IIoT Data Engineering Lab Assignment

Overview

In this lab assignment, you will build a complete Industrial Internet of Things (IIoT) data pipeline that simulates real-world sensor data collection, processes it in real-time, and stores it for both immediate and historical analysis. This hands-on project will give you practical experience with modern data engineering tools and streaming architectures.

Duration: 5-6 hours
Team Size: 2 students

Learning Objectives

By completing this assignment, you will:

- Design and implement an end-to-end streaming data pipeline
- Work with message brokers (Kafka/Redpanda) for event streaming
- Implement stream processing using Apache Flink
- Perform batch ETL jobs using Apache Spark
- Orchestrate workflows with Prefect
- Store time-series data efficiently with TimescaleDB
- Work with data lakes using Delta Lake format
- Deploy and manage containerized applications with Docker Compose

---

Prerequisites

Required Software

- WSL 2 & Docker

Architecture Overview

You will build the following data pipeline:

```
 [Simulated Machines]
        ↓
[Python Producer] —→ [Redpanda Topic]
                            ↓
                    [Apache Flink] —→ [TimescaleDB]
                                            ↓
                    [Spark Jobs] ←—— [Prefect]
                            ↓
                    [Delta Lake on Azurite]
```

Components to Implement:

1. **Redpanda**: Kafka-compatible message broker
2. **Data Ingestion**: Python service simulating machine sensors
3. **Apache Flink**: Real-time stream processing
4. **TimescaleDB**: Time-series database storage
5. **Apache Spark**: Batch processing engine
6. **Prefect**: Workflow orchestration
7. **Delta Lake on Azurite**: Data lake storage

Tasks

Task 1: Infrastructure Setup (30-45 minutes)

1.1 Create Docker Compose Configuration

Create a `docker-compose.yml` file that orchestrates all required services:

Required Services:

- **Redpanda**
  - Include health checks
- **Redpanda Console**
  - Web UI
  - Connect to Redpanda broker
- **TimescaleDB**
  - PostgreSQL on port 5432
  - Database name: `iiot`
  - Username: `admin`
  - Password: `admin`
  - Mount initialization SQL script
  - Include health checks
- **Flink Cluster**
  - JobManager (with web UI)
  - At least 1 TaskManager
  - Mount job directory for Python scripts
- **Spark Cluster**
  - At least 1 Worker node
  - Mount job directory
- **Azurite** (Azure Blob Storage Emulator)
  - Blob service
  - Mount data volume
- **Prefect Server**
  - UI on port 4200
  - Include health checks

Tips:

- Create a custom bridge network for service communication
- Use named volumes for data persistence
- Configure proper service dependencies with health checks
- Use environment variables for configuration

1.2 Initialize TimescaleDB Schema

Create `init-db.sql` with the following:

Tables to create:

1. machine_sensors
2. sensor_aggregates

- Choose your columns & indexes.
- Add data retention policy (90 days)

1.3 Start and Verify Infrastructure

Verify all services are running and healthy.

Deliverable for Task 1:

- Working `docker-compose.yml`
- `init-db.sql` with proper schema
- All services running successfully

Task 2: Data Ingestion Service (45-60 minutes)

Create a Python service that simulates industrial machine sensors.

2.1 Setup Python Project

Create `ingestion/` directory with:

- `Dockerfile` - Container definition
- `requirements.txt` - Python dependencies
- `ingest_data.py` - Main ingestion script

2.2 Implement Data Generator

Requirements:

- Simulate **at least 3 machines** (you can add more)
    - Each machine should have: id, type, location and other fields
    - Example types: CNC Mill, Lathe, Press, Grinder, Welder,...
- Each machine generates **at least 3 sensor types** (you can add more):

Functional Requirements:

1. Generate sensor readings with realistic values using Gaussian distribution
2. Publish messages to Redpanda topic `machine-sensors`
3. Send messages every x seconds (configurable via environment variable)
4. At start of the script, generate a bunch of messages from the past week to spread sensor readings
5. Include proper error handling and reconnection logic
6. Log message delivery status

Use JSON as data structure. Think about QoS, partitioning,...

