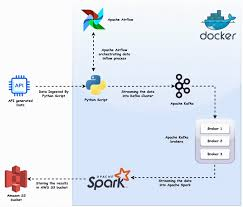
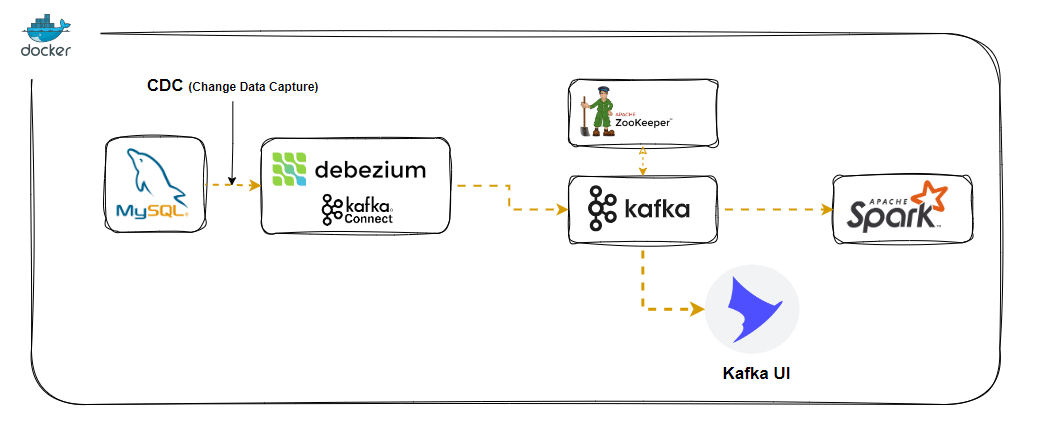
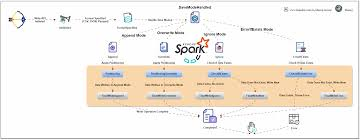
AI and Data-Analysis Task



* **Kafka Basics & Streaming** – Kafka is like a message carrier for big data. One part sends messages (Producer), and another part receives them (Consumer). You are using Kafka to move wind power data between systems.
* **Spark for Data Processing** – Spark is used to clean and analyze big amounts of data. You’re using it to read sensor data, fix timestamps, and calculate useful insights like averages.
* **Machine Learning Basics** – Machine Learning helps computers learn patterns and make predictions. You're building a model that looks at wind speed and other factors to predict how much energy will be generated.
* **Storing Data with Delta Lake** – Delta Lake organizes your data so it’s easy to find and update. You’re saving cleaned sensor data here for later use.
* **Analyzing Wind Power Trends** – You’re studying daily energy patterns, computing averages, and labeling energy production levels (Low, Medium, High, Exceptional).
* **Visualizing Data** – Charts and graphs help to understand how wind power changes over time, showing relationships between wind speed and energy output.
* **Using Kafka for Live Data** – Instead of saving files manually, Kafka moves data from sensors in real time so Spark can process it instantly.

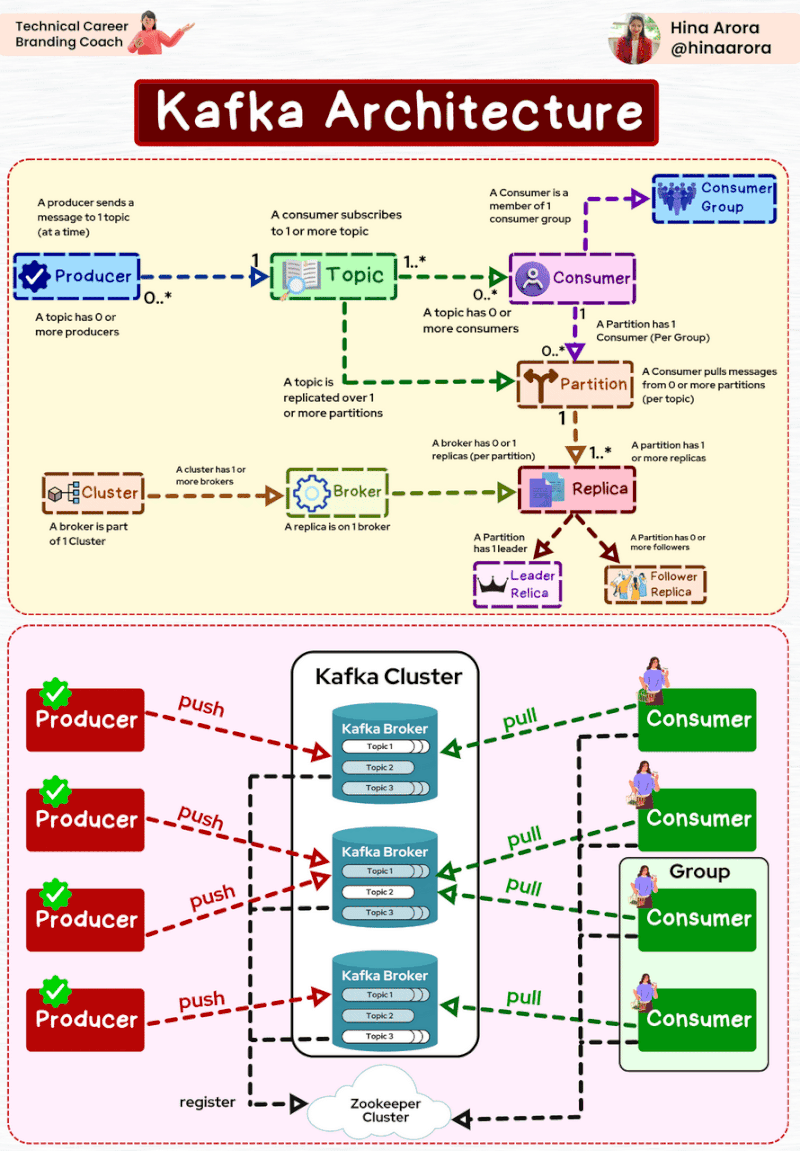
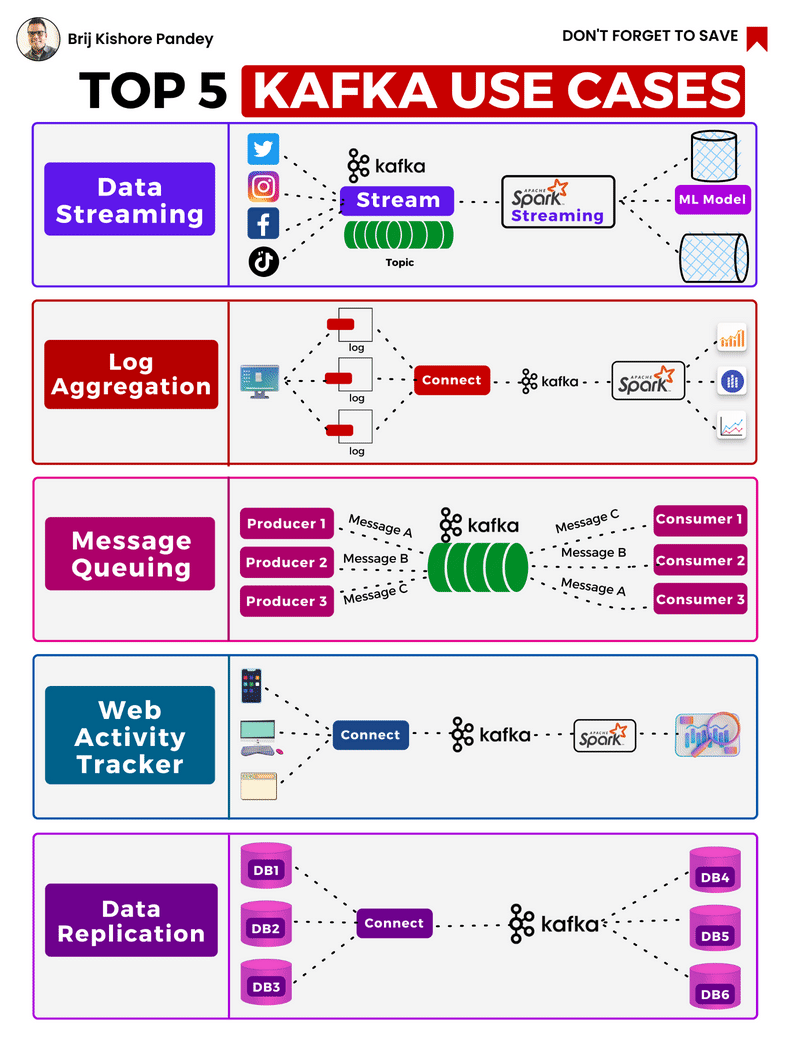
### **Apache Spark Architecture Overview**

Spark is a **distributed computing framework** used for big data processing. Its architecture includes:

* **Driver** – The central controller of Spark applications. It assigns tasks and coordinates execution.
* **Cluster Manager** – Manages computing resources (e.g., YARN, Mesos, Kubernetes, or Spark’s Standalone Cluster).
* **Executors** – Distributed worker nodes that process data in parallel. They store intermediate results in memory for speed.
* **Resilient Distributed Dataset (RDDs)** – The primary data structure Spark uses for processing.
* 

### **Apache Kafka Architecture Overview**

Kafka is a **distributed event-streaming platform** that efficiently handles real-time data processing. Its architecture consists of:

* **Producers** – Applications that send data (messages) to Kafka topics.
* **Topics & Partitions** – Kafka organizes data into topics, which are broken into partitions for scalability.
* **Brokers** – Kafka servers that store and manage data partitions.
* **Consumers** – Applications that **subscribe** to topics and consume messages.
* **Zookeeper** – Manages cluster coordination, leader election, and metadata.
* 
* 

# **Apache Kafka: A Comprehensive Guide**

## **1. Introduction**

Apache Kafka is a distributed event streaming platform used for building real-time data pipelines and streaming applications. It enables efficient handling of large-scale message processing by providing a high-throughput, fault-tolerant system.

### **Use Cases of Kafka**

Kafka is widely used for log aggregation (collecting logs from different sources), event-driven microservices (ensuring reliable communication between microservices), real-time data streaming (processing and analyzing data in real-time), and as a messaging queue replacement (an alternative to traditional message queues like RabbitMQ).

