ENPPI Climate IoT Data Engineering Project

Overview

The ENPPI Climate IoT Data Engineering Project is designed to monitor, collect, and analyze environmental data from IoT sensors deployed across multiple industrial facilities. This system provides a unified and fully automated platform for both real-time insights and historical analysis, leveraging cloud-native technologies and modern data engineering best practices. By treating data as a product, the platform delivers high-quality, actionable information for decision-makers, including climate engineers, mechanical engineers, and plant managers.

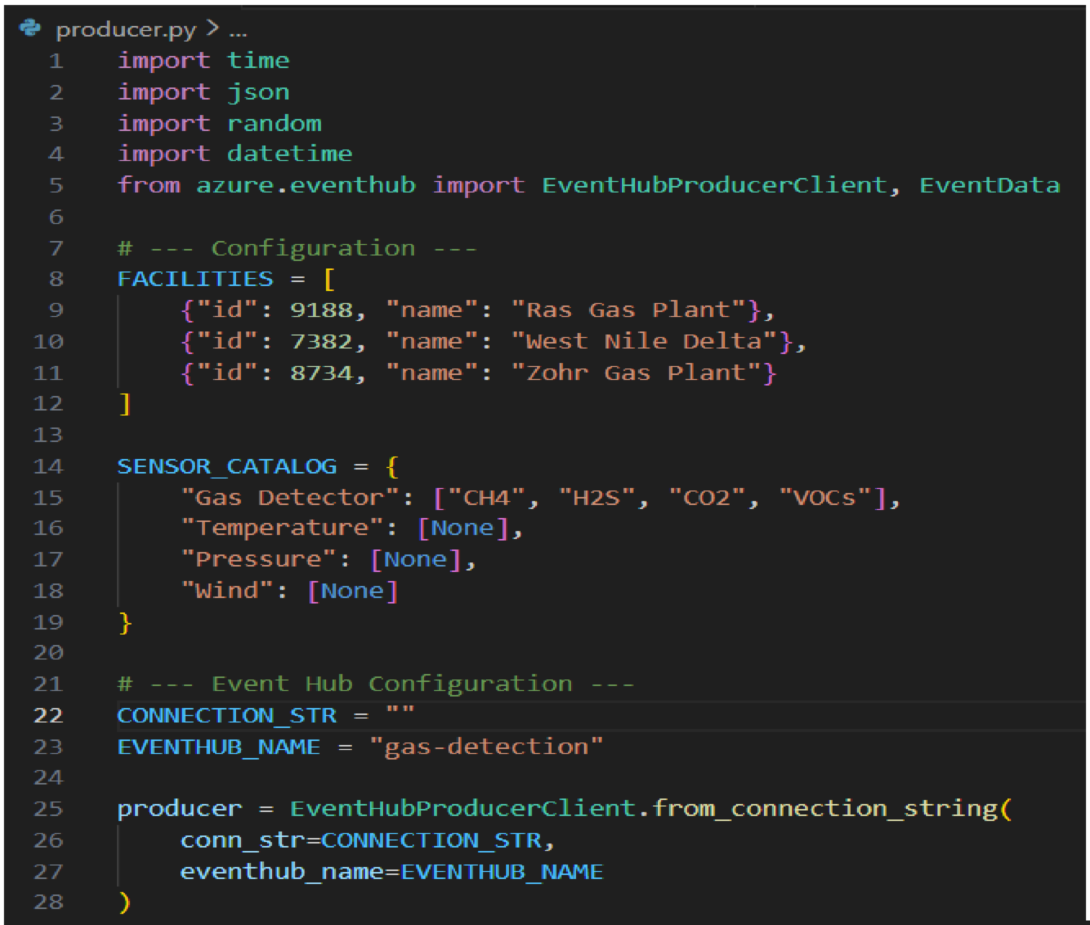
The system supports multiple objectives: detecting environmental risks, monitoring equipment health, optimizing operational efficiency, and enabling predictive analytics. The integration of an AI-driven conversational agent allows non-technical personnel to interact with data in plain language, making complex analytical results accessible to a wide range of stakeholders.

Architecture

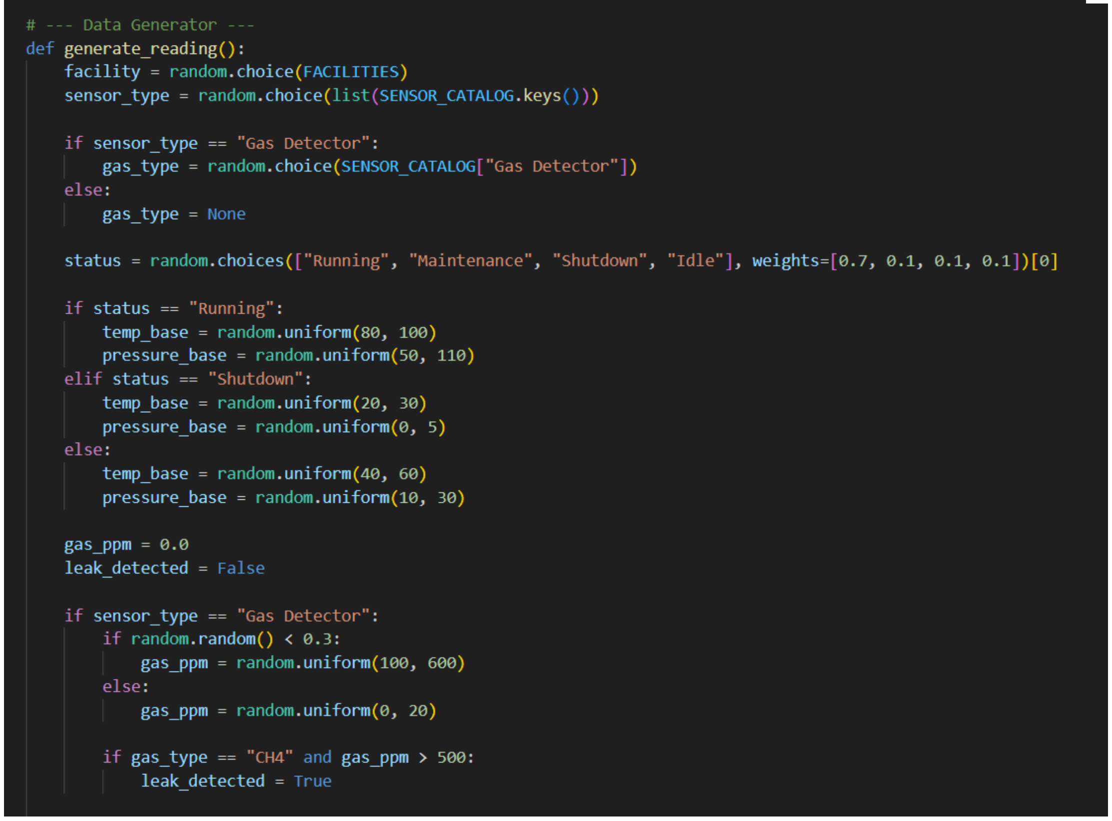
The project architecture combines **Lambda and Medallion architectures** to support real-time and batch processing while maintaining data quality and reliability.