Deliverable for Task 2:

- Working Python ingestion service
- Dockerfile and requirements.txt
- Messages flowing to Redpanda (verify in Console UI)

Task 3: Stream Processing with Apache Flink (75-90 minutes)

Implement real-time stream processing using Apache Flink's Table API & DataStream API.

3.1 Setup Flink Jobs Directory

Create `flink-jobs/` directory for your Python Flink jobs.

3.2 Implement Sensor Aggregation Job

Create `flink-jobs/sensor_aggregation.py` that:

Requirements:

1. Reads from Redpanda topic `machine-sensors`
2. Mix usage of **Flink Table API** (SQL-based approach) & **DataStream API**
3. Implements **tumbling & sliding windows** of 1 minute (mix usage)
4. Computes aggregations per window:
    - Average value
    - Minimum value
    - Maximum value
    - Count of readings
5. Groups by: machine_id, sensor_type
6. Writes results to TimescaleDB `sensor_aggregates` table
7. Uses watermarks for handling out-of-order events (5 seconds)
8. Also write raw machine data to a TimescaleDB table named `machine_sensors`.

Tips:

- Configure proper watermark strategy
- Handle late-arriving data appropriately

Deliverable for Task 3:

- Working Flink aggregation jobs (mix usage so you used everything at least once)
- Aggregated data visible in TimescaleDB

Task 4: Batch Processing with Apache Spark (45-60 minutes)

Implement nightly ETL job to move data from TimescaleDB to Delta Lake.

4.1 Create Spark Job

Create `spark-jobs/timescale_to_deltalake.py` that:

Requirements:

1. Connects to TimescaleDB using JDBC
2. Extracts data from tables:
    - `machine_sensors`
    - `sensor_aggregates`
3. Writes data to Delta Lake format
4. Stores in Azurite blob storage (container: `datalake`)
5. Partitions by timestamp for each run
6. Includes error handling and logging

Delta Lake Configuration:

- Use Azurite connection string:
  - Account: `devstoreaccount1`
  - Connection: `wasbs://datalake@devstoreaccount1.blob.core.windows.net/`
- Use Spark with Delta Lake

Tips:

- Use PySpark for implementation
- Configure SparkSession with Delta Lake support
- Test locally before orchestrating
- Use appropriate write modes (overwrite per run)

Deliverable for Task 4:

- Working Spark ETL job
- Data successfully written to Delta Lake
- Verifiable in Azurite storage

4.2 (Optional) Create Second Spark Job

Create a second nightly Spark Job that calculates the same aggregations as with Flink. But now with batch processing.

Task 5: Workflow Orchestration with Prefect (45-60 minutes)

Create automated workflow orchestration for the nightly ETL job.

5.1 Create Prefect Flow

Create `prefect-flows/nightly_etl_flow.py` that:

Requirements:

1. Defines a Prefect flow with tasks:
   - **Task 1**: Check TimescaleDB connection
   - **Task 2**: Check Azurite connection
   - **Task 3**: Submit Spark job
   - **Task 4**: Verify Delta Lake data
2. Implements proper error handling:
   - Retry failed tasks (2-3 retries)
   - Retry delays (10-30 seconds)
   - Clear error messages
3. Includes logging and monitoring:
   - Log each step progress
   - Report record counts
   - Summarize results

5.2 Create Deployment Script

Create `prefect-flows/deploy_flow.py` to:

- Register the flow with Prefect server
- Schedule for nightly execution (2:00 AM UTC)
- Use CronSchedule

Tips:

- Test flow manually first
- Monitor execution in Prefect UI (http://localhost:4200)

Deliverable for Task 5:

- Working Prefect flow
- Deployment script
- Scheduled nightly execution
- Successful manual test run