## **2. Kafka Architecture**

Kafka operates as a distributed system, comprising multiple components that work together.

### **2.1 Brokers**

A Kafka cluster contains multiple brokers. Each broker stores and manages messages. Load balancing and fault tolerance are achieved using multiple brokers.

### **2.2 Zookeeper**

Maintains metadata and coordinates leader elections. Keeps track of broker status and cluster management.

### **2.3 Topics and Partitions**

Kafka stores records inside "topics." A topic is divided into partitions to ensure scalability. Partitions allow parallel data processing across multiple consumers.

### **2.4 Producers and Consumers**

Producers push messages into Kafka topics. Consumers fetch and process messages. Kafka guarantees ordered processing within partitions.

## **3. Understanding Kafka Topics and Partitions**

Kafka topics and partitions enable high performance and scalability.

### **3.1 How Partitions Work**

Each partition is an ordered sequence of messages. Messages inside partitions are assigned an offset. Parallelism is achieved as multiple partitions allow distributed processing.

### **3.2 Example: Distributed Log Processing**

Imagine a system collecting logs from multiple servers. Instead of storing logs in a single database, Kafka partitions distribute logs across different nodes.

* Server A sends error logs to logs-topic, Partition 0.
* Server B sends info logs to logs-topic, Partition 1.
* Server C sends debug logs to logs-topic, Partition 2.

Consumers reading from these partitions can process log messages in parallel, increasing efficiency.

## **4. Producing and Consuming Messages Using Console**

Kafka provides command-line utilities for message processing.

### **4.1 Start Kafka Server**

Run Zookeeper: bin/zookeeper-server-start.sh config/zookeeper.properties

Run Kafka broker: bin/kafka-server-start.sh config/server.properties

### **4.2 Create a Kafka Topic**

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

### **4.3 Start a Producer**

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

Type messages and press enter.

### **4.4 Start a Consumer**

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

# **Apache Spark: A Comprehensive Guide**

## **1. Introduction**

Apache Spark is an open-source distributed computing system used for big data processing and analytics. It provides high-speed performance through in-memory computation, making it significantly faster than traditional Hadoop MapReduce. Spark enables batch processing, real-time streaming, SQL-based querying, machine learning, and graph processing within a single unified framework.

### **Use Cases of Spark**

Spark is widely used for large-scale data processing, ETL workflows, real-time streaming analytics, data science, machine learning, and graph computation. Companies leverage Spark for handling huge volumes of data across clusters efficiently.

## **2. Spark Architecture**

Spark follows a master-slave architecture, with key components working together to ensure distributed computation and fault tolerance. The **Driver** is the central orchestrator that manages the execution of Spark applications. It runs the main program and initiates parallel operations by creating the SparkContext. Executors are distributed processes that perform computations on worker nodes. Each executor processes tasks independently and stores the intermediate results in memory for faster computation. Spark supports multiple cluster managers, including Standalone Mode, YARN (integrated with Hadoop), Mesos (resource management across clusters), and Kubernetes (containerized Spark execution). Spark offers multiple abstractions for handling data efficiently: RDD (Resilient Distributed Dataset) is a low-level API used for fault-tolerant, distributed computing. DataFrame is optimized for structured data processing with SQL-like capabilities, and Dataset combines the best features of RDDs and DataFrames, offering both typed and structured processing. Spark builds a DAG (Directed Acyclic Graph) to schedule task execution efficiently. When a Spark job is submitted, it constructs a DAG representing transformations and actions.

## **3. Basic Concepts in Spark**

Apache Spark operates on a lazy evaluation model, meaning transformations are executed only when an action is called. Transformations are operations applied to RDDs/DataFrames/Datasets that result in a new dataset, such as map(), filter(), and groupBy(). Actions are operations that trigger execution and return results, such as count(), collect(), and show(). Spark divides jobs into tasks, which are then distributed across executors. Each executor processes a subset of the data, allowing parallel computation. Spark optimizes execution using a lazy evaluation model to reduce redundant computations.

### **Example: Word Count Program**

A simple Spark application counting words in a text file:

from pyspark.sql import SparkSession spark = SparkSession.builder.appName("WordCount").getOrCreate() text\_rdd = spark.sparkContext.textFile("input.txt") word\_counts = text\_rdd.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b) word\_counts.collect()

## **4. In-Depth Implementation of Spark Concepts**

Spark SQL allows structured data processing using SQL queries. It supports interaction with Hive, JDBC databases, and Parquet files.

### **Example:**

df = spark.read.csv("data.csv", header=True, inferSchema=True) df.createOrReplaceTempView("my\_table") spark.sql("SELECT name, age FROM my\_table WHERE age > 30").show()

Spark Streaming processes real-time data streams using micro-batches.

### **Example:**

from pyspark.streaming import StreamingContext sc = SparkContext("local[\*]", "StreamingExample") ssc = StreamingContext(sc, batchDuration=5) lines = ssc.socketTextStream("localhost", 9999) word\_counts = lines.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b) word\_counts.pprint() ssc.start() ssc.awaitTermination()

Spark MLlib provides machine learning algorithms such as regression, classification, clustering, and collaborative filtering.

### **Example:**

from pyspark.ml.classification import LogisticRegression data = spark.read.format("libsvm").load("data.txt") lr = LogisticRegression() model = lr.fit(data) predictions = model.transform(data) predictions.show()

## **5. Advanced Spark Concepts**

Broadcast Variables are used for distributing large read-only variables across executors efficiently. Example:

broadcast\_var = spark.sparkContext.broadcast([1, 2, 3]) print(broadcast\_var.value)

Accumulators allow aggregation of values across executors. Example:

accum = spark.sparkContext.accumulator(0) def count\_elements(x): global accum accum += x rdd.foreach(count\_elements) print(accum.value)

Efficient data distribution across nodes improves Spark performance. Spark supports Hash Partitioning, which distributes data based on hash values, and Range Partitioning, which splits data into predefined ranges.

Spark ensures fault tolerance using RDD lineage (ability to recompute lost data) and replication mechanisms for recovery.

## **6. Conclusion**

Apache Spark is a powerful distributed computing framework enabling real-time, batch, and machine learning applications. With its high-performance in-memory processing, fault tolerance, and scalability, Spark is widely adopted for big data analytics across industries.

## **5. Advanced Kafka Concepts**

### **5.1 Consumer Groups**

Multiple consumers can share a single topic by forming a consumer group. Kafka automatically balances partitions across consumers.

### **5.2 Message Retention and Replay**

Kafka retains messages for a defined period. Consumers can replay older messages by specifying offsets.

### **5.3 Fault Tolerance and Replication**

Kafka replicates partitions across brokers, ensuring data availability even if a broker fails.

## **6. Conclusion**

Kafka is a powerful tool for real-time data streaming, log processing, and distributed message handling. With fault tolerance, scalability, and parallel processing, it is widely used in modern applications.

# **Apache Spark Made Simple**

## **1. What is Apache Spark?**

Apache Spark is a tool that helps process huge amounts of data quickly. Imagine you have billions of rows of information, and you need to analyze them fast—Spark makes that possible. Unlike older systems that save data to a hard disk after every step, Spark keeps data in memory, making everything much quicker.

### **Where is Spark Used?**

Spark is used in things like:

* **Online Shopping** (for recommending products based on past purchases)
* **Banking** (to detect fraud in transactions)
* **Weather Forecasting** (to analyze climate data)
* **Social Media** (for showing trending topics)

## **2. How Does Spark Work?**

Spark follows a simple system to process data efficiently:

### **2.1 Driver Program**

Think of this as the "boss" of the whole process. It tells workers what tasks to do and manages the data flow.

### **2.2 Executors**

Executors are the "workers" that handle chunks of data, processing it quickly and sending the results back.

### **2.3 Cluster Manager**

This is like a team manager who assigns jobs to different workers. Spark can work with different managers such as:

* **Standalone** (default)
* **YARN** (used with Hadoop)
* **Mesos** (for managing multiple clusters)
* **Kubernetes** (for running tasks in containers)

### **2.4 RDDs, DataFrames, and Datasets**

Spark uses different ways to process data:

* **RDDs (Resilient Distributed Datasets)** – Good for low-level operations but a bit complex.
* **DataFrames** – Easier to use, similar to an Excel sheet.
* **Datasets** – A mix of RDDs and DataFrames, combining the best features of both.

### **2.5 DAG (Directed Acyclic Graph)**

When Spark gets a task, it creates a plan called a DAG, like a roadmap. This roadmap helps Spark avoid unnecessary steps and complete the job faster.

## **3. Basic Spark Concepts**

Spark follows a "lazy execution" style, meaning it doesn’t process data until needed.

### **3.1 Transformations vs. Actions**

* **Transformations** – Making changes to data, such as filtering, sorting, or grouping. Examples: map(), filter(), groupBy().
* **Actions** – Getting results from transformed data. Examples: count(), collect(), show().

### **3.2 How Spark Executes Tasks**

* Data is broken into small pieces and sent to multiple worker nodes.
* Each worker processes a piece of data.
* Spark gathers results and sends them back to the driver program.