1. **IoT Data Ingestion**:

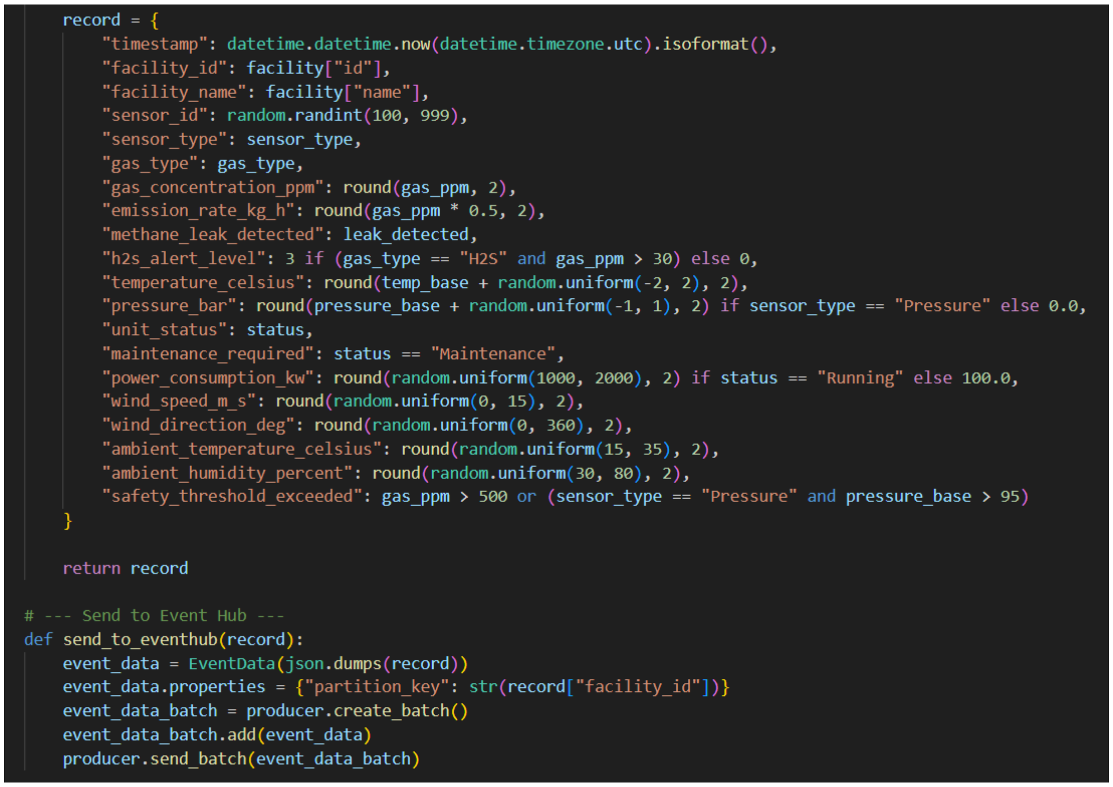
IoT devices installed in various facilities collect sensor readings every 5 seconds, including metrics such as methane and H2S concentrations, temperature, pressure, wind speed, and humidity.


producer.py > ...

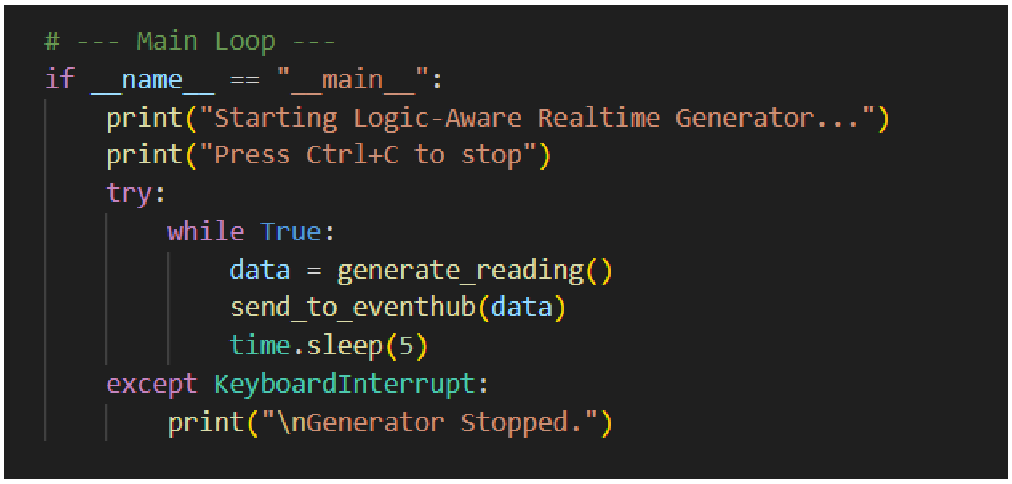
```python
import time
import json
import random
import datetime
from azure.eventhub import EventHubProducerClient, EventData

# --- Configuration ---
FACILITIES = [
    {"id": 9188, "name": "Ras Gas Plant"},
    {"id": 7382, "name": "West Nile Delta"},
    {"id": 8734, "name": "Zohr Gas Plant"}
]

SENSOR_CATALOG = {
    "Gas Detector": ["CH4", "H2S", "CO2", "VOCs"],
    "Temperature": [None],
    "Pressure": [None],
    "Wind": [None]
}

# --- Event Hub Configuration ---
CONNECTION_STR = ""
EVENTHUB_NAME = "gas-detection"

producer = EventHubProducerClient.from_connection_string(
    conn_str=CONNECTION_STR,
    eventhub_name=EVENTHUB_NAME
)
```



```python
# --- Data Generator ---
def generate_reading():
    facility = random.choice(FACILITIES)
    sensor_type = random.choice(list(SENSOR_CATALOG.keys()))

    if sensor_type == "Gas Detector":
        gas_type = random.choice(SENSOR_CATALOG["Gas Detector"])
    else:
        gas_type = None

    status = random.choices(["Running", "Maintenance", "Shutdown", "Idle"], weights=[0.7, 0.1, 0.1, 0.1])[0]

    if status == "Running":
        temp_base = random.uniform(80, 100)
        pressure_base = random.uniform(50, 110)
    elif status == "Shutdown":
        temp_base = random.uniform(20, 30)
        pressure_base = random.uniform(0, 5)
    else:
        temp_base = random.uniform(40, 60)
        pressure_base = random.uniform(10, 30)

    gas_ppm = 0.0
    leak_detected = False

    if sensor_type == "Gas Detector":
        if random.random() < 0.3:
            gas_ppm = random.uniform(100, 600)
        else:
            gas_ppm = random.uniform(0, 20)

        if gas_type == "CH4" and gas_ppm > 500:
            leak_detected = True
```



```python
    record = {
        "timestamp": datetime.datetime.now(datetime.timezone.utc).isoformat(),
        "facility_id": facility["id"],
        "facility_name": facility["name"],
        "sensor_id": random.randint(100, 999),
        "sensor_type": sensor_type,
        "gas_type": gas_type,
        "gas_concentration_ppm": round(gas_ppm, 2),
        "emission_rate_kg_h": round(gas_ppm * 0.5, 2),
        "methane_leak_detected": leak_detected,
        "h2s_alert_level": 3 if (gas_type == "H2S" and gas_ppm > 30) else 0,
        "temperature_celsius": round(temp_base + random.uniform(-2, 2), 2),
        "pressure_bar": round(pressure_base + random.uniform(-1, 1), 2) if sensor_type == "Pressure" else 0.0,
        "unit_status": status,
        "maintenance_required": status == "Maintenance",
        "power_consumption_kw": round(random.uniform(1000, 2000), 2) if status == "Running" else 100.0,
        "wind_speed_m_s": round(random.uniform(0, 15), 2),
        "wind_direction_deg": round(random.uniform(0, 360), 2),
        "ambient_temperature_celsius": round(random.uniform(15, 35), 2),
        "ambient_humidity_percent": round(random.uniform(30, 80), 2),
        "safety_threshold_exceeded": gas_ppm > 500 or (sensor_type == "Pressure" and pressure_base > 95)
    }

    return record

# --- Send to Event Hub ---
def send_to_eventhub(record):
    event_data = EventData(json.dumps(record))
    event_data.properties = {"partition_key": str(record["facility_id"])}
    event_data_batch = producer.create_batch()
    event_data_batch.add(event_data)
    producer.send_batch(event_data_batch)
```



