** making them accessible for real-time inference and large-scale processing. Instead of running models on local machines**, cloud services provide scalable and cost-effective environments** for deployment.

🔹 1.2 Why Deploy ML Models on the Cloud?

✓ **Scalability** – Cloud platforms handle large-scale requests efficiently.
✓ **Accessibility** – Models can be accessed via APIs from anywhere.
✓ **Cost-Effective** – Pay only for the computing resources used.
✓ **Performance Optimization** – Cloud GPUs/TPUs accelerate model inference.
✓ **Integration** – Cloud APIs allow easy connection with applications.

🔹 1.3 Steps in Deploying an ML Model to the Cloud

1️⃣ **Train & Save the Model** – Develop, test, and store the model.
2️⃣ **Choose a Cloud Platform** – Select **AWS, Google Cloud, Azure,** etc.
3️⃣ **Containerization** – Use **Docker** to package dependencies.
4️⃣ **Set Up API for Serving Predictions** – Use **Flask, FastAPI, or TensorFlow Serving**.
5️⃣ **Deploy on Cloud Infrastructure** – Use **Google Cloud Run, AWS Lambda, or Azure Functions**.

6️⃣ **Monitor & Maintain** – Track performance and update when needed.

📌 **Example:**
A **customer review sentiment analysis model** is deployed on **Google Cloud** to analyze real-time user feedback.

💡 **Conclusion:**
Cloud deployment makes machine learning models **efficient, accessible, and scalable for real-world applications**.

📌 Chapter 2: Preparing a Machine Learning Model for Deployment

🔹 2.1 Train and Save a Model

Before deployment, a **trained model** must be exported in a format suitable for cloud deployment.

Train & Save a Sample Model (Iris Dataset)

```
import numpy as np

import pandas as pd

import joblib

from sklearn.datasets import load_iris

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier


# Load dataset

iris = load_iris()
```

```
X_train, X_test, y_train, y_test = train_test_split(iris.data, iris.target, test_size=0.2, random_state=42)


# Train a model
model = RandomForestClassifier(n_estimators=100)
model.fit(X_train, y_train)


# Save the trained model
joblib.dump(model, "iris_model.pkl")


print("Model saved as iris_model.pkl")
```

📌 **Key Points:**

✓ **Model is trained on the Iris dataset.**

✓ **Model is saved in .pkl format for easy loading in cloud services.**

💡 **Conclusion:**

Saving models in joblib or TensorFlow SavedModel format allows **quick deployment on cloud platforms**.

📌 Chapter 3: Creating an API for the Model

🔹 3.1 What is an API?

An **API (Application Programming Interface)** allows applications to communicate with the model **via HTTP requests**. Flask or FastAPI can be used to create a **REST API for the ML model**.

3.2 Building an API using Flask

```
from flask import Flask, request, jsonify
import joblib
import numpy as np

# Load the trained model
model = joblib.load("iris_model.pkl")

# Initialize Flask app
app = Flask(__name__)

# Define API endpoint
@app.route("/predict", methods=["POST"])
def predict():
    data = request.json  # Get JSON input
    features = np.array(data["features"]).reshape(1, -1)  # Convert to array
    prediction = model.predict(features)  # Make prediction
    return jsonify({"prediction": int(prediction[0])})  # Return response

# Run the API
if __name__ == "__main__":
    app.run(host="0.0.0.0", port=5000, debug=True)
```

📌 **Key Features of the API:**

✓ Accepts **JSON requests** with input features.

✓ Uses joblib to **load the trained model**.

✓ Returns **predictions in JSON format**.

💡 **Conclusion:**

APIs provide a **simple way to expose ML models for cloud applications**.

📌 Chapter 4: Containerizing the Model with Docker

🔸 4.1 What is Docker?

Docker is a containerization tool that packages an application and its dependencies into a portable **container**, ensuring that the application runs **consistently across different environments**.

🔸 4.2 Writing a Dockerfile for the ML Model

Create a file called Dockerfile in the project directory:

```
# Use Python base image

FROM python:3.8


# Set working directory

WORKDIR /app


# Copy project files

COPY . .
```

```
# Install dependencies
RUN pip install flask numpy joblib scikit-learn

# Expose the API port
EXPOSE 5000

# Run the API
CMD ["python", "app.py"]
```

🔹 **4.3 Build and Run the Docker Container**

1️⃣ **Build the Docker Image**

```
docker build -t iris_model_api .
```

2️⃣ **Run the Container**

```
docker run -p 5000:5000 iris_model_api
```

📌 **Key Benefits of Docker:**

✓ **Ensures consistency** across cloud environments.
✓ **Packages dependencies with the model**.
✓ **Easily deployable to any cloud service.**

💡 **Conclusion:**

Docker simplifies ML deployment by **creating portable, reproducible environments**.