### **3.3 Example: Counting Words**

Imagine we want to count words from a text file using Spark:

from pyspark.sql import SparkSession spark = SparkSession.builder.appName("WordCount").getOrCreate() text\_rdd = spark.sparkContext.textFile("input.txt") word\_counts = text\_rdd.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b) word\_counts.collect()

## **4. Using Spark for Different Tasks**

### **4.1 Spark SQL**

Spark lets you run SQL queries to analyze structured data. Example:

df = spark.read.csv("data.csv", header=True, inferSchema=True) df.createOrReplaceTempView("my\_table") spark.sql("SELECT name, age FROM my\_table WHERE age > 30").show()

### **4.2 Spark Streaming**

Spark can process live data from websites or sensors. Example:

from pyspark.streaming import StreamingContext sc = SparkContext("local[\*]", "StreamingExample") ssc = StreamingContext(sc, batchDuration=5) lines = ssc.socketTextStream("localhost", 9999) word\_counts = lines.flatMap(lambda line: line.split(" ")).map(lambda word: (word, 1)).reduceByKey(lambda a, b: a + b) word\_counts.pprint() ssc.start() ssc.awaitTermination()

### **4.3 Spark Machine Learning (MLlib)**

Spark has built-in machine learning features for predictions and recommendations. Example:

from pyspark.ml.classification import LogisticRegression data = spark.read.format("libsvm").load("data.txt") lr = LogisticRegression() model = lr.fit(data) predictions = model.transform(data) predictions.show()

## **5. Advanced Spark Features**

### **5.1 Broadcast Variables**

Used when we need to share data across workers without sending multiple copies. Example:

broadcast\_var = spark.sparkContext.broadcast([1, 2, 3]) print(broadcast\_var.value)

### **5.2 Accumulators**

Used to collect values from multiple worker nodes. Example:

accum = spark.sparkContext.accumulator(0) def count\_elements(x): global accum accum += x rdd.foreach(count\_elements) print(accum.value)

### **5.3 Data Partitioning**

Spark breaks large data sets into small parts (partitions) to improve performance. The two main partitioning methods are:

* **Hash Partitioning** – Splitting data based on hash values.
* **Range Partitioning** – Dividing data into specific ranges.

### **5.4 Fault Tolerance**

Spark prevents data loss using:

* **RDD lineage** (can recreate lost data).
* **Replication** (copies data to prevent failures).

## **6. Conclusion**

Apache Spark is a powerful tool that makes data processing super fast. Whether it's analyzing customer trends, detecting fraud, or training machine learning models, Spark helps businesses handle massive amounts of data efficiently. Its speed, flexibility, and fault tolerance make it one of the most widely used technologies for big data analytics.

Machine Learning (ML) is a way of teaching computers to recognize patterns and make decisions without being directly programmed. Instead of telling the computer exactly what to do, ML allows it to learn from examples and improve over time.

There are **two main types of Machine Learning**: **Supervised Learning** and **Unsupervised Learning**.

### **Supervised Learning**

Supervised learning means we give the computer **examples with answers**. Imagine teaching a kid to recognize fruits. If you show them a picture of an apple and tell them, "This is an apple," they will learn to recognize apples. In the same way, a supervised learning algorithm learns from labeled data.

#### **Where is Supervised Learning Used?**

* **Spam Detection:** Email services analyze past spam emails to learn which emails should be marked as spam.
* **Price Prediction:** Businesses use past sales data to predict future prices.
* **Medical Diagnosis:** AI can detect diseases by analyzing labeled medical reports.

#### **How Supervised Learning Works?**

1. We collect past data (e.g., house prices over time).
2. We train a machine learning model using this data.
3. The model learns patterns and makes predictions on new data.

Imagine you have house prices based on **size** and **location**. If you feed enough examples, the computer can predict prices for houses it hasn’t seen before.

### **Unsupervised Learning**

Unsupervised learning is different because the computer **doesn't get any answers**. It has to **find patterns on its own**. Imagine dumping a big pile of fruits in front of a kid and asking them to group similar ones without knowing their names—that’s unsupervised learning!

#### **Where is Unsupervised Learning Used?**

* **Customer Segmentation:** Online stores group similar shoppers based on buying habits.
* **Fraud Detection:** Banks use ML to detect suspicious transactions.
* **Market Trends:** Companies analyze buying behaviors to recommend new products.

#### **How Unsupervised Learning Works?**

1. The computer is given a lot of data without labels.
2. It looks for similarities and groups similar items together.
3. Companies use these groups to make business decisions.

For example, imagine a clothing store wants to categorize shoppers. Unsupervised learning could **group customers based on what they buy** without needing labels.

### **Differences Between Supervised and Unsupervised Learning**

Supervised learning needs **labeled examples**, while unsupervised learning finds **hidden patterns** in data. Supervised learning **makes predictions**, while unsupervised learning **groups data**.

Machine Learning helps companies automate tasks, make smarter decisions, and improve user experiences. With more data, ML models get better over time.

# **Introduction to Machine Learning**

Machine Learning (ML) is a branch of Artificial Intelligence (AI) that allows computers to learn and improve from experience without being explicitly programmed. Instead of following pre-written instructions, ML models analyze data, identify patterns, and make predictions or decisions automatically.

### **Where is Machine Learning Used?**

ML is used in various fields, such as:

* **Healthcare** (predicting diseases, medical image analysis)
* **Finance** (fraud detection, stock market predictions)
* **E-commerce** (recommendation systems for products)
* **Social Media** (detecting fake accounts, suggesting content)
* **Self-driving Cars** (object detection and navigation)

## **Types of Machine Learning**

### **1. Supervised Learning**

In supervised learning, the model learns from labeled data, meaning it has both input data and the correct output for reference. The model looks at examples and maps relationships between inputs and outputs.

#### **Examples of Supervised Learning:**

* **Spam Detection:** ML models classify emails as spam or not based on past labeled examples.
* **House Price Prediction:** Using past property prices and features (size, location, etc.), the model predicts prices for new homes.
* **Weather Forecasting:** Based on historical weather data, models predict future conditions.

#### **Implementation of Supervised Learning:**

Supervised learning uses algorithms like:

* **Linear Regression** (predicting numerical values)
* **Logistic Regression** (classification problems)
* **Decision Trees** (splitting data based on conditions)
* **Random Forests** (ensemble learning with multiple decision trees)

### **How to Implement Supervised Learning in Python?**

python

from sklearn.model\_selection import train\_test\_split   
from sklearn.linear\_model import LinearRegression   
import pandas as pd   
  
# Load dataset   
data = pd.read\_csv("house\_prices.csv")   
X = data[["size", "location\_index"]] # Features   
y = data["price"] # Target variable   
  
# Split data into training and testing sets   
X\_train, X\_test, y\_train, y\_test = train\_test\_split(X, y, test\_size=0.2, random\_state=42)   
  
# Train model   
model = LinearRegression()   
model.fit(X\_train, y\_train)   
  
# Predict house prices   
predictions = model.predict(X\_test)   
print(predictions)

### **2. Unsupervised Learning**

In unsupervised learning, the model is given unlabeled data and must find patterns or relationships without knowing the correct answers beforehand.

#### **Examples of Unsupervised Learning:**

* **Customer Segmentation:** Grouping customers based on purchasing behavior.
* **Anomaly Detection:** Identifying unusual transactions that might be fraud.
* **Market Basket Analysis:** Discovering which products are often bought together.

#### **Implementation of Unsupervised Learning:**

Unsupervised learning uses algorithms like:

* **K-Means Clustering** (grouping similar data points)
* **Principal Component Analysis (PCA)** (reducing dimensions of data)
* **Hierarchical Clustering** (creating a hierarchy of groups)

### **How to Implement Unsupervised Learning in Python?**

python

from sklearn.cluster import KMeans   
import numpy as np   
  
# Sample data representing customer purchases   
data = np.array([[10, 200], [15, 300], [30, 600], [50, 1000], [75, 1500]])   
  
# Apply K-Means Clustering   
kmeans = KMeans(n\_clusters=2)   
kmeans.fit(data)   
  
# Get cluster labels   
labels = kmeans.labels\_   
print(labels)

## **Key Differences Between Supervised and Unsupervised Learning**

|  |  |  |
| --- | --- | --- |
| **Feature** | **Supervised Learning** | **Unsupervised Learning** |
| Labeled Data | Yes | No |
| Used for Predictions | Yes | No (Pattern Discovery) |
| Example Applications | Spam Detection, House Prices | Customer Segmentation, Fraud Detection |
| Common Algorithms | Linear Regression, Decision Trees | K-Means, PCA |

## **Conclusion**

Machine Learning enables computers to make data-driven decisions and automate tasks. Supervised learning is great for making predictions with labeled data, while unsupervised learning is useful for finding patterns in unlabeled data. With Python libraries like **scikit-learn**, implementing ML models is easier than ever.

Task Implementation:

DockerCompose

### **version: '3.8'**

This specifies the version of Docker Compose being used. Version **3.8** is a stable version that supports modern Docker features.

### **services:**

This section defines the services that will be running in Docker. In this case, **Zookeeper** and **Kafka** are the two services.

### **Zookeeper Service Configuration**

### **zookeeper:**

This is the name of the service for Zookeeper. It helps Kafka manage metadata and leader elections.

### **image: wurstmeister/zookeeper**

This tells Docker to pull and use the **Wurstmeister Zookeeper** image from the Docker registry.

### **container\_name: zookeeper**

This assigns a custom name, zookeeper, to the container instead of using a random name.

### **ports:**

This section **maps ports** between the container and the local machine.

**- "2181:2181"** This means port **2181** inside the container is mapped to **2181** on the host machine. Since Kafka needs to communicate with Zookeeper, it listens on port **2181**.

### **networks:**

This assigns Zookeeper to the custom **network kafka-net**, allowing communication with Kafka.

**- kafka-net** Zookeeper will run inside the Kafka network to ensure proper interaction with other services.

### **Kafka Service Configuration**

### **kafka:**

This is the name of the Kafka service.

### **image: wurstmeister/kafka**

This tells Docker to pull and use the **Wurstmeister Kafka** image from the Docker registry.

### **container\_name: kafka**

This assigns a custom name, kafka, to the container.

### **ports:**

Kafka listens on **port 9092**, which is mapped as follows: **- "9092:9092"** This means **port 9092** inside the container is mapped to **9092** on the host machine, allowing external applications to connect.

### **environment:**

This section defines **environment variables** to configure the Kafka service.

#### **KAFKA\_ADVERTISED\_LISTENERS: PLAINTEXT://localhost:9092**

Kafka tells external applications that it can be reached at localhost:9092. This ensures proper connectivity when apps want to send or receive messages.

#### **KAFKA\_LISTENERS: PLAINTEXT://0.0.0.0:9092**

This makes Kafka listen for incoming connections from **any IP address**, ensuring it accepts traffic from different sources within the network.

#### **KAFKA\_ZOOKEEPER\_CONNECT: zookeeper:2181**

Kafka needs Zookeeper to run properly. This line ensures Kafka connects to Zookeeper at **port 2181** inside the Kafka network.

### **networks:**

Kafka is assigned to the same network as Zookeeper.

