Comprehensive Wind Turbine Data Pipeline Using Databricks & Medallion Architecture

# Introduction

This document outlines a detailed and scalable data pipeline design implemented using the Databricks platform based on the Medallion Architecture. The solution ingests raw data from wind turbines, cleanses and processes it to extract actionable insights, and facilitates real-time reporting.

# Problem Statement

Wind turbines generate large volumes of sensor data daily, which often includes missing values, outliers, and sensor inconsistencies. Engineers and stakeholders require accurate, cleaned, and reliable data to monitor performance and identify malfunctioning units. This necessitates an automated pipeline that can handle ingestion, transformation, anomaly detection, and data delivery efficiently.

# Architectural Overview

The pipeline follows a Medallion Architecture pattern comprising three core layers: Bronze, Silver, and Gold. Each layer has a specific role in improving data quality and usability in progressive stages.

• Bronze Layer: Raw ingestion from CSVs, capturing data without transformations.

• Silver Layer: Cleansing and calculation of daily turbine statistics (min, max, avg).

• Gold Layer: Detection of anomalies based on statistical deviation from the mean.

# Bronze Layer – Raw Data Ingestion

This stage uses Databricks' Auto Loader or direct Spark CSV reads to load raw turbine data stored in Azure Blob Storage. The data is truncated and overwritten daily to ensure freshness and consistency. A 'file\_source' column is added to help trace the data origin.

Code:

from pyspark.sql.functions import input\_file\_name  
  
raw\_df = (  
 spark.read  
 .option("header", True)  
 .csv("/mnt/raw-data/")  
 .withColumn("file\_source", input\_file\_name())  
)  
  
raw\_df.write.mode("overwrite").saveAsTable("bronze\_raw\_turbine\_data")

# Silver Layer – Data Cleaning and Statistics

This layer transforms the raw data into meaningful cleaned data. It removes rows with missing values and filters outliers using Z-score logic (values beyond 3 standard deviations from the mean). After cleansing, the pipeline computes daily summary statistics for each turbine.

Cleaning Code:

from pyspark.sql.functions import col, avg, stddev, abs as abs\_  
  
# Remove nulls  
no\_nulls\_df = raw\_df.dropna(subset=["power\_output", "wind\_speed"])  
  
# Calculate stats for Z-Score cleaning  
stats\_df = no\_nulls\_df.groupBy("turbine\_id").agg(  
 avg("power\_output").alias("mean\_output"),  
 stddev("power\_output").alias("std\_output")  
)  
  
# Join and filter out outliers  
cleaned\_df = no\_nulls\_df.join(stats\_df, on="turbine\_id") \ .filter(abs\_(col("power\_output") - col("mean\_output")) <= 3 \* col("std\_output")) \ .select("timestamp", "turbine\_id", "wind\_speed", "wind\_direction", "power\_output")  
  
cleaned\_df.write.mode("overwrite").saveAsTable("silver\_cleaned\_turbine\_data")

Statistics Code:

from pyspark.sql.functions import to\_date, min, max, count  
  
daily\_stats\_df = cleaned\_df.withColumn("reading\_date", to\_date("timestamp")) \ .groupBy("reading\_date", "turbine\_id") \ .agg(  
 min("power\_output").alias("min\_output"),  
 max("power\_output").alias("max\_output"),  
 avg("power\_output").alias("avg\_output"),  
 count("power\_output").alias("record\_count")  
 )  
  
daily\_stats\_df.write.mode("overwrite").saveAsTable("silver\_turbine\_daily\_stats")

# Gold Layer – Anomaly Detection

In this layer, the pipeline identifies abnormal turbine behavior by comparing each turbine's power output to its average, flagging records where the deviation exceeds two standard deviations. This identifies malfunctioning or underperforming turbines proactively.

Anomaly Detection Code:

# Calculate mean and std dev for anomaly detection  
anomaly\_stats\_df = cleaned\_df.withColumn("reading\_date", to\_date("timestamp")) \ .groupBy("reading\_date", "turbine\_id") \ .agg(  
 avg("power\_output").alias("mean\_output"),  
 stddev("power\_output").alias("std\_output")  
 )  
  
# Join with cleaned data and detect anomalies  
anomalies\_df = cleaned\_df.withColumn("reading\_date", to\_date("timestamp")) \ .join(anomaly\_stats\_df, ["reading\_date", "turbine\_id"]) \ .withColumn("is\_anomaly",  
 (abs\_(col("power\_output") - col("mean\_output")) > 2 \* col("std\_output")).cast("boolean"))  
  
anomalies\_df.select(  
 "reading\_date", "turbine\_id", "power\_output", "mean\_output", "std\_output", "is\_anomaly"  
).write.mode("overwrite").saveAsTable("gold\_turbine\_anomalies")

# Monitoring and Alerts

The solution uses Databricks Job UI and alerting features to monitor the health of the pipeline. Custom alerts can be configured for unusual values like zero output or frequent anomalies. Logs and diagnostics are stored to help with root-cause analysis.

# Reporting and Dashboarding

The processed data is made available to Power BI using Delta Sharing or through Azure SQL. Business users can use interactive dashboards to view daily performance summaries, anomaly logs, and historical trends.

# Why This Approach Works – Scalability and Benefits

This architecture separates data concerns across layers, ensuring a clean lineage and better management. It’s fundamentally robust because:

- Data issues are handled close to the source (in the Silver Layer).

- Performance optimizations are possible through Delta Lake and partitioning.

- It supports large-scale processing using Spark and can handle petabytes of data on Azure Blob Storage.

- Business-ready insights are produced automatically and reliably.