📌 Chapter 5: Deploying on Cloud (Google Cloud, AWS, Azure)

🔹 **5.1 Deploying on Google Cloud Run**

1️⃣ Install Google Cloud SDK & Authenticate

gcloud auth login

gcloud config set project [PROJECT_ID]

2️⃣ Build & Push Docker Image to Google Container Registry

gcloud builds submit --tag gcr.io/[PROJECT_ID]/iris_model_api

3️⃣ Deploy to Google Cloud Run

gcloud run deploy iris-model-api --image gcr.io/[PROJECT_ID]/iris_model_api --platform managed --allow-unauthenticated

📌 Deployment URL Example:

https://iris-model-api-xyz.a.run.app/predict

🔹 5.2 Deploying on AWS Lambda & API Gateway

1️⃣ Package Model & API in a ZIP File

zip -r iris_lambda.zip app.py iris_model.pkl

2️⃣ Upload to AWS Lambda

✔ Create a **Lambda function** in AWS.
✔ Set **runtime to Python 3.8**.
✔ Upload iris_lambda.zip.

3️⃣ Expose as an API using AWS API Gateway

✔ Create a **REST API** in API Gateway.
✔ Add a **POST method** to invoke Lambda.
✔ Deploy & obtain an API endpoint.

🔸 5.3 Deploying on Azure Functions

1️⃣ Install Azure CLI & Authenticate

az login

az group create --name iris-group --location eastus

2️⃣ Deploy Function to Azure

az functionapp create --name iris-api --resource-group iris-group --consumption-plan-location eastus --runtime python --os-type Linux

📌 Example Endpoint:

https://iris-api.azurewebsites.net/api/predict

💡 Conclusion:
Cloud services like **Google Cloud Run, AWS Lambda, and Azure Functions** make **serverless deployment efficient and scalable**.

📌 Summary & Next Steps

✅ Key Takeaways:

✓ Cloud deployment makes ML models scalable and accessible.
✓ Flask & FastAPI APIs expose ML models for real-world use.
✓ Docker containers ensure deployment consistency.
✓ Google Cloud, AWS, and Azure offer powerful hosting solutions.

📌 Next Steps:

- **Deploy a deep learning model on AWS or Google Cloud.**
- **Use Kubernetes for large-scale ML deployment.**
- **Integrate CI/CD pipelines for automated model updates.** 🚀

◇ Serverless Computing for Data Science Applications

📌 Chapter 1: Introduction to Serverless Computing

🔹 1.1 What is Serverless Computing?

Serverless computing is a cloud computing execution model where the cloud provider manages the **infrastructure, scaling, and maintenance,** allowing developers to focus solely on writing code. Unlike traditional cloud computing, **serverless does not require managing servers** or provisioning resources manually.

Key Features of Serverless Computing:

✓ **No Server Management** – The cloud provider handles provisioning, scaling, and maintenance.
✓ **Auto-Scaling** – Resources scale automatically based on demand.
✓ **Pay-Per-Use** – Users are charged only for execution time, reducing costs.
✓ **Event-Driven Execution** – Functions execute in response to events like API calls, database updates, or file uploads.
✓ **Fast Deployment** – No need to configure server instances, leading to faster application development.

📌 **Example:**
A **data scientist** can deploy a **Python script for preprocessing a dataset** using AWS Lambda without worrying about managing servers.

💡 **Conclusion:**
Serverless computing simplifies **deployment, scaling, and cost**

management, making it ideal for **data science workflows and machine learning applications**.

📌 Chapter 2: Serverless Computing in Data Science

🔹 2.1 Why Use Serverless Computing for Data Science?

Data Science workflows involve **data preprocessing, model training, deployment, and monitoring**. Serverless computing helps by:

✓ **Reducing Infrastructure Overhead** – No need to manage virtual machines or Kubernetes clusters.

✓ **Auto-Scaling for Large Datasets** – Automatically handles large datasets with distributed execution.

✓ **Cost Efficiency** – Pay only for the computation time used.

✓ **Event-Driven Execution** – Triggers workflows based on new data arrival or user requests.