**- kafka-net** Kafka runs in kafka-net to ensure communication with Zookeeper and other services.

### **depends\_on:**

This ensures that Kafka only starts **after Zookeeper** is up and running.

**- zookeeper** Kafka waits for Zookeeper to start first, preventing connection issues.

### **Defining Networks**

### **networks:**

This section defines the **custom network** used by Kafka and Zookeeper.

### **kafka-net:**

This is the **network name**, ensuring Kafka and Zookeeper communicate properly.

### **driver: bridge**

This means the network **isolates Kafka and Zookeeper** while allowing them to communicate.

### **Summary**

This Docker Compose file sets up a **Kafka environment** with Zookeeper for metadata management. It ensures proper connectivity between services using a **custom network**, correct port mappings, and essential Kafka configurations. The **depends\_on** setting guarantees Kafka starts only after Zookeeper.

Main.py

Certainly! I'll go through the code **line by line**, explaining what each part does.

### **Importing Required Libraries**

from pyspark.sql import SparkSession

* This imports the SparkSession class, which is required to create and manage a **Spark environment**.

from delta import configure\_spark\_with\_delta\_pip

* This imports the **Delta Lake configuration module**, which helps integrate Delta Lake with Apache Spark.

from pyspark.sql.functions import col, to\_date, to\_timestamp, current\_date, current\_timestamp, array, map\_from\_arrays, lit

* These are **functions** from PySpark used for **data transformation**.
  + col() selects columns from a DataFrame.
  + to\_date() converts a string to a date format.
  + to\_timestamp() converts a string to a timestamp format.
  + current\_date() returns the current system date.
  + current\_timestamp() returns the current system timestamp.
  + array() helps create **arrays of values**.
  + map\_from\_arrays() creates **key-value maps** from two arrays.
  + lit() helps assign static values to new columns.

### **Configuring Spark with Delta**

builder = SparkSession.builder \

* Starts building a **SparkSession** using .builder.

.appName("CSV to Delta Kafka") \

* Names the Spark application "CSV to Delta Kafka", useful for identifying it in logs.

.master("local[\*]") \

* Runs Spark in **local mode**, using all CPU cores ([\*]).

.config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \

* Enables **Delta Lake functionality** in Spark.

.config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")

* Configures Spark to use **Delta Catalog**, which allows Delta operations.

spark = configure\_spark\_with\_delta\_pip(builder).getOrCreate()

* Calls configure\_spark\_with\_delta\_pip(builder) to **apply Delta configurations**, then starts **Spark** using .getOrCreate().

### **Reading CSV Data**

csv\_path = "data/T1.csv"

* Defines the **path** of the CSV file to be read.

df = spark.read.csv(csv\_path, header=True, inferSchema=True)

* Reads the CSV file into a **Spark DataFrame**.
  + header=True ensures that the **first row** is treated as column names.
  + inferSchema=True automatically detects **column data types**.

### **Renaming Columns**

df = df.withColumnRenamed("Date/Time", "DateTime") \

* Renames the "Date/Time" column to "DateTime" to avoid issues with special characters.

.withColumnRenamed("LV ActivePower (kW)", "LV\_ActivePower\_kW") \

* Renames "LV ActivePower (kW)" to "LV\_ActivePower\_kW", making it **compatible** for Delta operations.

.withColumnRenamed("Wind Speed (m/s)", "Wind\_Speed\_m\_s") \

* Renames "Wind Speed (m/s)" to "Wind\_Speed\_m\_s", avoiding spaces and special characters.

.withColumnRenamed("Theoretical\_Power\_Curve (KWh)", "Theoretical\_Power\_Curve\_KWh") \

* Renames "Theoretical\_Power\_Curve (KWh)" to "Theoretical\_Power\_Curve\_KWh".

.withColumnRenamed("Wind Direction (°)", "Wind\_Direction\_deg")

* Renames "Wind Direction (°)" to "Wind\_Direction\_deg", replacing the **degree symbol**.

### **Creating New Columns for Processing**

df\_transformed = df.withColumn("signal\_date", to\_date(col("DateTime"), "yyyy-MM-dd"))

* Creates a new column **"signal\_date"**, extracting only the **date portion** from "DateTime".

.withColumn("signal\_ts", to\_timestamp(col("DateTime"), "yyyy-MM-dd'T'HH:mm:ss"))

* Creates **"signal\_ts"**, converting "DateTime" into a **full timestamp**.

.withColumn("create\_date", current\_date())

* Creates **"create\_date"**, capturing the **current system date**.

.withColumn("create\_ts", current\_timestamp())

* Creates **"create\_ts"**, storing the **exact processing timestamp**.

### **Handling Duplicate Keys in Maps**

spark.conf.set("spark.sql.mapKeyDedupPolicy", "LAST\_WIN")

* Ensures that if **duplicate keys exist in a map**, the **latest value wins**.

### **Defining Key and Value Arrays**

signal\_keys = array(lit("LV\_ActivePower\_kW"), lit("Wind\_Speed\_m\_s"), lit("Theoretical\_Power\_Curve\_KWh"), lit("Wind\_Direction\_deg"))

* Creates an **array of column names** using lit(), treating them as fixed labels.

signal\_values = array(col("LV\_ActivePower\_kW").cast("string"), col("Wind\_Speed\_m\_s").cast("string"), col("Theoretical\_Power\_Curve\_KWh").cast("string"), col("Wind\_Direction\_deg").cast("string"))

* Creates an **array of column values**, converting them to **string format** for consistency.

### **Creating a 'signals' Map Column**

df\_transformed = df\_transformed.withColumn("signals", map\_from\_arrays(signal\_keys, signal\_values))

* Adds a new column **"signals"**, which maps signal\_keys to corresponding signal\_values.

### **Selecting Final Columns**

final\_df = df\_transformed.select("signal\_date", "signal\_ts", "create\_date", "create\_ts", "signals")

* Extracts **only the necessary columns** for storage.

### **Writing Data to Delta Format**

delta\_path = "data/delta\_output/T1\_delta"

* Defines the **location** to store the output in **Delta format**.

final\_df.write.format("delta").option("overwriteSchema", "true").mode("overwrite").save(delta\_path)

* Saves the DataFrame in **Delta format** at delta\_path.
  + "overwriteSchema=true" ensures **schema consistency**.
  + "mode=overwrite" replaces existing data with new data.

### **Writing Data to Parquet Format for Kafka Producer**

parquet\_path = "data/final\_df.parquet"

* Defines the **path** for storing the output in **Parquet format**.

final\_df.write.mode("overwrite").parquet(parquet\_path)

* Saves the processed DataFrame as a **Parquet file**.

### **Printing Success Message**

print(f" Data successfully written to Delta format at {delta\_path} and Parquet at {parquet\_path}")

* Displays a confirmation message showing **where the data was saved**.

### **Summary**

This script reads a **CSV file**, processes the data using **Apache Spark**, and stores the results in **Delta Lake and Parquet format**.

It performs the following key operations:

1. **Loads CSV data** into Spark.
2. **Cleans column names** to prevent errors.
3. **Formats timestamps** for accurate processing.
4. **Creates structured data maps** for key-value storage.
5. **Saves data** in **Delta format (for analytics)** and **Parquet format (for Kafka integration)**.
6. **Handles duplicate values** using Spark's map policies.

Would you like me to further simplify any part of this explanation? Let me know how I can help!

Producer.py

This Python script reads data from a **Parquet file**, formats it, and sends it to **Apache Kafka** for further processing. Let me explain it **line by line** so you can fully understand how it works.

### **Importing Required Libraries**

import json – This imports Python’s built-in json module, which is used for **serializing data** into JSON format before sending it to Kafka.

from kafka import KafkaProducer – This imports the **KafkaProducer class** from the kafka-python library, which allows us to send messages to a Kafka topic.

from pyspark.sql import SparkSession – This imports SparkSession, which is needed to create a **Spark environment** to read the Parquet file.

### **Initializing Spark Session**

spark = SparkSession.builder.appName("Kafka Producer").getOrCreate() – This creates a new **Spark session** with the name "Kafka Producer", allowing us to interact with **Apache Spark**.

### **Reading Parquet File**

parquet\_path = "data/final\_df.parquet" – This defines the **file path** where the Parquet file is stored.

df = spark.read.parquet(parquet\_path) – This **reads the Parquet file** into a Spark DataFrame for further processing.

### **Kafka Configuration**

bootstrap\_servers = 'localhost:9092' – This sets **Kafka’s bootstrap server**, meaning Kafka runs on the **local machine** at localhost:9092.

topic\_name = 'delta\_topic' – This specifies the **Kafka topic name**, where the processed data will be sent.

### **Initializing Kafka Producer**

producer = KafkaProducer(bootstrap\_servers=bootstrap\_servers, value\_serializer=lambda v: json.dumps(v, default=str).encode('utf-8'))

* Creates a **Kafka Producer**, which will send messages to Kafka.
* Uses **JSON serialization** (json.dumps(v, default=str).encode('utf-8')) to ensure proper data formatting before sending.

### **Processing Each Row in DataFrame**

for row in df.collect(): – This **loops through every row** in the DataFrame to prepare and send it to Kafka.

### **Handling Data Formatting**

The script converts each row into a **structured dictionary** (record) before sending it to Kafka:

* "signal\_date": row["signal\_date"].strftime("%Y-%m-%d") if row["signal\_date"] else None
  + Converts signal\_date to the format **YYYY-MM-DD**.
  + Uses .strftime() only if the value is not None, **avoiding errors**.
* "signal\_ts": row["signal\_ts"].strftime("%Y-%m-%dT%H:%M:%S") if row["signal\_ts"] else None
  + Converts signal\_ts to **ISO timestamp format** (YYYY-MM-DDTHH:MM:SS).
* "create\_date": row["create\_date"].strftime("%Y-%m-%d") if row["create\_date"] else None
  + Stores the **creation date** in YYYY-MM-DD format.
* "create\_ts": row["create\_ts"].strftime("%Y-%m-%dT%H:%M:%S") if row["create\_ts"] else None
  + Stores the exact **timestamp when processing happened**.
* "signals": row["signals"]
  + Keeps **sensor values** as key-value pairs from the dataset.