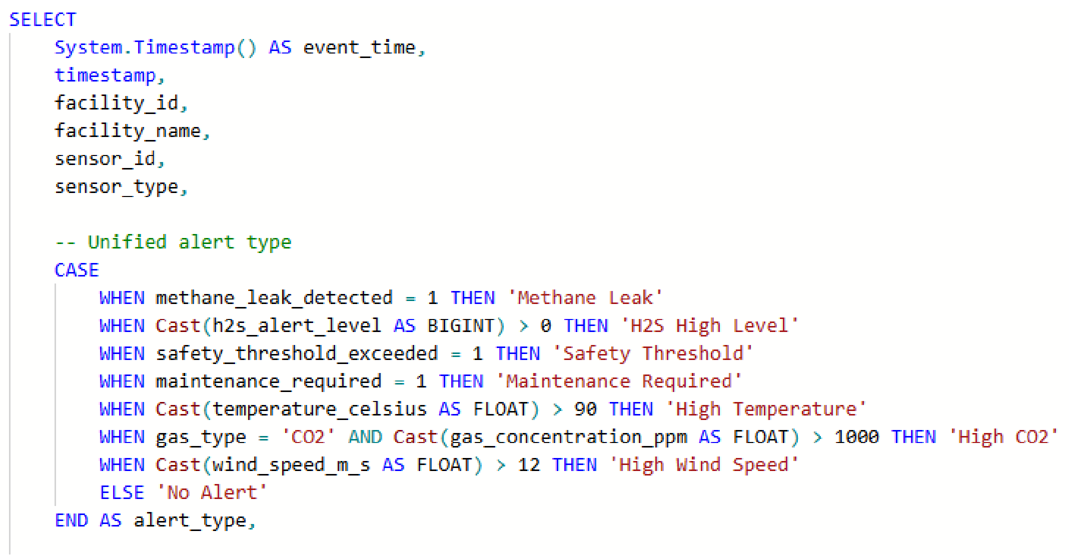
```python
# --- Main Loop ---
if __name__ == "__main__":
    print("Starting Logic-Aware Realtime Generator...")
    print("Press Ctrl+C to stop")
    try:
        while True:
            data = generate_reading()
            send_to_eventhub(data)
            time.sleep(5)
    except KeyboardInterrupt:
        print("\nGenerator Stopped.")
```

These readings are sent to **Azure Event Hub**, a scalable, cloud-based event ingestion service that handles high-frequency streams reliably.

○

2. **Stream Processing**:

○ **Azure Stream Analytics** processes the incoming data in real-time, performing transformations, anomaly detection, and routing to different endpoints.



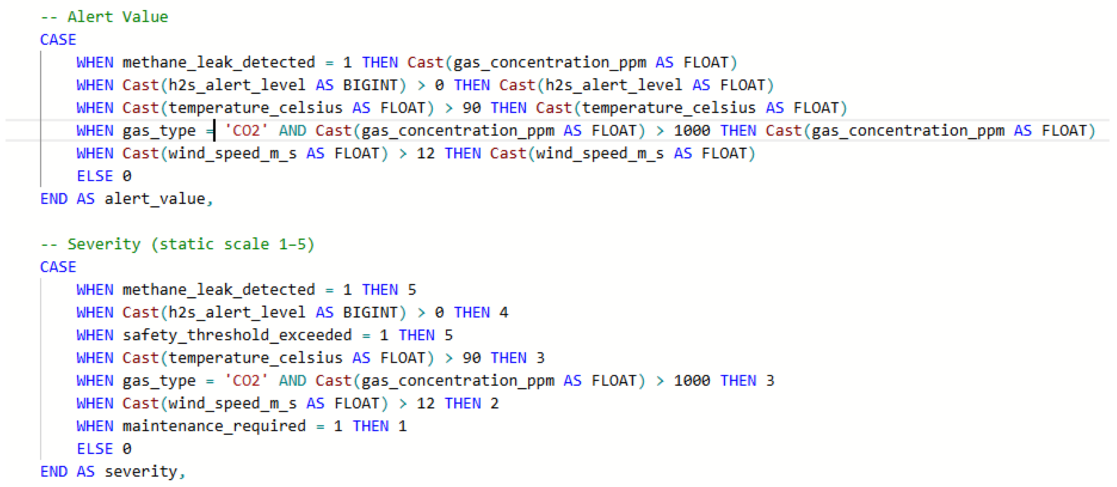
```sql
SELECT
    System.Timestamp() AS event_time,
    timestamp,
    facility_id,
    facility_name,
    sensor_id,
    sensor_type,

    -- Unified alert type
    CASE
        WHEN methane_leak_detected = 1 THEN 'Methane Leak'
        WHEN Cast(h2s_alert_level AS BIGINT) > 0 THEN 'H2S High Level'
        WHEN safety_threshold_exceeded = 1 THEN 'Safety Threshold'
        WHEN maintenance_required = 1 THEN 'Maintenance Required'
        WHEN Cast(temperature_celsius AS FLOAT) > 90 THEN 'High Temperature'
        WHEN gas_type = 'CO2' AND Cast(gas_concentration_ppm AS FLOAT) > 1000 THEN 'High CO2'
        WHEN Cast(wind_speed_m_s AS FLOAT) > 12 THEN 'High Wind Speed'
        ELSE 'No Alert'
    END AS alert_type,
```