✓ **Easy Integration with Cloud Services** – Works with AWS Lambda, Google Cloud Functions, and Azure Functions.

📌 Example:

✓ A **serverless function** triggers an **ETL pipeline** when a new dataset is uploaded to AWS S3.

✓ A **real-time inference model** processes incoming images using a serverless API.

💡 Conclusion:

Serverless computing **removes infrastructure barriers,** making Data Science applications more **efficient and scalable**.

📌 Chapter 3: Serverless Platforms for Data Science

🔹 3.1 Popular Serverless Platforms

Several cloud providers offer **serverless computing solutions** suitable for Data Science applications:

Platform	Serverless Service	Best Use Case
AWS Lambda	Serverless functions for real-time processing	ETL, data preprocessing, event-driven ML
Google Cloud Functions	Serverless event-driven computing	API-driven data science, ML inference
Azure Functions	Serverless execution on Microsoft Azure	IoT, real-time data processing
IBM Cloud Functions	Functions as a Service (FaaS)	AI-powered chatbots, serverless automation
AWS Fargate	Serverless container orchestration	Large-scale ML training, batch processing

📌 **Example:**
A **serverless pipeline** in AWS Lambda can automate **data ingestion from an API**, preprocess the data, and store it in a database.

💡 **Conclusion:**
Choosing the right **serverless platform** depends on **data science requirements, integration needs, and pricing models**.

📌 Chapter 4: Implementing Serverless Data Science Workflows

🔹 4.1 Example 1: Data Preprocessing with AWS Lambda

Step 1: Install AWS CLI and Boto3

```
pip install boto3
```

Step 2: Create a Python Function for Data Cleaning

```
import boto3
import pandas as pd
import json

def lambda_handler(event, context):
    # Load Data from S3
    s3 = boto3.client('s3')
    bucket_name = 'your-bucket-name'
    file_key = 'dataset.csv'

    response = s3.get_object(Bucket=bucket_name, Key=file_key)
    df = pd.read_csv(response['Body'])

    # Data Cleaning
    df.dropna(inplace=True)  # Remove missing values
    df = df[df['price'] > 0]  # Remove invalid prices

    # Save cleaned data back to S3
    cleaned_key = 'cleaned_dataset.csv'
    csv_buffer = df.to_csv(index=False)
```

```
    s3.put_object(Bucket=bucket_name, Key=cleaned_key,
Body=csv_buffer)


    return {
        'statusCode': 200,
        'body': json.dumps("Data Preprocessing Completed!")
    }
```

📌 **Execution Flow:**

✓ This function is **triggered when a new dataset is uploaded** to an S3 bucket.

✓ It **cleans the data** and saves the processed dataset back to S3.

💡 **Conclusion:**

Serverless data preprocessing **automates ETL pipelines** with **minimal infrastructure cost**.

🔹 **4.2 Example 2: Serverless Machine Learning Model Deployment**

A **serverless API** can be used to **serve machine learning predictions**.

Step 1: Train & Save an ML Model

```
from sklearn.ensemble import RandomForestClassifier
import pickle

# Load dataset
from sklearn.datasets import load_iris
```

```
iris = load_iris()
X, y = iris.data, iris.target

# Train model
model = RandomForestClassifier()
model.fit(X, y)

# Save model as a pickle file
with open("model.pkl", "wb") as file:
    pickle.dump(model, file)
```

Step 2: Deploy Model in AWS Lambda

```
import boto3
import pickle
import json
import numpy as np

s3 = boto3.client('s3')

def lambda_handler(event, context):
    # Load Model from S3
    bucket_name = 'your-model-bucket'
    model_key = 'model.pkl'
```

```
response = s3.get_object(Bucket=bucket_name, Key=model_key)
model = pickle.loads(response['Body'].read())

# Read input data
input_data = np.array(event['data']).reshape(1, -1)

# Make Prediction
prediction = model.predict(input_data)

return {
    'statusCode': 200,
    'body': json.dumps({'prediction': prediction.tolist()})
}
```

📌 **Execution Flow:**

✓ The model is **stored in S3** and loaded inside AWS Lambda.

✓ The Lambda function **receives input data, runs the model, and returns predictions**.

✓ This function can be **triggered via an API Gateway** to act as a REST API.

💡 **Conclusion:**

Serverless ML deployment makes model inference **cost-effective, scalable, and real-time**.