### **Sending Data to Kafka**

producer.send(topic\_name, value=record) – This **sends the processed row** (record) to the Kafka topic "delta\_topic".

### **Flushing Kafka Producer**

producer.flush() – This ensures that all messages are **properly sent to Kafka** before finishing execution.

### **Printing Confirmation Message**

print(f" Data successfully produced to topic '{topic\_name}'") – This confirms **successful data transfer** to Kafka.

### **Summary**

This script:

1. **Reads a Parquet file** using Spark.
2. **Formats the data** to ensure timestamps and missing values are handled correctly.
3. **Initializes a Kafka producer** to send messages.
4. **Loops through the data** and converts each row to JSON format.
5. **Sends processed data** to Kafka in a structured way.
6. **Confirms successful execution** by printing a message.

Would you like further explanations or modifications? Let me know!

Consumer.py

This script sets up a **Kafka Consumer** to read messages from a Kafka topic. I'll explain **line by line** what each part does.

### **Importing Required Libraries**

from kafka import KafkaConsumer

* Imports the KafkaConsumer class from the kafka-python library, which allows receiving messages from Kafka topics.

import json

* Imports Python's built-in json module to **deserialize** messages.

### **Kafka Topic Name Configuration**

topic\_name = "delta\_topic"

* Defines the Kafka **topic name**, ensuring it matches the producer’s topic.
* The producer also sends data to "delta\_topic", so the consumer listens to the same topic.

### **Setting Up Kafka Consumer**

consumer = KafkaConsumer(

* Creates a **Kafka Consumer** to subscribe to the topic and receive messages.

#### **Kafka Consumer Parameters**

topic\_name,

* Specifies the topic to subscribe to.

bootstrap\_servers="localhost:9092",

* Defines the Kafka server location.
* In this case, Kafka runs on localhost:9092.

auto\_offset\_reset="earliest",

* Ensures the consumer starts reading **from the beginning** of the topic.
* "latest" would make it start from the most recent messages instead.

enable\_auto\_commit=True,

* Allows automatic **offset management**, meaning Kafka keeps track of which messages have been processed.

value\_deserializer=lambda x: json.loads(x.decode('utf-8')) if x else None

* Ensures messages are properly **converted from JSON format**.
* It first **decodes UTF-8 bytes** and then **converts them into a Python dictionary** using json.loads().
* If the message is None, it prevents errors.

### **Starting Message Consumption**

print(f" Consuming messages from Kafka topic '{topic\_name}'...")

* Displays a message confirming that the consumer is **ready to receive data**.

### **Processing Incoming Messages**

for message in consumer:

* Iterates **through each message** received from the Kafka topic.

#### **Handling Message Data**

try:

* Starts a **try-except block** to handle potential errors.

if message.value:

* Checks if the message contains **valid data**.

print(f" Consumed record: {json.dumps(message.value, indent=4)}")

* Prints the message in **JSON format**, making it easier to read.

else:

* Handles empty messages.

print("Skipped empty message.")

* If a message contains no data, it skips processing.

### **Error Handling**

except json.JSONDecodeError as e:

* Catches errors that occur if **JSON decoding fails**.

print(f"JSON decoding error: {e}")

* Displays the **error message**, explaining what went wrong.

except Exception as e:

* Catches **any other errors** that might occur during processing.

print(f"Error while processing message: {e}")

* Prints a **general error message** if any unexpected issues arise.

### **Summary**

This script:

1. **Connects to a Kafka topic** (delta\_topic) as a consumer.
2. **Reads messages continuously** from the topic.
3. **Deserializes JSON messages** for easy processing.
4. **Handles errors gracefully**, avoiding crashes.
5. **Prints received data**, making debugging easier.

Would you like any modifications or additional explanations? Let me know how I can help!

A **Data Flow Diagram (DFD)** helps visualize how data moves between different components in a system. Based on the four code files you provided, the **DFD for your Kafka-Spark pipeline** includes **four major processes**:

1. **Data Ingestion (Reading CSV)**
2. **Data Transformation (Processing in Spark)**
3. **Data Publishing (Producing to Kafka)**
4. **Data Consumption (Consuming from Kafka)**

Now, let’s break each step **in easy language** and explain what happens.

### **Step 1: Data Ingestion (Reading CSV)**

**Relevant Code:** First script (main.py)

* You start by reading a CSV file (T1.csv) using **Spark**.
* Spark **automatically detects column names and data types** using inferSchema=True.
* Some column names have **special characters and spaces** that might create issues later.

So, the script **renames them** to make them suitable for **Delta Lake** storage.

💡 **Imagine this step as organizing a messy spreadsheet before working with it.**

### **Step 2: Data Transformation (Processing in Spark)**

**Relevant Code:** Still in main.py

* You create **new columns** to format timestamps (signal\_date, signal\_ts) and add **processing time** (create\_date, create\_ts).
* To store multiple sensor readings properly, you use **maps** (signals column), which store **key-value pairs** (e.g., "Wind\_Speed": 5.6).
* The transformed data is then **saved in Delta format** for structured storage and **Parquet format** for Kafka ingestion.

💡 **Think of this step like preparing the data before putting it into a database—cleaning, formatting, and organizing.**

### **Step 3: Data Publishing (Kafka Producer)**

**Relevant Code:** Producer script (kafka\_producer.py)

* Spark reads the **Parquet file** generated in the previous step.
* **Kafka Producer is initialized** to send messages to Kafka.
* Each row from the Parquet file is converted into a **JSON structure** before being sent.
* The producer **ensures messages are safely sent** and confirms successful execution.

💡 **This step is like sending each row of the spreadsheet to a messaging system so other applications can use it in real-time.**

### **Step 4: Data Consumption (Kafka Consumer)**

**Relevant Code:** Consumer script (kafka\_consumer.py)

* Kafka **continuously listens** to the topic (delta\_topic) and fetches data.
* The consumer **decodes JSON messages** and prints structured data.
* It also has **error handling** to catch issues like empty messages or decoding failures.

💡 **This step is like opening emails one by one as they arrive, checking if they are valid, and reading their contents.**

### **Final Overview (DFD Summary)**

Your system follows **a clear data pipeline**:

1️⃣ **CSV File → Read by Spark** (Step 1)

2️⃣ **Data → Transformed, Cleaned, Stored in Delta/Parquet** (Step 2)

3️⃣ **Processed Data → Sent to Kafka by Producer** (Step 3)

4️⃣ **Kafka → Consumer Reads, Parses, and Displays Data** (Step 4)

Would you like a **graphical representation** of this **Data Flow Diagram (DFD)**? Let me know! 🚀

Apache Spark plays a crucial role in your **Kafka-Spark pipeline** by handling data processing efficiently. Let's break down exactly **where and how Spark is used** in your four scripts:

### **1. Data Ingestion (Reading CSV with Spark)**

In main.py, **Spark** reads a CSV file using:

df = spark.read.csv(csv\_path, header=True, inferSchema=True)

* Spark automatically detects column types and **structures the raw data** into a DataFrame.
* It removes **special characters** in column names to ensure compatibility with Delta Lake.

### **2. Data Transformation (Cleaning & Formatting)**

After ingestion, Spark transforms the data using functions like:

* to\_date(col("DateTime"), "yyyy-MM-dd") → Extracts only the date.
* to\_timestamp(col("DateTime"), "yyyy-MM-dd'T'HH:mm:ss") → Converts to a proper timestamp.
* current\_date() → Stores the **date of processing**.
* map\_from\_arrays(signal\_keys, signal\_values) → Creates a **key-value mapping** of different sensor readings.

Spark ensures that each row follows a **consistent format**, making it ready for further storage and streaming.

### **3. Data Storage (Writing to Delta and Parquet Formats)**

Spark writes the processed data using:

* final\_df.write.format("delta").mode("overwrite").save(delta\_path) → **Stores data in Delta Lake**, ensuring versioning and reliability.
* final\_df.write.mode("overwrite").parquet(parquet\_path) → **Stores data in Parquet format** for efficient Kafka ingestion.

This step is crucial because **structured storage improves query performance** and makes the data accessible to **both batch processing and real-time streaming**.

### **4. Data Publishing (Producing to Kafka)**

In kafka\_producer.py, Spark reads the **Parquet file created earlier**:

df = spark.read.parquet(parquet\_path)

* Spark efficiently loads large datasets from **Parquet**, which is optimized for storage and retrieval.

The producer then **iterates over the DataFrame** using for row in df.collect(), converting each row into JSON before sending it to Kafka.

### **5. Data Consumption (Receiving and Processing Kafka Messages)**

In kafka\_consumer.py, Spark is **not explicitly used**, but it can be integrated for further processing of Kafka messages.

For example, instead of printing messages, **a Spark Streaming job** could be used to process and analyze real-time Kafka events.

### **Key Benefits of Using Spark in This Pipeline**

✅ **High-Speed Processing** – Spark processes large datasets much faster than traditional methods.

✅ **Fault-Tolerance** – With Delta Lake, Spark ensures **data reliability** and supports **reprocessing** in case of failures.

✅ **Scalability** – Spark can scale from **local execution** to **cluster-based distributed processing**.

✅ **Real-Time & Batch Processing** – Spark enables **real-time analytics** (via Kafka integration) and **batch processing** (via Parquet & Delta Lake).

### **Final Summary**

Spark is used for: 1️⃣ **Reading CSV files** and structuring data

2️⃣ **Cleaning, transforming, and formatting data**

3️⃣ **Storing processed data in Delta & Parquet formats**

4️⃣ **Reading structured data for Kafka producers**

5️⃣ **Enabling scalable big-data processing and analytics**

Would you like me to show how **Spark Streaming** could be integrated into your consumer for **real-time analytics**? 🚀

Apache Kafka plays a crucial role in your **data pipeline**, acting as a **real-time message broker** for transferring processed data efficiently. Here's how Kafka is used in your setup:

### **1. Kafka as a Messaging System**

Kafka **connects different components** in your pipeline, ensuring data flows smoothly between **Spark processing** and **external consumers**.

* Instead of saving results in a database directly, you **publish processed data** into a Kafka topic.
* Other applications (or consumers) can then **read data in real time** from Kafka.