○

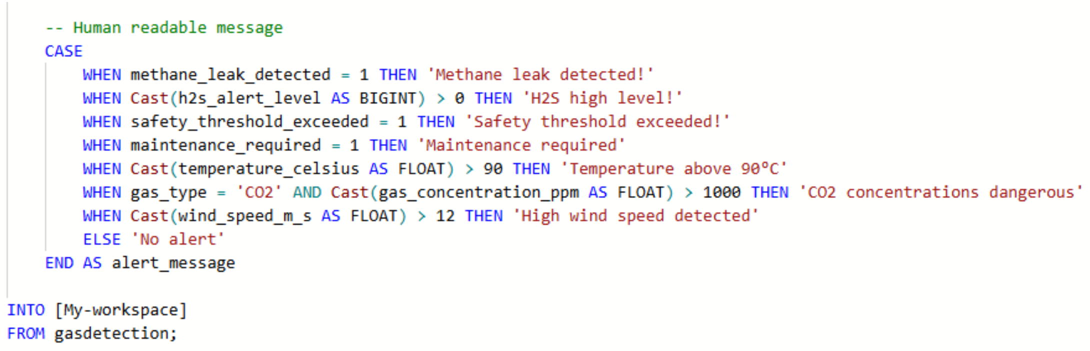
○

○

○



```sql
-- Alert Value
CASE
    WHEN methane_leak_detected = 1 THEN Cast(gas_concentration_ppm AS FLOAT)
    WHEN Cast(h2s_alert_level AS BIGINT) > 0 THEN Cast(h2s_alert_level AS FLOAT)
    WHEN Cast(temperature_celsius AS FLOAT) > 90 THEN Cast(temperature_celsius AS FLOAT)
    WHEN gas_type = 'CO2' AND Cast(gas_concentration_ppm AS FLOAT) > 1000 THEN Cast(gas_concentration_ppm AS FLOAT)
    WHEN Cast(wind_speed_m_s AS FLOAT) > 12 THEN Cast(wind_speed_m_s AS FLOAT)
    ELSE 0
END AS alert_value,

-- Severity (static scale 1-5)
CASE
    WHEN methane_leak_detected = 1 THEN 5
    WHEN Cast(h2s_alert_level AS BIGINT) > 0 THEN 4
    WHEN safety_threshold_exceeded = 1 THEN 5
    WHEN Cast(temperature_celsius AS FLOAT) > 90 THEN 3
    WHEN gas_type = 'CO2' AND Cast(gas_concentration_ppm AS FLOAT) > 1000 THEN 3
    WHEN Cast(wind_speed_m_s AS FLOAT) > 12 THEN 2
    WHEN maintenance_required = 1 THEN 1
    ELSE 0
END AS severity,
```



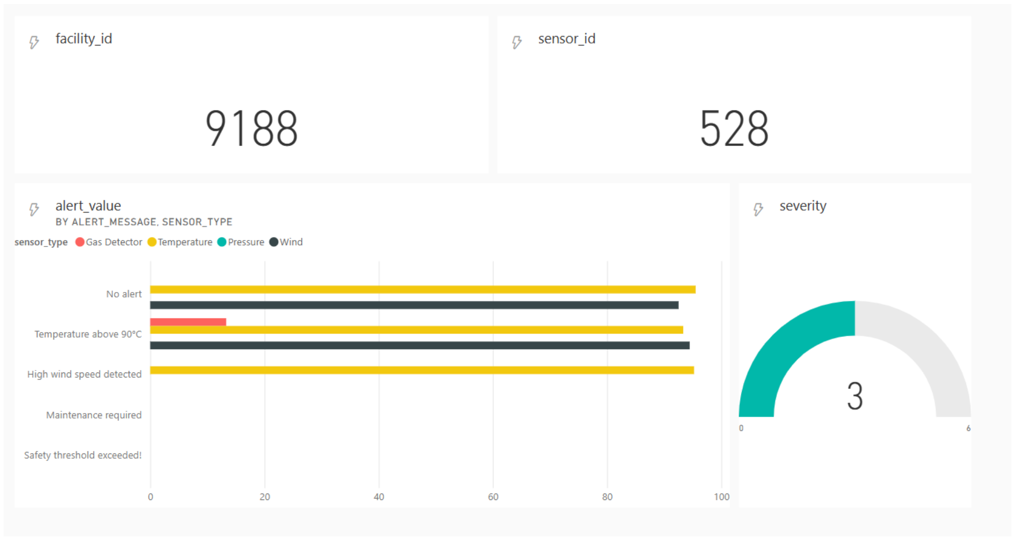
```sql
    -- Human readable message
    CASE
        WHEN methane_leak_detected = 1 THEN 'Methane leak detected!'
        WHEN Cast(h2s_alert_level AS BIGINT) > 0 THEN 'H2S high level!'
        WHEN safety_threshold_exceeded = 1 THEN 'Safety threshold exceeded!'
        WHEN maintenance_required = 1 THEN 'Maintenance required'
        WHEN Cast(temperature_celsius AS FLOAT) > 90 THEN 'Temperature above 90°C'
        WHEN gas_type = 'CO2' AND Cast(gas_concentration_ppm AS FLOAT) > 1000 THEN 'CO2 concentrations dangerous'
        WHEN Cast(wind_speed_m_s AS FLOAT) > 12 THEN 'High wind speed detected'
        ELSE 'No alert'
    END AS alert_message

INTO [My-workspace]
FROM gasdetection;
```

○

```
INTO [alerts-output-hub]
FROM gasdetection;


select *
INTO blobarchive
FROM gasdetection;
```

Real-time streaming data flows to:

▪ **Power BI** for live dashboards, enabling instant monitoring of sensor readings and environmental conditions.



▪ **Azure Data Lake** on an hourly basis for batch storage and long-term analytics.

3. **Data Lakehouse with Medallion Architecture**:

○ **Bronze Layer:** Stores raw, unprocessed sensor data incrementally. This layer preserves the fidelity of the source and serves as a foundation for all downstream processes.

○ **Silver Layer:** Applies transformations such as data cleansing, outlier detection, type casting, timestamping, and risk level calculations. Deduplication ensures data uniqueness per sensor, timestamp, and facility.

○ **Gold Layer:** Aggregates and enriches data into fact and dimension tables for analytics and reporting. This layer supports historical dashboards, predictive analytics, and AI queries.

4. **Dimensional Modeling in Gold Layer**:

○ **Fact tables**: Capture aggregated sensor readings, environmental risk levels, safety violations, and equipment status.

○ **Dimension tables**: Include **time**, **facility**, and **sensor** dimensions, allowing for detailed slicing and dicing of the data.

o **Slowly Changing Dimensions (SCD Type 2)**: Applied for sensors to track changes over time, ensuring historical analysis reflects accurate device characteristics.