📌 Chapter 5: Serverless vs. Traditional Computing for Data Science

Feature	Serverless Computing	Traditional Computing
Infrastructure Management	Fully managed by cloud provider	Requires manual setup
Scalability	Auto-scaling	Manual scaling needed
Pricing Model	Pay-as-you-go (per execution)	Pay for allocated resources
Ideal Use Case	Event-driven tasks (e.g., ML inference, ETL)	Large-scale ML training, batch processing

📌 **Example:**

✓ **Serverless** is ideal for **real-time inference (chatbots, fraud detection)**.

✓ **Traditional servers** are better for **deep learning training (GPU-intensive tasks)**.

📌 Summary & Next Steps

✅ **Key Takeaways:**

✓ **Serverless computing eliminates server management** for Data Science applications.

✓ **AWS Lambda, Google Cloud Functions, and Azure Functions** enable event-driven ML workflows.

✓ **Serverless pipelines are cost-effective, scalable, and ideal for real-time inference.**

📌 **Next Steps:**

- **Deploy a machine learning model using AWS Lambda and API Gateway.**
- **Automate data processing with serverless ETL workflows.**
- **Explore serverless AI pipelines with AWS SageMaker, Vertex AI, or Azure ML.** 🚀

◇ Real-time Data Processing with Apache Kafka

📌 Chapter 1: Introduction to Real-time Data Processing

🔹 1.1 What is Real-time Data Processing?

Real-time data processing refers to the **continuous processing of data streams** as soon as they are generated. Unlike traditional batch processing, which processes data in fixed intervals, real-time processing allows businesses to respond to events **instantly**.

Why is Real-time Processing Important?

✓ **Low Latency** – Provides insights and actions immediately.
✓ **Scalability** – Can handle large volumes of data.
✓ **Fault Tolerance** – Ensures reliability even when failures occur.
✓ **Event-driven Architecture** – Reacts to data changes as they happen.

Industries Using Real-time Processing

✓ **Financial Services** – Fraud detection, stock market analysis.
✓ **E-commerce** – Personalized recommendations, order tracking.
✓ **Social Media** – Live feed updates, trending topic analysis.
✓ **IoT & Smart Devices** – Sensor data streaming, predictive maintenance.

📌 **Example:**
A **fraud detection system** continuously monitors credit card transactions. If an unusual transaction occurs, **an alert is generated immediately**.

💡 **Conclusion:**
Real-time data processing **enhances decision-making, improves user experiences, and supports automation**.

📌 Chapter 2: Introduction to Apache Kafka

🔹 2.1 What is Apache Kafka?

Apache Kafka is an **open-source distributed event streaming platform** used for handling real-time data. It is designed for **scalability, durability, and high throughput**, making it ideal for large-scale data processing.

Key Features of Kafka

✓ **Publish-Subscribe Messaging Model** – Supports real-time message streaming.
✓ **Distributed Architecture** – Runs across multiple nodes for fault tolerance.
✓ **High Throughput** – Can handle **millions of messages per second**.
✓ **Persistent Storage** – Logs messages durably for later retrieval.
✓ **Scalability** – Easily scales across multiple servers.

🔹 2.2 Kafka Architecture Overview

Kafka consists of four main components:

Component	Description
Producer	Sends (publishes) data to Kafka topics.
Broker	Stores and manages messages (Kafka servers).

Topic	Logical group of messages, similar to a message queue.
Consumer	Reads (subscribes to) messages from Kafka topics.

📌 **Example:**

A **stock market application** uses Kafka to **stream real-time price updates**, where:

✓ **Producers** send stock prices.

✓ **Kafka Brokers** store and distribute them.

✓ **Consumers** (traders, analytics systems) receive updates instantly.

💡 **Conclusion:**

Kafka is a **high-performance, scalable** solution for **real-time messaging and event streaming**.

📌 Chapter 3: Setting Up Apache Kafka

🔸 3.1 Installing Kafka Locally

Step 1: Download and Extract Kafka

```
wget https://downloads.apache.org/kafka/3.0.0/kafka_2.13-3.0.0.tgz

tar -xvzf kafka_2.13-3.0.0.tgz

cd kafka_2.13-3.0.0
```

Step 2: Start Zookeeper (Required for Kafka Coordination)

```
bin/zookeeper-server-start.sh config/zookeeper.properties
```

Step 3: Start Kafka Broker

```
bin/kafka-server-start.sh config/server.properties
```

📌 **Why Zookeeper?**

✓ Zookeeper manages **metadata, leader election, and broker coordination** in Kafka.