### **2. Kafka Producer (Sending Data)**

In kafka\_producer.py:

* Spark reads the **processed Parquet file** containing structured data.
* Kafka **initializes a producer** (KafkaProducer) to send data to a Kafka topic (delta\_topic).
* Each row in the DataFrame is **converted to JSON** and sent to Kafka.
* This allows **multiple consumers** to read structured data **in parallel**.

💡 **Think of Kafka like an instant messaging system—it ensures data moves from one service to another reliably.**

### **3. Kafka Consumer (Receiving Data)**

In kafka\_consumer.py:

* Kafka **continuously listens** to the topic (delta\_topic).
* It **reads incoming JSON messages**, making the data available for further use.
* The consumer prints the **structured data** after safely decoding it.
* Error handling ensures **smooth processing**, even if a message is empty or malformed.

💡 **Imagine a news feed—Kafka lets different systems "subscribe" to incoming data so they can process it in real time.**

### **4. Kafka's Benefits in This Pipeline**

✅ **Decoupling of Services** – Spark processes data **independently**, and Kafka ensures it can be accessed when needed.

✅ **Fault Tolerance** – Even if one consumer fails, **messages remain available** in Kafka.

✅ **Scalability** – Multiple consumers can **read the same data**, enabling distributed processing.

✅ **Real-Time Data Streaming** – Kafka ensures processed sensor data is available **immediately** for further analysis.

### **Final Summary**

Kafka acts as the **middleman**, allowing Spark to **publish processed data** and enabling **real-time data consumption** across different applications.

Would you like me to show how **Kafka Streams** could be used to process data inside Kafka itself? 🚀

Main.py

The script begins by importing necessary libraries including SparkSession, Delta Lake configurations, and PySpark functions used for handling data transformations. It then configures Spark with Delta Lake extensions, ensuring compatibility with Delta tables, and initializes the Spark session for local execution using all available cores.

Next, the script loads a CSV file containing raw sensor data. The first row is treated as headers, and column data types are automatically detected. Since column names contain spaces and special characters, they are renamed for compatibility with Delta Lake storage.

To facilitate structured data processing, new columns are created. DateTime values are converted into date format for easier grouping and timestamp format for precise analysis. Additional columns capture the processing date and timestamp.

The script ensures duplicate keys in mapped structures do not cause conflicts by setting the mapKeyDedupPolicy configuration to keep the latest value. It then creates two arrays, one storing sensor column names and another holding their corresponding values, which are converted into a key-value mapping format.

Distinct signal timestamps per day are calculated by grouping the transformed data by date and counting unique occurrences. Additionally, the script computes the hourly average values for all sensor measurements by extracting the hour from timestamps and calculating mean values.

To classify power generation output, the script introduces a new column named generation\_indicator, which categorizes readings into predefined levels such as low, medium, high, and exceptional based on energy output.

A mapping dictionary is created to standardize sensor names for easier referencing in future analytics. The script dynamically renames signals using an expression that applies these mappings systematically. Only relevant columns are selected for final storage, ensuring that unnecessary data is omitted.

The processed DataFrame is written to Delta Lake and Parquet formats for efficient storage and retrieval. Additionally, the hourly average calculations are separately stored in both formats to support real-time and historical analytics.

Finally, confirmation messages are printed to indicate successful execution, verifying where the data has been stored and computed values recorded. The script is designed to handle sensor readings efficiently, allowing structured analysis and real-time streaming while ensuring reliable storage for later use.

Absolutely! I'll break down the code **line by line** in simpler language so you can understand exactly what is happening.

### **Step 1: Importing Required Libraries**

from pyspark.sql import SparkSession  
from delta import configure\_spark\_with\_delta\_pip  
from pyspark.sql.functions import (  
 col, to\_date, to\_timestamp, current\_date, current\_timestamp,   
 array, map\_from\_arrays, lit, when, avg, countDistinct, date\_format, expr  
)

* **SparkSession** helps create an environment where Spark can run.
* **Delta Library** allows Spark to work with **Delta Lake**, which helps with big data storage.
* **PySpark functions** help in processing data (like formatting dates, creating arrays, calculating averages).

### **Step 2: Setting Up Spark with Delta Lake**

builder = SparkSession.builder \  
 .appName("CSV to Delta Kafka Analysis") \  
 .master("local[\*]") \  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")  
  
spark = configure\_spark\_with\_delta\_pip(builder).getOrCreate()

* **Creates a Spark session** with the name "CSV to Delta Kafka Analysis".
* **Runs Spark locally** using all CPU cores (local[\*]).
* **Adds Delta Lake features** to enable advanced data storage options.

### **Step 3: Loading CSV into Spark**

csv\_path = "data/T1.csv"  
df = spark.read.csv(csv\_path, header=True, inferSchema=True)

* **Loads the CSV file** into a **Spark DataFrame**.
* **Reads column names** from the first row (header=True).
* **Automatically detects** column data types (inferSchema=True).

### **Step 4: Renaming Columns to Remove Special Characters**

df = df.withColumnRenamed("Date/Time", "DateTime") \  
 .withColumnRenamed("LV ActivePower (kW)", "LV\_ActivePower\_kW") \  
 .withColumnRenamed("Wind Speed (m/s)", "Wind\_Speed\_m\_s") \  
 .withColumnRenamed("Theoretical\_Power\_Curve (KWh)", "Theoretical\_Power\_Curve\_KWh") \  
 .withColumnRenamed("Wind Direction (°)", "Wind\_Direction\_deg")

* **Removes spaces and special characters** from column names.
* **Makes them suitable** for storage in **Delta Lake**.

### **Step 5: Creating New Columns for Better Data Organization**

df\_transformed = df.withColumn(  
 "signal\_date", to\_date(col("DateTime"), "dd MM yyyy HH:mm")  
).withColumn(  
 "signal\_ts", to\_timestamp(col("DateTime"), "dd MM yyyy HH:mm")  
).withColumn(  
 "create\_date", current\_date()  
).withColumn(  
 "create\_ts", current\_timestamp()  
)

* Converts "DateTime" to **date format** (signal\_date).
* Converts "DateTime" to **timestamp format** (signal\_ts).
* Adds **current date** (create\_date) when processing happens.
* Adds **current timestamp** (create\_ts) when processing happens.

### **Step 6: Handling Duplicate Keys in Maps**

spark.conf.set("spark.sql.mapKeyDedupPolicy", "LAST\_WIN")

* If two values have the **same key**, Spark **keeps the last one**.

### **Step 7: Creating a Map for Sensor Readings**

signal\_keys = array(  
 lit("LV\_ActivePower\_kW"),  
 lit("Wind\_Speed\_m\_s"),  
 lit("Theoretical\_Power\_Curve\_KWh"),  
 lit("Wind\_Direction\_deg")  
)  
  
signal\_values = array(  
 col("LV\_ActivePower\_kW").cast("string"),  
 col("Wind\_Speed\_m\_s").cast("string"),  
 col("Theoretical\_Power\_Curve\_KWh").cast("string"),  
 col("Wind\_Direction\_deg").cast("string")  
)  
  
df\_transformed = df\_transformed.withColumn(  
 "signals", map\_from\_arrays(signal\_keys, signal\_values)  
)

* Creates an **array of keys** (signal\_keys) representing sensor names.
* Creates an **array of values** (signal\_values) storing actual readings.
* Combines them into a **key-value pair map** (signals).

### **Step 8: Counting Distinct Signal Timestamps Per Day**

distinct\_signal\_ts\_per\_day = df\_transformed.groupBy("signal\_date") \  
 .agg(countDistinct("signal\_ts").alias("distinct\_signal\_ts\_count"))

* Groups data **by date** (signal\_date).
* Counts **distinct timestamps** for each day.

### **Step 9: Calculating Averages Per Hour**

avg\_signals\_per\_hour = df\_transformed.withColumn(  
 "hour", date\_format("signal\_ts", "yyyy-MM-dd HH:00:00")  
).groupBy("hour").agg(  
 avg(col("LV\_ActivePower\_kW")).alias("avg\_LV\_ActivePower\_kW"),  
 avg(col("Wind\_Speed\_m\_s")).alias("avg\_Wind\_Speed\_m\_s"),  
 avg(col("Theoretical\_Power\_Curve\_KWh")).alias("avg\_Theoretical\_Power\_Curve\_KWh"),  
 avg(col("Wind\_Direction\_deg")).alias("avg\_Wind\_Direction\_deg")  
)

* Extracts **hour** from timestamps (signal\_ts).
* Groups data **by hour** and **calculates averages** for all sensors.

### **Step 10: Adding a New Column for Power Generation Status**

df\_transformed = df\_transformed.withColumn(  
 "generation\_indicator",  
 when(col("LV\_ActivePower\_kW") < 200, "Low")  
 .when((col("LV\_ActivePower\_kW") >= 200) & (col("LV\_ActivePower\_kW") < 600), "Medium")  
 .when((col("LV\_ActivePower\_kW") >= 600) & (col("LV\_ActivePower\_kW") < 1000), "High")  
 .otherwise("Exceptional")  
)

* Categorizes **power generation levels** (Low, Medium, High, Exceptional) based on **LV Active Power**.

### **Step 11: Mapping Signal Names to Standard Names**

signal\_mapping\_dict = {  
 "LV\_ActivePower\_kW": "active\_power\_average",  
 "Wind\_Speed\_m\_s": "wind\_speed\_average",  
 "Theoretical\_Power\_Curve\_KWh": "theo\_power\_curve\_average",  
 "Wind\_Direction\_deg": "wind\_direction\_average"  
}

* Creates **mapping** for sensor names to **standard names**.

df\_transformed = df\_transformed.withColumn(  
 "signals", expr(map\_expr)  
)

* Dynamically **updates signal names** to match **standard mappings**.

### **Step 12: Selecting Final Columns**

final\_df = df\_transformed.select(  
 "signal\_date",  
 "signal\_ts",  
 "create\_date",  
 "create\_ts",  
 "signals",  
 "generation\_indicator"  
)

* Keeps **only relevant columns** for storage.