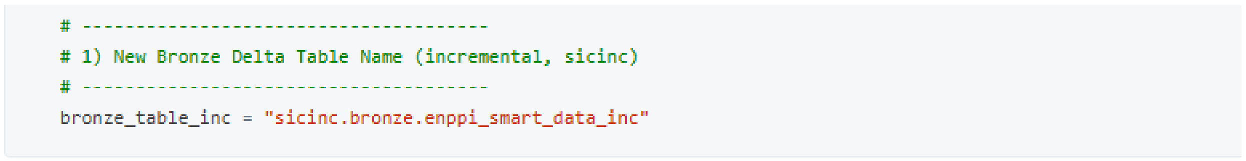
5. **Automation & Cloud-Native Features**:

o End-to-end pipelines are fully automated using **Azure Databricks** and **PySpark**, eliminating manual ETL interventions.

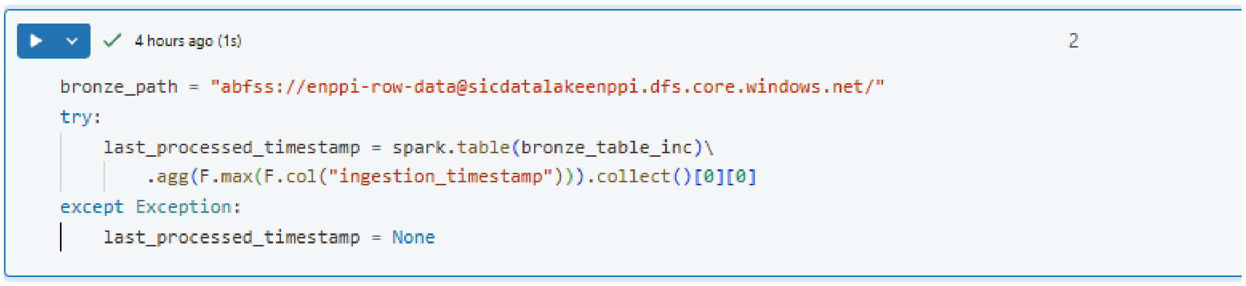
o Delta Lake ensures ACID-compliant merges, incremental processing, and schema enforcement.

o **Z-order optimization** enhances query performance in Gold tables, making analytics and AI-driven queries efficient even on large datasets.

Bronze Layer: - Raw sensor data is ingested incrementally and stored as Delta tables. - Supports automatic handling of new and late-arriving data using Delta Merge operations. - Example code snippet shows merging incremental data while maintaining historical records.

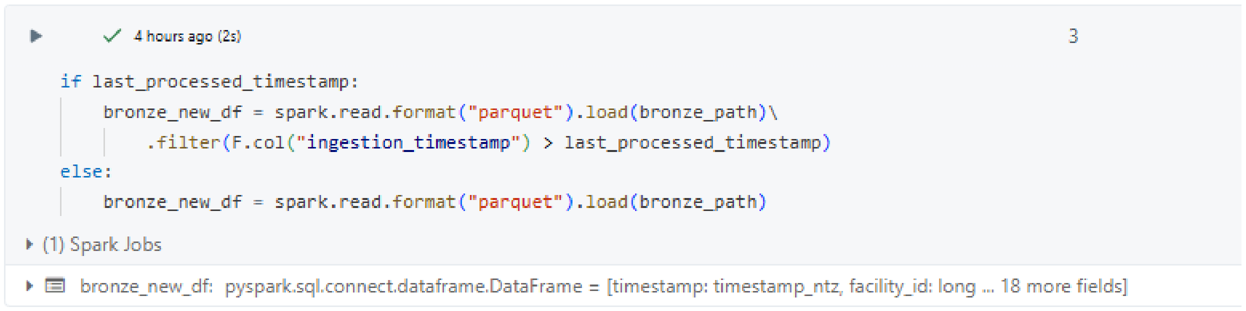


```python
# --------------------------------------
# 1) New Bronze Delta Table Name (incremental, sicinc)
# --------------------------------------
bronze_table_inc = "sicinc.bronze.enppi_smart_data_inc"
```



4 hours ago (1s) 2

```python
bronze_path = "abfss://enppi-row-data@sicdatalakeenppi.dfs.core.windows.net/"
try:
    last_processed_timestamp = spark.table(bronze_table_inc)\
        .agg(F.max(F.col("ingestion_timestamp"))).collect()[0][0]
except Exception:
    last_processed_timestamp = None
```

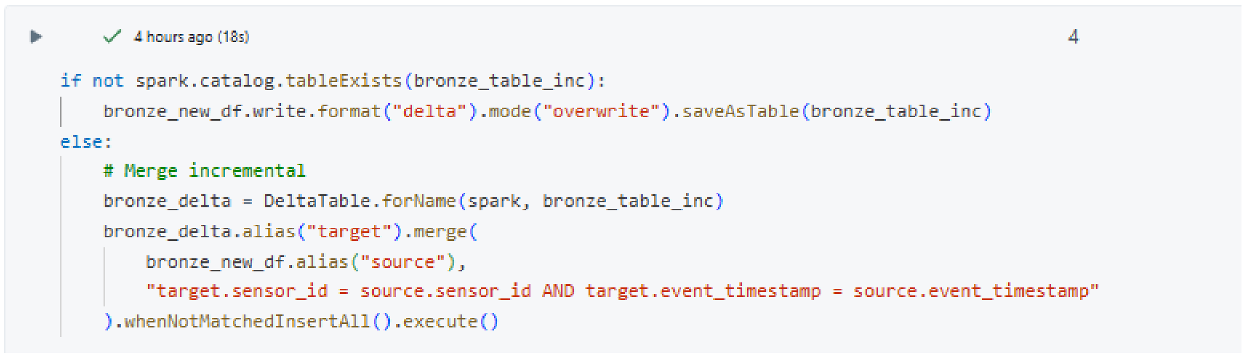


4 hours ago (2s) 3

```python
if last_processed_timestamp:
    bronze_new_df = spark.read.format("parquet").load(bronze_path)\
        .filter(F.col("ingestion_timestamp") > last_processed_timestamp)
else:
    bronze_new_df = spark.read.format("parquet").load(bronze_path)
```