🔹 **3.2 Creating Topics and Sending Messages**

Step 1: Create a Kafka Topic

bin/kafka-topics.sh --create --topic my-topic --bootstrap-server localhost:9092 --partitions 3 --replication-factor 1

Step 2: List Topics

bin/kafka-topics.sh --list --bootstrap-server localhost:9092

Step 3: Start a Kafka Producer (Send Messages)

bin/kafka-console-producer.sh --topic my-topic --bootstrap-server localhost:9092

📌 **Type a message and press Enter. It will be sent to Kafka.**

Step 4: Start a Kafka Consumer (Read Messages)

bin/kafka-console-consumer.sh --topic my-topic --from-beginning --bootstrap-server localhost:9092

📌 **Example Output:**

Producer: Hello Kafka!

Consumer: Hello Kafka!

💡 **Conclusion:**

Kafka **handles real-time messaging efficiently**, allowing **multiple consumers** to process events.

📌 Chapter 4: Real-time Data Processing with Kafka

🔸 4.1 Streaming Data with Kafka & Python

Step 1: Install Kafka-Python Library

```
pip install kafka-python
```

Step 2: Create a Kafka Producer in Python

```
from kafka import KafkaProducer


producer = KafkaProducer(bootstrap_servers='localhost:9092')


# Send a message
producer.send('my-topic', b'Hello from Python!')
producer.flush()
print("Message Sent!")
```

Step 3: Create a Kafka Consumer in Python

```
from kafka import KafkaConsumer


consumer = KafkaConsumer('my-topic',
bootstrap_servers='localhost:9092', auto_offset_reset='earliest')


print("Reading Messages:")
for message in consumer:
  print(message.value.decode())
```

📌 **Expected Output:**

Message Sent!

Reading Messages:

Hello from Python!

💡 **Conclusion:**
Using Python, we can **integrate Kafka into machine learning pipelines, IoT applications, and real-time dashboards**.

📌 Chapter 5: Integrating Kafka with Streaming Frameworks

🔸 5.1 Kafka + Apache Spark Streaming

Apache Spark is a **real-time data processing engine** that works well with Kafka.

Streaming Data from Kafka to Spark

```
from pyspark.sql import SparkSession

from pyspark.sql.functions import col


# Initialize Spark Session

spark = SparkSession.builder.appName("KafkaStreaming").getOrCreate()


# Read Kafka Stream

df = spark.readStream.format("kafka")\

  .option("kafka.bootstrap.servers", "localhost:9092")\

  .option("subscribe", "my-topic")\
```

```
.load()


# Extract values and display

df.selectExpr("CAST(value AS
STRING)").writeStream.outputMode("append").format("console").st
art().awaitTermination()
```

📌 **Use Cases:**

✓ **Live dashboards (Real-time analytics).**

✓ **Anomaly detection (Fraud, Security).**

✓ **Predictive analytics (Stock market, Customer behavior).**

📌 CHAPTER 6: KAFKA VS. OTHER MESSAGING SYSTEMS

Feature	Kafka	RabbitMQ	Apache Flink
Best Use Case	Real-time streaming	Message queuing	Stream processing
Scalability	High	Medium	High
Performance	High throughput	Low latency	High latency
Persistence	Yes (logs messages)	Optional	Yes

📌 **Example:**

✓ **Kafka** is best for **log aggregation, event-driven architectures, and microservices**.

✓ **RabbitMQ** is best for **reliable message delivery in enterprise applications**.

✓ **Apache Flink** is best for **complex event processing and real-time analytics**.

📌 Summary & Next Steps

✅ Key Takeaways:

✓ Kafka enables **real-time data streaming at scale**.

✓ Kafka uses **Producers, Topics, Brokers, and Consumers** for message distribution.

✓ **Python and Apache Spark** can process Kafka streams for analytics.

✓ Kafka is ideal for **fraud detection, log monitoring, and IoT applications**.

📌 Next Steps:

🔹 **Integrate Kafka with machine learning models (e.g., real-time fraud detection).**

🔹 **Deploy a Kafka cluster using Docker/Kubernetes for scalability.**

🔹 **Use Kafka Connect for data ingestion from databases (MySQL, PostgreSQL, MongoDB).** 🚀

ASSIGNMENT 1:

DEPLOY A MACHINE LEARNING MODEL AS A REST API USING FLASK & AWS.