### **Step 13: Writing Data to Delta & Parquet Formats**

delta\_path = "data/delta\_output/T1\_delta"  
final\_df.write.format("delta").option("overwriteSchema", "true").mode("overwrite").save(delta\_path)  
  
parquet\_path = "data/final\_df.parquet"  
final\_df.write.mode("overwrite").parquet(parquet\_path)

* Saves data in **Delta format** (delta\_path) for structured storage.
* Saves data in **Parquet format** (parquet\_path) for efficient use with Kafka.

### **Step 14: Writing Average Sensor Data to Delta & Parquet**

avg\_signals\_per\_hour.write.format("delta").option("overwriteSchema", "true").mode("overwrite").save(avg\_delta\_path)  
avg\_signals\_per\_hour.write.mode("overwrite").parquet(avg\_parquet\_path)

* Stores **hourly averages** in **Delta** and **Parquet** for analysis.

### **Final Summary**

✅ **Reads sensor data from CSV**

✅ **Formats timestamps & renames columns**

✅ **Categorizes power generation levels**

✅ **Stores structured data in Delta Lake & Parquet**

✅ **Prepares data for real-time Kafka streaming**

Would you like me to suggest **how to integrate this with Kafka Streaming** for real-time analysis? 🚀

Ml.py

This script is designed to **read data from a Delta table using Apache Spark**, making it ready for further processing, including machine learning tasks. I'll explain each line in **simple terms**.

**from pyspark.sql import SparkSession** – Imports SparkSession, which is required to interact with Spark and manage distributed computing.

**from delta import configure\_spark\_with\_delta\_pip** – Imports a helper function that configures Spark to work with **Delta Lake**, which improves data management.

**builder = SparkSession.builder \** – Creates a **SparkSession builder**, which allows setting configurations for how Spark should run.

**.appName("Delta Data for ML") \** – Names the Spark application "Delta Data for ML", which makes tracking and logging easier.

**.master("local[\*]") \** – Runs Spark **locally**, using all available CPU cores ([\*]), meaning it won’t run on a cluster.

**.config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \** – Enables **Delta Lake features**, so Spark can work with Delta tables for efficient storage and querying.

**.config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")** – Configures Spark to use **Delta Catalog**, allowing it to interact with Delta tables like a database.

**spark = configure\_spark\_with\_delta\_pip(builder).getOrCreate()** – Applies the Delta configuration and **creates** the SparkSession, making Spark ready to process data.

**delta\_path = "data/delta\_output/T1\_delta"** – Defines the **file path** where the Delta table is stored, so Spark knows where to find it.

**df = spark.read.format("delta").load(delta\_path)** – **Loads the Delta table** from the specified path into a **Spark DataFrame**, making the data ready for analysis.

**df.show(5)** – Displays the **first five rows** of the DataFrame, letting the user see a preview of the data.

This setup makes it easy to **read structured data stored in Delta format**, preparing it for machine learning tasks or further transformations. Let me know if you need modifications or extra details!

Ml.pynb

This script integrates **Apache Spark, Delta Lake, Machine Learning, and Data Visualization** to process sensor data, analyze correlations, and make predictions using **Linear Regression**. Here's a detailed breakdown of what each section does:

**Stopping an Existing Spark Context:** Before starting a new Spark session, the script checks if an active SparkContext exists. If found, it stops it to prevent conflicts.

**Configuring Spark Session:** A new Spark session is created with specific settings, ensuring smooth execution. The session enables Delta Lake integration, allowing structured data operations.

**Validating Delta Table Path:** The script checks if the path to the Delta table exists before loading data, preventing errors from missing files.

**Loading Data from Delta Table:** If the path is valid, Spark reads data stored in Delta format, making it available for further analysis.

**Extracting a Specific Column:** The script selects the avg\_Wind\_Speed\_m\_s column, ensuring data exists before converting it into a JSON-formatted output.

**Converting Data to Pandas for Visualization:** The Spark DataFrame is transformed into a Pandas DataFrame, making it easier to use libraries like **Matplotlib** and **Seaborn**.

**Generating a Correlation Matrix:** Seaborn is used to visualize relationships between different sensor readings, highlighting how variables interact.

**Creating Histograms of Sensor Readings:** Histograms display the distributions of power generation and wind speed to understand variations in data.

**Generating Scatter Plots:** Scatter plots visualize relationships between factors like wind speed and power output, aiding pattern recognition.

**Checking Data Statistics:** The script prints summary statistics and schema details, ensuring data quality checks.

**Identifying Null Values:** A function calculates the number of missing values per column, ensuring data completeness for ML models.

**Setting Up a Machine Learning Model:** It prepares **Linear Regression** by assembling selected features into a structured format for ML training.

**Splitting Data for Training and Testing:** The dataset is divided into 80% training and 20% testing to evaluate model performance.

**Training the Model:** A linear regression model learns from training data to predict **power output** based on **wind speed and direction**.

**Evaluating Model Performance:** Metrics like **Root Mean Squared Error (RMSE)** and **R2 score** measure how well predictions align with real data.

**Predicting on New Data:** A single row of manually created sensor data is transformed and fed into the model to generate a power output prediction.

### **Data Flow Diagram (DFD) Overview**

1️⃣ **Data Ingestion:** Read from Delta Lake and convert to Pandas for visualization.

2️⃣ **Data Analysis:** Compute correlations, generate plots, and check missing values.

3️⃣ **Model Training:** Prepare data, build a linear regression model, and evaluate accuracy.

4️⃣ **Prediction:** Test the model with real-world values and generate predictions.

Would you like an actual **graphical DFD representation** of this workflow? 🚀

Certainly! I'll explain each line of code in **simple terms**, so you fully understand how it works.

### **1. Importing Necessary Libraries**

from pyspark.sql import SparkSession

This imports **SparkSession**, which is required to create a Spark environment for **processing large datasets**.

from delta import configure\_spark\_with\_delta\_pip

This imports a **Delta Lake configuration tool**, allowing Spark to interact with Delta tables efficiently.

import os

This imports the **OS module**, which helps in handling file paths and checking if certain files exist.

### **2. Stopping Existing Spark Context**

if SparkContext.\_active\_spark\_context is not None:  
 print("Stopping active SparkContext...")  
 SparkContext.getOrCreate().stop()

This **checks** if a Spark session is already running.

* If an active Spark context is found, it **stops the existing session** to prevent conflicts when starting a new one.

### **3. Creating a Spark Session**

builder = SparkSession.builder \  
 .appName("Delta Data for ML") \  
 .master("local[\*]") \  
 .config("spark.driver.port", "4051") \  
 .config("spark.driver.bindAddress", "127.0.0.1") \  
 .config("spark.driver.host", "127.0.0.1") \  
 .config("spark.sql.extensions", "io.delta.sql.DeltaSparkSessionExtension") \  
 .config("spark.sql.catalog.spark\_catalog", "org.apache.spark.sql.delta.catalog.DeltaCatalog")

* **Creates a Spark session** named "Delta Data for ML" for handling data processing tasks.
* **Runs locally (local[\*])**, meaning all CPU cores on your machine are used.
* **Configures Spark driver settings** to bind to localhost (127.0.0.1) and set a port (4051) for communication.
* **Enables Delta Lake extensions** so Spark can read and write Delta tables.

spark = configure\_spark\_with\_delta\_pip(builder).getOrCreate()

This applies the **Delta Lake configurations** and **creates the Spark session**.

### **4. Checking if Delta Table Path Exists**

delta\_path = "data/delta\_output/avg\_signals\_per\_hour\_delta"  
if not os.path.exists(delta\_path):  
 print(f"Error: The path {delta\_path} does not exist.")

* **Defines the path** where the Delta table is stored.
* **Checks if the folder/file exists** before loading data to prevent errors.

### **5. Loading Data from Delta Table**

df = spark.read.format("delta").load(delta\_path)  
df.show(5)

* **Reads Delta Lake table** from the specified path.
* **Displays the first five rows** using .show(5), helping users see a preview of the data.

### **6. Extracting a Specific Column**

row\_data = df.select('avg\_Wind\_Speed\_m\_s').limit(1).collect()

* **Selects one row** from the column "avg\_Wind\_Speed\_m\_s" and **collects it** for processing.

if row\_data:  
 data\_value = row\_data[0]['avg\_Wind\_Speed\_m\_s']  
 data\_json = json.dumps(data\_value, indent=2)  
 print(data\_json)  
else:  
 print("No data found in the 'avg\_Wind\_Speed\_m\_s' column.")

* **Checks if data exists**, preventing errors.
* **Converts the extracted value** to a JSON-formatted output.
* **Prints the result**, ensuring visibility.

### **7. Converting DataFrame to Pandas for Visualization**

import pandas as pd  
import matplotlib.pyplot as plt  
pandas\_df = df.toPandas()  
print(pandas\_df.head())

* **Imports Pandas and Matplotlib**, essential for data analysis and visualization.
* **Converts the Spark DataFrame into a Pandas DataFrame**, making it easier to work with.
* **Displays the first five rows** of the Pandas DataFrame.

### **8. Generating Correlation Matrix**

import seaborn as sns  
correlation\_matrix = pandas\_df.iloc[:,1:].corr()

* **Imports Seaborn**, a Python visualization library.
* **Computes the correlation matrix**, showing how sensor values relate to each other.

plt.figure(figsize=(8, 6))  
sns.heatmap(correlation\_matrix, annot=True, cmap="coolwarm", fmt=".2f")  
plt.title("Correlation Matrix between Signals")  
plt.show()

* **Plots the correlation matrix using a heatmap**, highlighting relationships between variables.

### **9. Creating Histograms of Sensor Readings**

columns\_to\_plot = [  
 'avg\_LV\_ActivePower\_kW',  
 'avg\_Wind\_Speed\_m\_s',  
 'avg\_Theoretical\_Power\_Curve\_KWh',  
 'avg\_Wind\_Direction\_deg'  
]  
fig, axes = plt.subplots(2, 2, figsize=(12, 10))  
axes = axes.flatten()

* **Defines sensor columns to visualize**.
* **Creates a 2x2 grid** of subplots for multiple histogram plots.

for i, col in enumerate(columns\_to\_plot):  
 sns.histplot(data=pandas\_df, x=col, ax=axes[i], kde=True, color='skyblue')  
 axes[i].set\_title(f"Distribution of {col}")  
 axes[i].set\_xlabel(col)  
 axes[i].set\_ylabel('Count')

* **Loops through each column**, plotting a histogram to visualize distributions.