(1) Spark Jobs

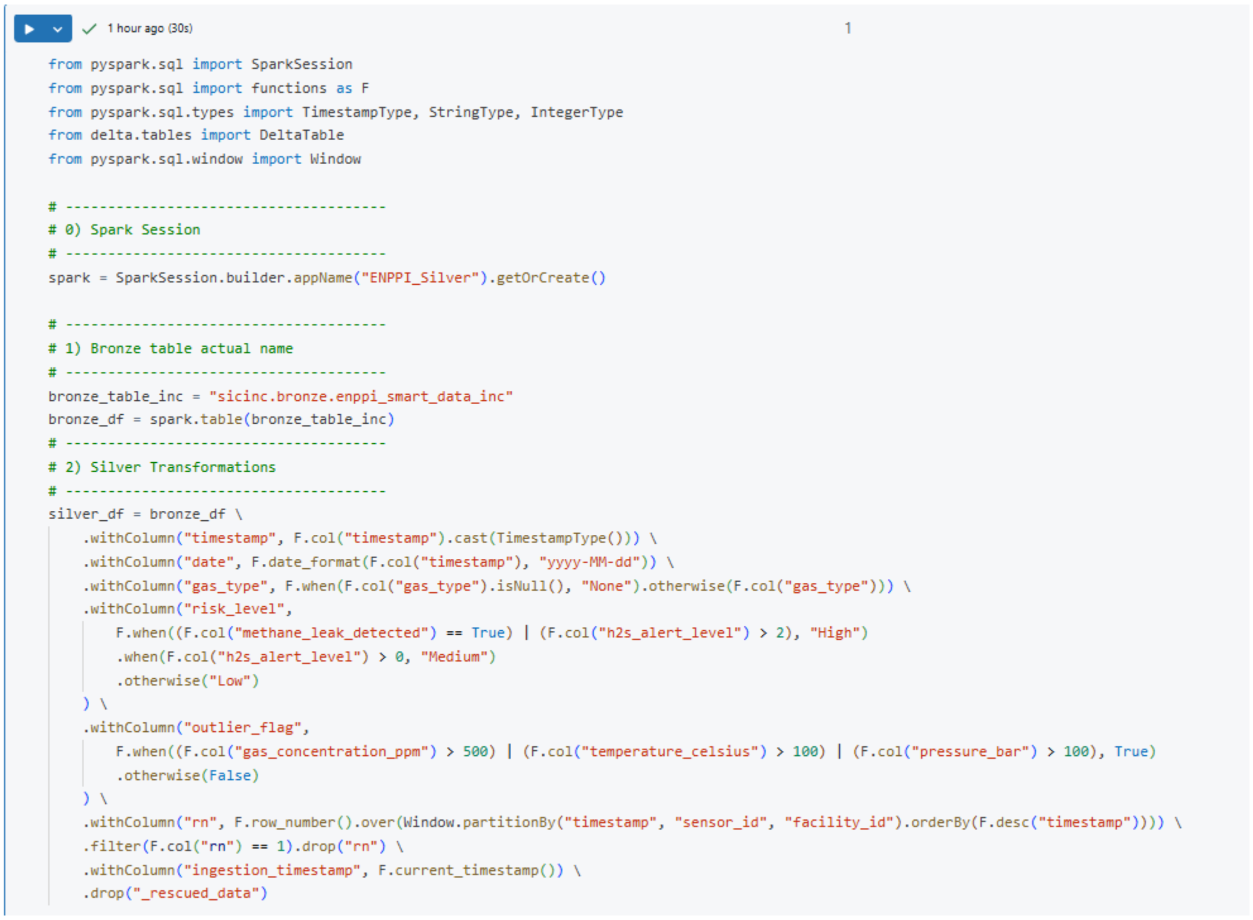
bronze_new_df: pyspark.sql.connect.dataframe.DataFrame = [timestamp: timestamp_ntz, facility_id: long ... 18 more fields]



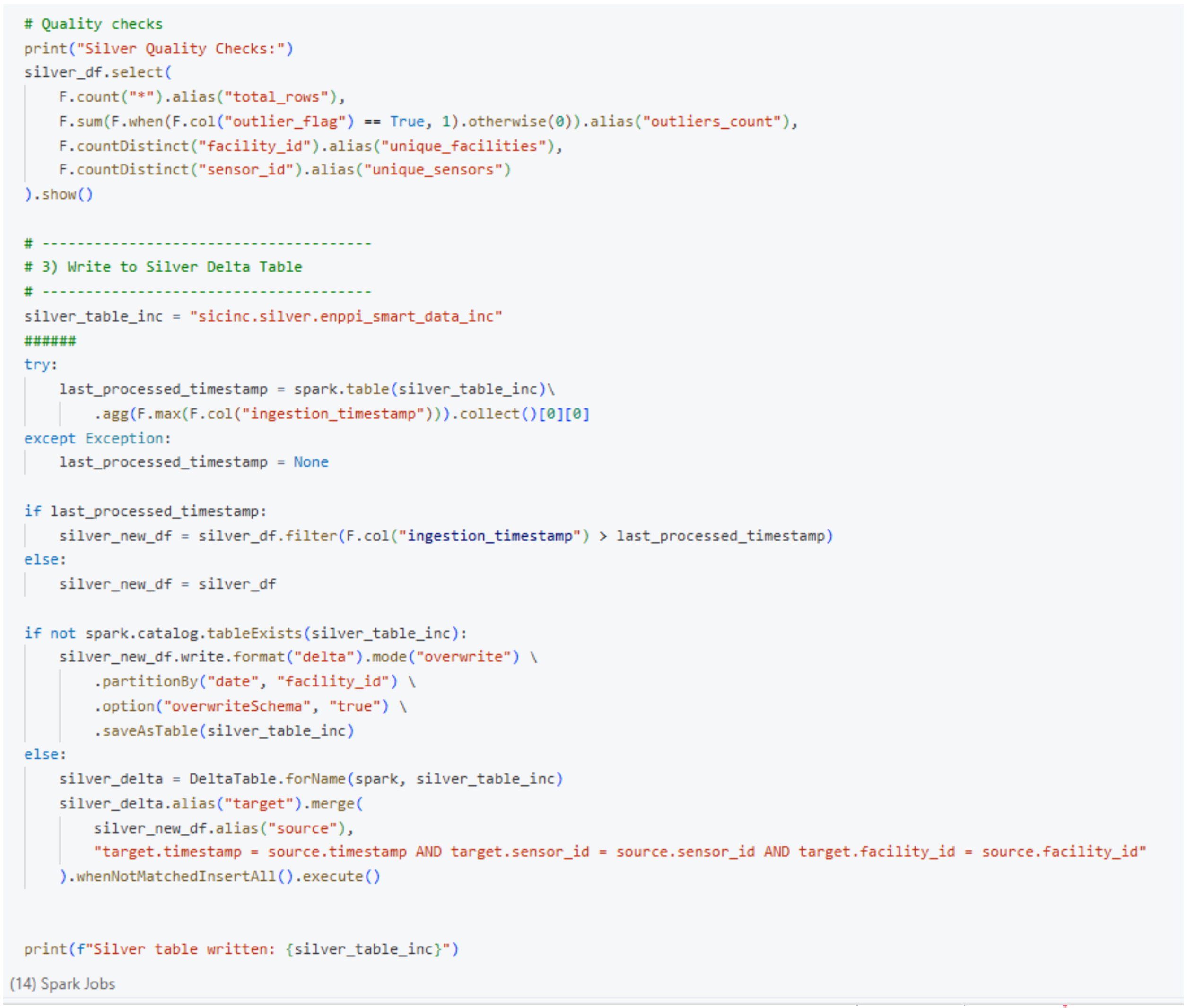
4 hours ago (18s) 4

```python
if not spark.catalog.tableExists(bronze_table_inc):
    bronze_new_df.write.format("delta").mode("overwrite").saveAsTable(bronze_table_inc)
else:
    # Merge incremental
    bronze_delta = DeltaTable.forName(spark, bronze_table_inc)
    bronze_delta.alias("target").merge(
        bronze_new_df.alias("source"),
        "target.sensor_id = source.sensor_id AND target.event_timestamp = source.event_timestamp"
    ).whenNotMatchedInsertAll().execute()
```

Silver Layer: - Data cleansing, transformation, and enrichment steps include: - Converting timestamps and standardizing data types. - Calculating risk levels based on sensor readings. - Flagging outliers and anomalies in gas concentrations, temperature, and pressure. - Deduplication of multiple readings from the same sensor within a short timeframe. - Quality checks are applied to monitor total rows, outlier counts, unique facilities, and sensors. - Incremental writes to Delta tables ensure only new or updated records are processed, optimizing storage and compute usage.