📌 SOLUTION: DEPLOYING A MACHINE LEARNING MODEL AS A REST API USING FLASK & AWS

- **Objective**

The goal of this assignment is to **train and deploy a machine learning model as a REST API using Flask and host it on AWS EC2**. The API will take user input (features), pass it to the model, and return predictions.

- **Step 1: Train and Save a Machine Learning Model**

Before deploying, we must train a machine learning model and save it for inference. Here, we'll use the **Iris dataset** and train a **Random Forest Classifier**.

1.1 Import Required Libraries

```
import numpy as np

import pandas as pd

import joblib

from sklearn.model_selection import train_test_split

from sklearn.ensemble import RandomForestClassifier

from sklearn.datasets import load_iris
```

1.2 Load and Preprocess Data

```
# Load the Iris dataset

iris = load_iris()
```

```
X = iris.data  # Features
y = iris.target  # Labels


# Split dataset into training and test sets
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

1.3 Train the Model

```
# Train a Random Forest Classifier
model = RandomForestClassifier(n_estimators=100, random_state=42)
model.fit(X_train, y_train)
```

1.4 Save the Trained Model

```
# Save model using joblib
joblib.dump(model, "iris_model.pkl")
print("Model saved successfully!")
```

📌 **The model is now saved as iris_model.pkl, which will be used for deployment.**

🔹 Step 2: Create a REST API using Flask

Now, we will create a Flask API to **serve the trained model**.

2.1 Install Flask

```
pip install flask
```

2.2 Create the Flask Application (app.py)

```
from flask import Flask, request, jsonify
import joblib
import numpy as np

# Initialize Flask App
app = Flask(__name__)

# Load the trained model
model = joblib.load("iris_model.pkl")

@app.route("/")
def home():
    return "Iris Classification API is Running!"

@app.route("/predict", methods=["POST"])
def predict():
    try:
        # Get JSON input
        data = request.get_json()
        features = np.array(data["features"]).reshape(1, -1)  # Convert to
array

        # Make prediction
```

```
        prediction = model.predict(features)[0]
        response = {"prediction": int(prediction)}

        return jsonify(response)

    except Exception as e:
        return jsonify({"error": str(e)})

# Run Flask API
if __name__ == "__main__":
    app.run(debug=True, host="0.0.0.0", port=5000)
```

- **Step 3: Test the API Locally**

Before deploying, let's test the API on **localhost**.

3.1 Run the Flask App

python app.py

✓ The API will start at **http://127.0.0.1:5000/**

3.2 Test API using Postman or CURL

✓ Using Postman:

- Select **POST** method.
- Enter URL: http://127.0.0.1:5000/predict

- Set Body → Raw → JSON:

```
{
  "features": [5.1, 3.5, 1.4, 0.2]
}
```

- Click **Send**, and you should receive:

```
{
  "prediction": 0
}
```

✓ Using CURL in Terminal:

```
curl -X POST "http://127.0.0.1:5000/predict" -H "Content-Type: application/json" -d '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

📌 **The API is now working locally! Let's deploy it on AWS.**

🔹 **Step 4: Deploy Flask API on AWS EC2**

4.1 Launch an AWS EC2 Instance

1. Go to **AWS EC2 Dashboard**.
2. Click **Launch Instance**.
3. Select **Ubuntu 20.04** as OS.
4. Choose **t2.micro (Free Tier)**.
5. Configure Security Group:
   - Allow **port 22** (SSH).
   - Allow **port 5000** (Flask API).

6. Click **Launch** and create a new key pair.

4.2 Connect to EC2 via SSH

ssh -i your-key.pem ubuntu@your-ec2-public-ip

4.3 Install Dependencies on EC2

sudo apt update

sudo apt install python3-pip -y

pip3 install flask joblib numpy pandas scikit-learn

4.4 Transfer Model & App to EC2

On your local machine, run:

scp -i your-key.pem iris_model.pkl app.py ubuntu@your-ec2-public-ip:~

4.5 Run the Flask App on EC2

python3 app.py

✓ The API will now run on http://0.0.0.0:5000/.

◆ Step 5: Access Flask API from the Internet

5.1 Find Public IP of EC2

Run:

curl ifconfig.me

Your public IP (e.g., 3.123.45.67) will be displayed.