### **10. Creating Scatter Plots**

fig, axes = plt.subplots(2, 2, figsize=(12, 10))  
axes = axes.flatten()  
axes[0].scatter(pandas\_df['avg\_Wind\_Speed\_m\_s'], pandas\_df['avg\_LV\_ActivePower\_kW'], color='blue', alpha=0.5)  
axes[0].set\_title('Wind Speed vs Active Power')

* **Creates scatter plots** to analyze relationships between different sensors.
* **Example: Wind speed vs power generation**.

### **11. Checking Data Statistics**

df.describe().show()  
df.printSchema()

* **Shows summary statistics**, such as min, max, mean, and standard deviation.
* **Prints the schema**, displaying data types.

from pyspark.sql.functions import col, sum  
null\_counts = df.select([sum(col(c).isNull().cast("int")).alias(c) for c in df.columns]).show()

* **Counts missing values** in each column.

### **12. Setting Up a Machine Learning Model**

from pyspark.ml.feature import VectorAssembler  
from pyspark.ml.regression import LinearRegression

* **Imports Spark ML components** required for machine learning.

feature\_cols = ["avg\_Wind\_Speed\_m\_s", "avg\_Theoretical\_Power\_Curve\_KWh", "avg\_Wind\_Direction\_deg"]  
assembler = VectorAssembler(inputCols=feature\_cols, outputCol="features")  
df\_transformed = assembler.transform(df).select("features", "avg\_LV\_ActivePower\_kW")

* **Combines features into a single vector**, preparing data for ML training.

train\_data, test\_data = df\_transformed.randomSplit([0.8, 0.2], seed=42)

* **Splits data into training (80%) and testing (20%) sets**.

lr = LinearRegression(labelCol="avg\_LV\_ActivePower\_kW", featuresCol="features", maxIter=10)  
lr\_model = lr.fit(train\_data)

* **Creates and trains the Linear Regression model**.

predictions = lr\_model.transform(test\_data)  
predictions.select("features", "avg\_LV\_ActivePower\_kW", "prediction").show()

* **Tests the trained model on new data**.

### **13. Making Predictions on New Sensor Data**

single\_row\_data = Row(avg\_Wind\_Speed\_m\_s=4.3, avg\_Theoretical\_Power\_Curve\_KWh=206.0, avg\_Wind\_Direction\_deg=195.0)  
single\_row\_df = spark.createDataFrame([single\_row\_data])

* **Creates a single row with wind speed, direction, and theoretical power**.

single\_row\_transformed = assembler.transform(single\_row\_df).select("features")  
single\_row\_prediction = lr\_model.transform(single\_row\_transformed)  
single\_row\_prediction.select("features", "prediction").show()

* **Transforms the single row** and **predicts its power output**.

Let me know if you'd like a **graphical DFD representation** of this process! 🚀

I can provide a detailed **data flow diagram (DFD)** representation for your pipeline. However, I am unable to directly generate graphical diagrams. Instead, I can describe the structure, helping you visualize it or create it using tools like **draw.io**, **Microsoft Visio**, or **Lucidchart**.

### **DFD Structure for Your Workflow**

Your workflow involves **data ingestion, processing, storage, analysis, and machine learning**. The data flow consists of the following components:

1️⃣ **Data Source**

* CSV File (Raw Sensor Data)
* Existing Delta Table

2️⃣ **Data Processing with Spark**

* Reads data from **CSV** and **Delta Table**
* Cleans data (renaming columns, handling missing values)
* Converts timestamps into structured formats
* Stores processed data in **Delta Lake**

3️⃣ **Data Analysis and Visualization**

* Converts Spark DataFrame to Pandas for easy analysis
* Computes **correlations** between sensor readings
* Generates **histograms** and **scatter plots** to explore relationships

4️⃣ **Machine Learning Model Training**

* Prepares data (VectorAssembler to structure features)
* Splits data (training: 80%, testing: 20%)
* Fits **Linear Regression model** to predict **active power output**

5️⃣ **Model Evaluation and Prediction**

* Evaluates model using **RMSE and R² score**
* Tests model on **real-world sensor data**
* Generates **predictions for wind speed and power**

### **Graphical Representation (How to Draw It Yourself)**

You can create a **DFD diagram** using tools like **draw.io**, **Microsoft Visio**, or **Lucidchart** by following this structure:

* **External Entity:** CSV File, Delta Lake
* **Processes:** Data Cleaning, Feature Engineering, Model Training, Model Prediction
* **Data Stores:** Delta Table, Parquet Files
* **Flows:** Arrows connecting each component

Would you like me to format this into a **text-based ASCII diagram** to help you visualize it better? 🚀

### **Overview of What’s Happening**

Your files **perform data processing, transformation, storage, analysis, visualization, and machine learning predictions** using Apache Spark and Delta Lake.

#### **Step 1: Data Ingestion & Processing (First File)**

* Reads data from a **Delta Table**.
* Cleans and transforms the **sensor data**.
* Computes **average values** and **distinct timestamps** for sensor readings.
* Saves processed data into **Delta Lake & Parquet files**.

#### **Step 2: Data Analysis & Visualization (Second File)**

* Loads structured sensor data into Pandas.
* Computes **correlations** between power generation and wind speed.
* Generates **histograms & scatter plots** to visualize sensor patterns.
* Checks missing values in the dataset.
* Summarizes statistics of the data.

#### **Step 3: Machine Learning Predictions (Third File)**

* Prepares data for **Linear Regression training**.
* Splits the dataset into **training (80%)** and **testing (20%)**.
* Trains a **predictive model** using wind speed and direction.
* Evaluates model performance using **RMSE and R² metrics**.
* Tests the model with **real-world sensor values** to predict power output.

### **DFD Structure for Your Workflow**

Your workflow follows a structured **data pipeline**:

1️⃣ **Data Source**

* Delta Table (Preprocessed Sensor Data)
* Raw CSV Files

2️⃣ **Data Processing with Spark**

* Cleans raw sensor data
* Formats timestamps for consistency
* Computes averages & distinct timestamps
* Stores structured data in **Delta Lake**

3️⃣ **Data Analysis and Visualization**

* Converts Spark DataFrame to Pandas
* Computes **correlations between variables**
* Generates **histograms & scatter plots**
* Detects missing values in dataset

4️⃣ **Machine Learning Model Training & Prediction**

* Prepares data for **feature engineering**
* Trains **Linear Regression model**
* Evaluates model accuracy using **test data**
* Predicts **power output based on wind speed & direction**

Here’s a **high-level summary** of what’s happening across these seven files:

1️⃣ **Data Ingestion & Processing:** Spark reads raw data from CSV files and Delta tables. It cleans, transforms, and stores structured data. 2️⃣ **Delta Lake Storage:** Processed data is saved in **Delta format** for efficient querying and historical tracking. 3️⃣ **Data Publishing & Streaming with Kafka:** Kafka producers send structured data, and Kafka consumers receive and process messages. 4️⃣ **Data Analysis & Visualization:** Using Pandas and Matplotlib, the data is explored, correlations are computed, and sensor trends are plotted. 5️⃣ **Machine Learning Model Training:** A **Linear Regression model** is built to predict power output based on wind speed and direction. 6️⃣ **Model Evaluation & Predictions:** The model is tested on real-world sensor data, and predictions are made for energy generation. 7️⃣ **Real-Time Data Integration:** Kafka, Spark, and Delta Lake work together to enable real-time analytics and AI-driven insights.

Yes! Based on your task breakdown, you are executing **each stage correctly**. Below, I'll provide a **detailed explanation** of what is happening and **a Data Flow Diagram (DFD)** representation for your workflow.

### **Understanding What Is Happening in Each Task**

Your project follows a structured **data pipeline**, integrating **Kafka, Spark, Delta Lake, and Machine Learning** to process and analyze wind power generation data.

#### **1️⃣ Kafka Producer (Data Ingestion & Streaming)**

* Spark **reads sensor data** from a **CSV file**.
* Kafka **publishes the data** in a **streaming fashion**, meaning records are sent continuously.
* Kafka acts as a **message broker**, ensuring reliable transmission.

#### **2️⃣ Kafka Consumer (Receiving & Storing Data in Delta Lake)**

* Kafka **streams data continuously**, which is **received** by Spark.
* The data is **formatted into Delta Lake** (with structured schema for timestamps and signal mapping).
* **Delta tables** ensure data **versioning and efficient querying**.

#### **3️⃣ Analysis Task (Data Processing & Computation)**

* **Reads data** from Delta Lake using Spark.
* **Calculates unique timestamps per day** (144 expected based on 10-minute intervals).
* **Computes hourly averages** for all sensor measurements.
* **Adds a classification column** (generation\_indicator) based on **power output** levels.

#### **4️⃣ Signal Name Mapping**

* A **JSON structure** defines **standardized names** for each signal.
* A **broadcast join** updates the dataset with **consistent signal names**.

#### **5️⃣ Machine Learning Model (Prediction & Training)**

* **Prepares data** using feature selection.
* **Trains a Linear Regression model** to **predict power output**.
* **Evaluates the model** using **RMSE and R² metrics**.
* **Uses real-world sensor values** to predict **active power**.

### **Data Flow Diagram (DFD Representation)**

Your system follows a structured **data pipeline** connecting different components:

#### **Level 1: Overview of Data Flow**

1️⃣ **Data Sources**

* CSV Files (Raw Sensor Data)
* Delta Table (Processed Data)

2️⃣ **Kafka Streaming Process**

* Kafka Producer reads from **CSV and streams data**
* Kafka Consumer receives **streamed data** and **writes to Delta Lake**

3️⃣ **Data Processing with Spark**

* Reads data from **Delta Lake**
* Computes **distinct timestamps per day**
* Calculates **hourly averages**
* Classifies **power generation levels**

4️⃣ **Machine Learning Model Training**

* Prepares data using **feature engineering**
* Trains **Linear Regression model**
* Evaluates performance using **test data**
* Predicts **power output based on wind speed & direction**

Would you like me to **draw a text-based DFD diagram** that you can use as a reference? 🚀 Let me know how you'd like to proceed!