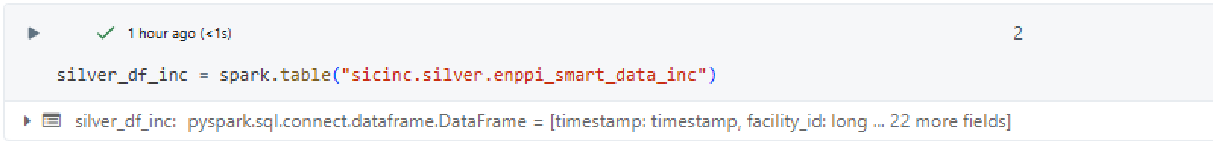

1 hour ago (30s)

```
from pyspark.sql import SparkSession
from pyspark.sql import functions as F
from pyspark.sql.types import TimestampType, StringType, IntegerType
from delta.tables import DeltaTable
from pyspark.sql.window import Window

# -------------------------------------
# 0) Spark Session
# -------------------------------------
spark = SparkSession.builder.appName("ENPPI_Silver").getOrCreate()

# -------------------------------------
# 1) Bronze table actual name
# -------------------------------------
bronze_table_inc = "sicinc.bronze.enppi_smart_data_inc"
bronze_df = spark.table(bronze_table_inc)
# -------------------------------------
# 2) Silver Transformations
# -------------------------------------
silver_df = bronze_df \
    .withColumn("timestamp", F.col("timestamp").cast(TimestampType())) \
    .withColumn("date", F.date_format(F.col("timestamp"), "yyyy-MM-dd")) \
    .withColumn("gas_type", F.when(F.col("gas_type").isNull(), "None").otherwise(F.col("gas_type"))) \
    .withColumn("risk_level",
        F.when((F.col("methane_leak_detected") == True) | (F.col("h2s_alert_level") > 2), "High")
        .when(F.col("h2s_alert_level") > 0, "Medium")
        .otherwise("Low")
    ) \
    .withColumn("outlier_flag",
        F.when((F.col("gas_concentration_ppm") > 500) | (F.col("temperature_celsius") > 100) | (F.col("pressure_bar") > 100), True)
        .otherwise(False)
    ) \
    .withColumn("rn", F.row_number().over(Window.partitionBy("timestamp", "sensor_id", "facility_id").orderBy(F.desc("timestamp")))) \
    .filter(F.col("rn") == 1).drop("rn") \
    .withColumn("ingestion_timestamp", F.current_timestamp()) \
    .drop("_rescued_data")
```



```
# Quality checks
print("Silver Quality Checks:")
silver_df.select(
    F.count("*").alias("total_rows"),
    F.sum(F.when(F.col("outlier_flag") == True, 1).otherwise(0)).alias("outliers_count"),
    F.countDistinct("facility_id").alias("unique_facilities"),
    F.countDistinct("sensor_id").alias("unique_sensors")
).show()

# -------------------------------------
# 3) Write to Silver Delta Table
# -------------------------------------
silver_table_inc = "sicinc.silver.enppi_smart_data_inc"
######
try:
    last_processed_timestamp = spark.table(silver_table_inc)\
        .agg(F.max(F.col("ingestion_timestamp"))).collect()[0][0]
except Exception:
    last_processed_timestamp = None

if last_processed_timestamp:
    silver_new_df = silver_df.filter(F.col("ingestion_timestamp") > last_processed_timestamp)
else:
    silver_new_df = silver_df

if not spark.catalog.tableExists(silver_table_inc):
    silver_new_df.write.format("delta").mode("overwrite") \
        .partitionBy("date", "facility_id") \
        .option("overwriteSchema", "true") \
        .saveAsTable(silver_table_inc)
else:
    silver_delta = DeltaTable.forName(spark, silver_table_inc)
    silver_delta.alias("target").merge(
        silver_new_df.alias("source"),
        "target.timestamp = source.timestamp AND target.sensor_id = source.sensor_id AND target.facility_id = source.facility_id"
    ).whenNotMatchedInsertAll().execute()


print(f"Silver table written: {silver_table_inc}")
```

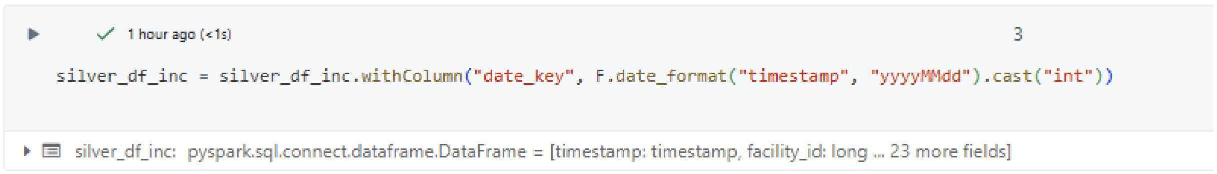
(14) Spark Jobs



1 hour ago (<1s) 2

```
silver_df_inc = spark.table("sicinc.silver.enppi_smart_data_inc")
```

```
silver_df_inc:  pyspark.sql.connect.dataframe.DataFrame = [timestamp: timestamp, facility_id: long ... 22 more fields]
```