5.2 Test API Using Postman or CURL

✓ Postman:

- Use the public IP of your EC2 instance:
- http://3.123.45.67:5000/predict
- Body:

```
{
  "features": [5.1, 3.5, 1.4, 0.2]
}
```

✓ CURL Command:

```
curl -X POST "http://3.123.45.67:5000/predict" -H "Content-Type: application/json" -d '{"features": [5.1, 3.5, 1.4, 0.2]}'
```

📌 Expected Response:

```
{
  "prediction": 0
}
```

📌 Step 6: Run Flask as a Background Process

By default, Flask stops when you exit SSH. To keep it running:

```
nohup python3 app.py > output.log 2>&1 &
```

✓ Now, the Flask API **runs continuously**, even after logging out.

📌 Step 7: Secure the API with Nginx & Gunicorn

For production, use **Nginx + Gunicorn** instead of running Flask directly.

7.1 Install Gunicorn & Nginx

sudo apt install nginx -y

pip3 install gunicorn

7.2 Configure Gunicorn

Run:

gunicorn -w 4 -b 0.0.0.0:5000 app:app

✓ Now, your API runs **efficiently** with multiple worker threads.

📌 Summary & Next Steps

✅ Key Takeaways:

✓ We **trained and saved a machine learning model**.
✓ We **created a REST API using Flask**.
✓ We **deployed the API on AWS EC2**.
✓ We **tested predictions using Postman & CURL**.

📌 Next Steps:

- **Use Docker to containerize the API for better deployment.**
- **Deploy the API on AWS Lambda for serverless execution.**
- **Secure API using authentication tokens (JWT, OAuth2).** 🚀

ASSIGNMENT 2:

PROCESS A BIG DATA PIPELINE USING APACHE SPARK.

📌 Solution: Process a Big Data Pipeline using Apache Spark

🔹 Objective

The goal of this assignment is to process **Big Data using Apache Spark** by building a **data pipeline** that includes **data ingestion, transformation, and analysis**. Apache Spark is widely used for **large-scale data processing** due to its **speed, scalability, and distributed computing capabilities**.

🔹 Step 1: Install and Set Up Apache Spark

🔸 1.1 Installing Apache Spark in a Local Environment

1️⃣ **Download and Install Spark**

- Go to the official Apache Spark website.
- Select **Spark 3.0+ (Pre-built for Hadoop 3.2)**.
- Extract the downloaded file to a local directory.

2️⃣ **Set Environment Variables** (Linux/macOS)

```
export SPARK_HOME=~/spark-3.0.0-bin-hadoop3.2
export PATH=$SPARK_HOME/bin:$PATH
export PYSPARK_PYTHON=python3
```

3️⃣ **Verify Spark Installation**

```
spark-shell  # Opens Spark interactive shell
```

🔸 1.2 Installing Apache Spark in Google Colab (Cloud-based Setup)

Google Colab does not come with Spark pre-installed, so we need to **install and configure Spark manually**.

```
!apt-get install openjdk-8-jdk-headless -qq > /dev/null

!wget -q https://downloads.apache.org/spark/spark-3.1.2/spark-3.1.2-bin-hadoop3.2.tgz

!tar xf spark-3.1.2-bin-hadoop3.2.tgz

!pip install findspark


import os

import findspark


os.environ["JAVA_HOME"] = "/usr/lib/jvm/java-8-openjdk-amd64"

os.environ["SPARK_HOME"] = "/content/spark-3.1.2-bin-hadoop3.2"


findspark.init()
```

📌 **Example Output:**

✓ Spark is successfully installed and configured.

💡 **Conclusion:**

Apache Spark can be set up **locally or in cloud environments (Google Colab, AWS, Azure, Databricks)**.

🔹 Step 2: Initialize a Spark Session

A **SparkSession** is required to start processing data in Apache Spark.

```
from pyspark.sql import SparkSession


# Initialize Spark Session

spark = SparkSession.builder.appName("BigDataPipeline").getOrCreate()


# Print Spark Configuration

print(spark)
```

📌 **Why SparkSession?**

✓ Provides **entry point** for interacting with Spark.

✓ Manages **distributed computing** for big data processing.

✓ Supports **DataFrame API** for structured data analysis.

🔹 **Step 3: Load and Explore a Big Data File**

3.1 Loading a Large Dataset (CSV Format)

We will use a **large dataset** (e.g., NYC Taxi Data) for processing.

```
# Load a CSV dataset into Spark DataFrame

df = spark.read.csv("nyc_taxi_data.csv", header=True, inferSchema=True)


# Display first few rows

df.show(5)
```

📌 **Dataset Details:**

✓ **nyc_taxi_data.csv** contains millions of taxi ride records with columns like **trip duration, fare amount, pickup/drop-off locations**.

3.2 Understanding the Data

```
# Print Schema (Column names & data types)

df.printSchema()


# Count total rows

print(f"Total Rows in Dataset: {df.count()}")


# Summary statistics

df.describe().show()
```

📌 **Output Example:**

✓ The dataset contains **millions of rows** and **multiple numerical & categorical features**.

💡 **Conclusion:**

Loading data into **Spark DataFrame** enables efficient handling of **large-scale datasets**.

🔹 Step 4: Data Cleaning & Transformation

4.1 Handling Missing Values

```
# Check for missing values

df.select([df[c].isNull().sum().alias(c) for c in df.columns]).show()
```

```
# Drop rows with missing values
df_cleaned = df.na.drop()
```

📌 **Why Handle Missing Data?**

✓ Missing values **reduce model accuracy**.

✓ Spark provides efficient **null handling functions**.

4.2 Converting Data Types

```
from pyspark.sql.functions import col

# Convert 'trip_duration' to integer and 'fare_amount' to float
df_transformed = df_cleaned.withColumn("trip_duration", col("trip_duration").cast("integer")) \
                           .withColumn("fare_amount", col("fare_amount").cast("float"))

# Verify data types
df_transformed.printSchema()
```

📌 **Why Convert Data Types?**

✓ **Ensures numerical consistency** for analysis.

✓ **Prepares data** for machine learning models.

4.3 Feature Engineering (Adding a New Column)

```
from pyspark.sql.functions import when
```

```
# Create a new column 'trip_category' based on trip duration
df_transformed = df_transformed.withColumn(
   "trip_category",
   when(col("trip_duration") < 300, "Short") \
   .when((col("trip_duration") >= 300) & (col("trip_duration") < 1200), "Medium") \
   .otherwise("Long")
)

df_transformed.show(5)
```

📌 **Why Feature Engineering?**

✓ Helps in **categorizing trips for better analysis**.

✓ Improves **machine learning model performance**.

🔹 **Step 5: Data Aggregation & Analysis**

5.1 Grouping Data by Categories

```
df_transformed.groupBy("trip_category").count().show()
```

📌 **Example Output:**

```
+-------------+------+
| trip_category | count |
+-------------+------+
| Short       | 550000 |
```

| Medium | 300000 |

| Long | 150000 |

+-------------+------+

✓ **Short trips are most common,** indicating frequent **city commutes**.

5.2 Computing Average Fare by Trip Category

df_transformed.groupBy("trip_category").agg({"fare_amount": "avg"}).show()

📌 **Example Output:**

✓ **Long trips have higher average fares,** as expected.

🔹 Step 6: Storing & Exporting Processed Data

6.1 Save Processed Data to Parquet (Optimized Format)

df_transformed.write.mode("overwrite").parquet("processed_nyc_taxi_data.parquet")

📌 **Why Use Parquet?**

✓ **Efficient storage** – Compressed columnar format.

✓ **Faster queries** – Optimized for Spark.

🔹 Step 7: Machine Learning with Spark MLlib

We can now apply **machine learning models** to predict **taxi fare**.

from pyspark.ml.regression import LinearRegression

from pyspark.ml.feature import VectorAssembler

```
# Select features & target variable
feature_columns = ["trip_duration", "passenger_count"]
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
df_ml = assembler.transform(df_transformed)

# Train-test split
train_data, test_data = df_ml.randomSplit([0.8, 0.2], seed=42)

# Train Linear Regression Model
lr = LinearRegression(featuresCol="features", labelCol="fare_amount")
lr_model = lr.fit(train_data)

# Evaluate Model
predictions = lr_model.transform(test_data)
predictions.select("fare_amount", "prediction").show(5)
```

📌 **Why Use Spark MLlib?**

✓ **Handles large-scale datasets**.

✓ **Distributed processing** makes training faster.

📌 Summary & Next Steps

✅ **Key Takeaways:**

✓ **Apache Spark** efficiently processes **big data**.

✓ **Data Cleaning & Transformation** ensures high-quality input.

✓ **Feature Engineering** improves **predictive modeling**.

✓ **Spark MLlib** enables **distributed machine learning**.

📌 **Next Steps:**

- **Try Spark Streaming for real-time data processing.**
- **Deploy Spark pipelines on AWS, Google Cloud, or Databricks.**
- **Use deep learning models (TensorFlow on Spark) for advanced AI tasks.** 🚀