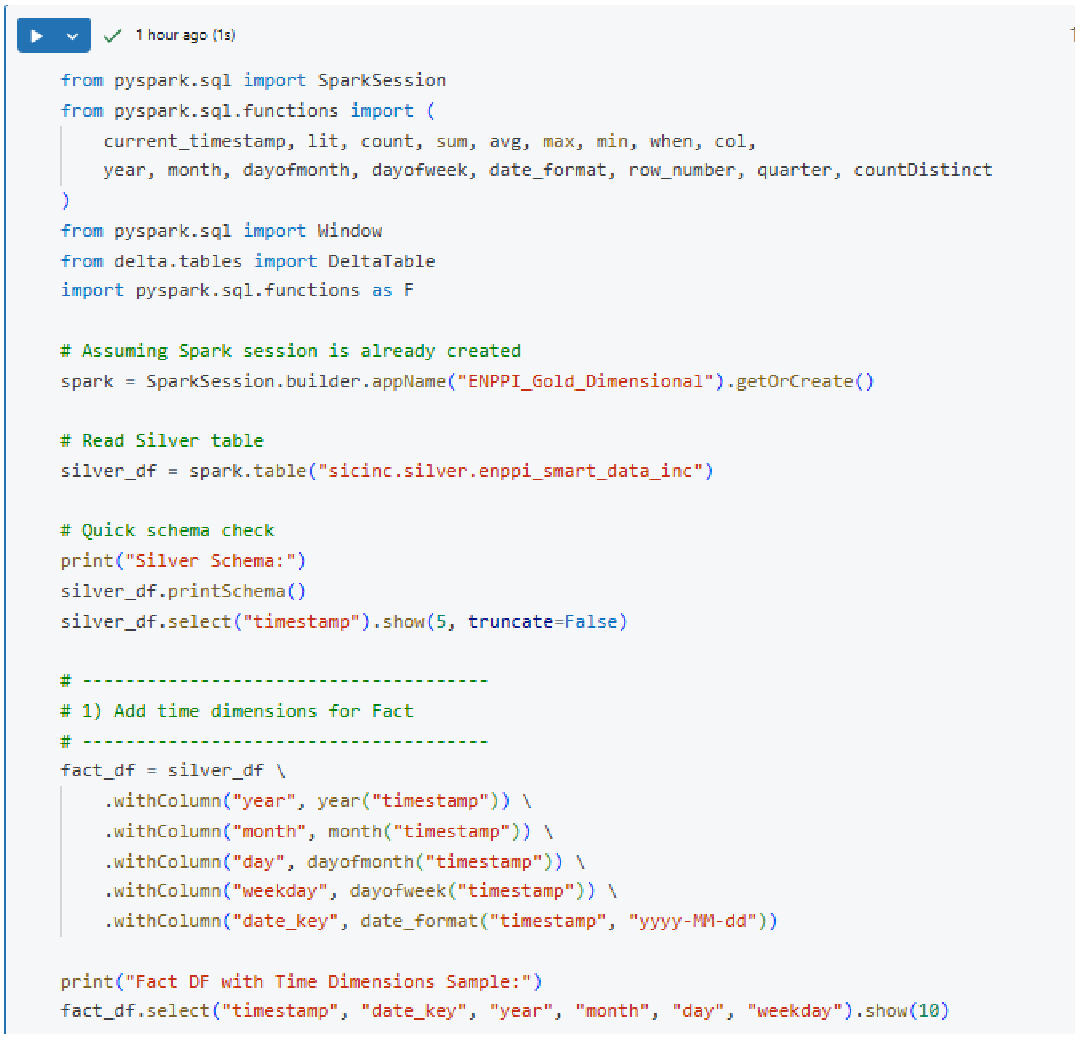


1 hour ago (<1s) 3

```
silver_df_inc = silver_df_inc.withColumn("date_key", F.date_format("timestamp", "yyyyMMdd").cast("int"))
```

```
silver_df_inc:  pyspark.sql.connect.dataframe.DataFrame = [timestamp: timestamp, facility_id: long ... 23 more fields]
```

Gold Layer: - Aggregates daily metrics for analytical use cases. - Fact tables include emissions, average gas concentrations, leak counts, safety violations, running unit counts, and environmental metrics. - Dimensions include: - **Dim_Time**: Tracks year, month, day, weekday, quarter, and derived date keys. - **Dim_Facility**: Maintains unique facility records with surrogate keys. - **Dim_Sensor**: Tracks sensor metadata with SCD Type 2 for historical changes. - Joins between fact and dimension tables generate a fully enriched dataset ready for analytics, AI, and reporting.



```
from pyspark.sql import SparkSession
from pyspark.sql.functions import (
    current_timestamp, lit, count, sum, avg, max, min, when, col,
    year, month, dayofmonth, dayofweek, date_format, row_number, quarter, countDistinct
)
from pyspark.sql import Window
from delta.tables import DeltaTable
import pyspark.sql.functions as F

# Assuming Spark session is already created
spark = SparkSession.builder.appName("ENPPI_Gold_Dimensional").getOrCreate()

# Read Silver table
silver_df = spark.table("sicinc.silver.enppi_smart_data_inc")

# Quick schema check
print("Silver Schema:")
silver_df.printSchema()
silver_df.select("timestamp").show(5, truncate=False)

# ----------------------------------------
# 1) Add time dimensions for Fact
# ----------------------------------------
fact_df = silver_df \
    .withColumn("year", year("timestamp")) \
    .withColumn("month", month("timestamp")) \
    .withColumn("day", dayofmonth("timestamp")) \
    .withColumn("weekday", dayofweek("timestamp")) \
    .withColumn("date_key", date_format("timestamp", "yyyy-MM-dd"))

print("Fact DF with Time Dimensions Sample:")
fact_df.select("timestamp", "date_key", "year", "month", "day", "weekday").show(10)
```

Data Products

1. **Live Dashboard (Power BI)**:
   - Displays real-time monitoring of environmental metrics, safety alerts, and operational status.
   - Provides actionable insights to plant managers and engineers for immediate operational decisions.
2. **Historical Analysis**:
   - Gold layer supports daily, weekly, monthly, and yearly trend analysis.
   - Enables correlation studies between environmental conditions and operational outcomes.
   - Provides data for regulatory reporting and internal audits.
3. **AI Agent**:
   - Non-technical users can query historical and real-time data using plain language.
   - Converts natural language queries into SQL and returns insights in a human-readable format.
   - Example use case: querying equipment maintenance needs or identifying high-risk facilities.

Key Features

- Incremental and fully automated data ingestion and processing.
- Cloud-native implementation leveraging Azure services.
- Medallion architecture ensures high-quality, structured data across all layers.
- Lambda architecture supports both real-time analytics and batch processing.
- AI agent interface for non-technical stakeholders.
- Metadata tracking and Z-order optimization for efficient query performance.

Quality & Optimization

- Outliers and anomalies flagged at the Silver layer.
- Z-order indexing improves query performance in Gold tables.
- Delta Merge operations ensure ACID-compliant incremental updates.
- Metadata columns track ingestion and update timestamps for auditing and pipeline monitoring.

Use Cases

- Real-time detection of gas leaks and environmental risks.
- Monitoring operational performance and equipment health.
- Predictive maintenance planning using historical sensor data.
- Regulatory and internal reporting for safety and environmental compliance.
- Decision-making support for non-technical personnel through AI-driven natural language queries.

Conclusion

The ENPPI Climate IoT Data Engineering Project demonstrates a fully automated, cloud-native, and scalable solution for environmental monitoring. By combining real-time streaming with batch processing, medallion architecture, and dimensional modeling, the system delivers accurate, timely, and actionable insights. The platform enables operational optimization, enhances safety management, and provides AI-driven analytics for both technical and non-technical users, ensuring that data becomes a strategic product within the organization.