Advanced-Real-Time-Data-Pipeline-and-Analytical-Processing

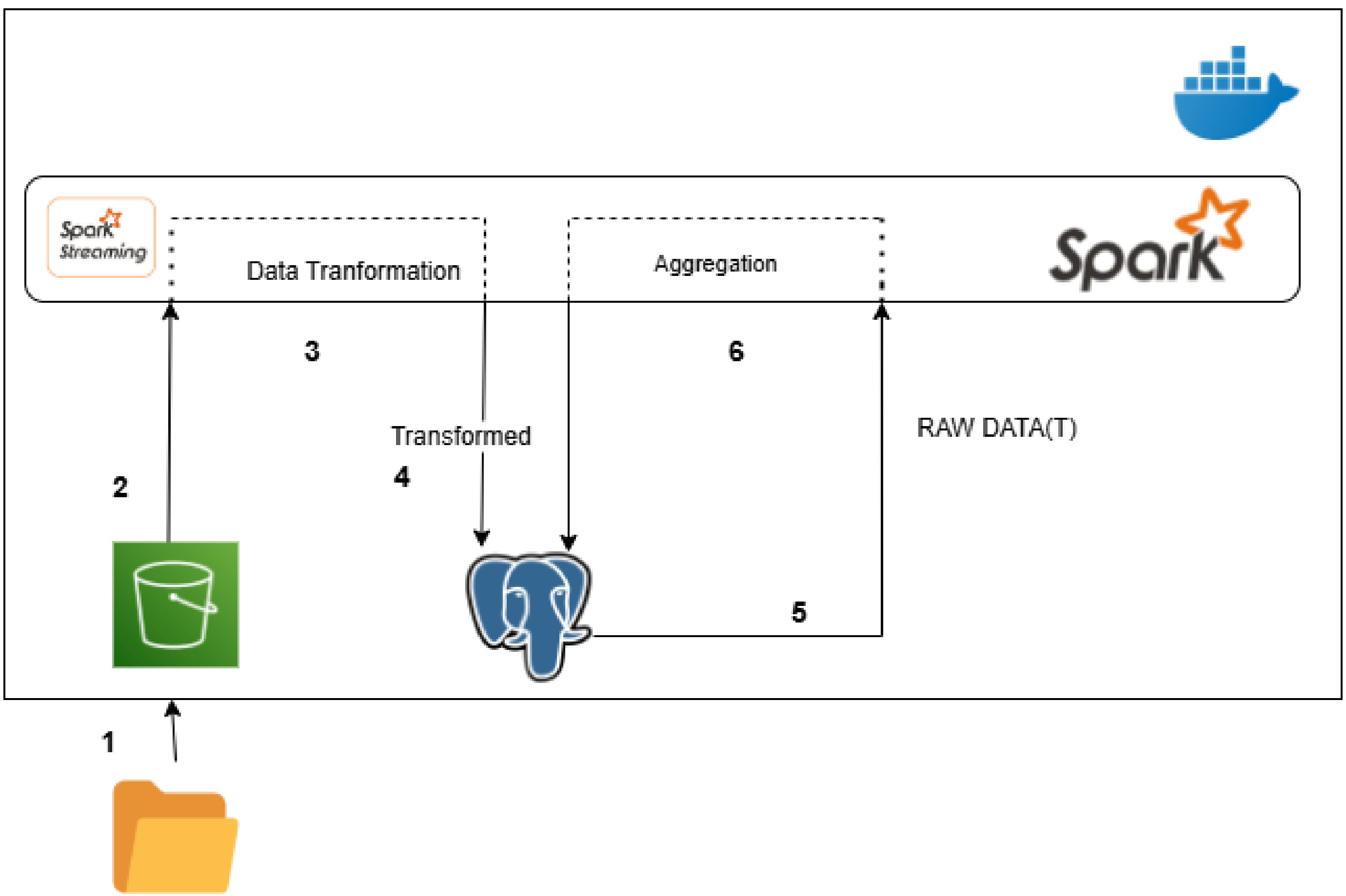
Sudhanshu Pukale

August 2025

1 Objective

Designed and implemented a scalable real-time data pipeline that monitors a directory for incoming data, processes it based on specific criteria, and stores the transformed data in a relational database for further analysis. Consider factors such as data integrity, performance, and scalability.

2 Architecture Workflow



The above architecture diagram flow indicates:-

2.1 Stage 1: Data Ingestion

Input: CSV/JSON files containing IoT sensor data (temperature, humidity, pressure)

Mechanism: `data_schema_upload.bat` script uploads files to `MinIO data/` folder
Schema Support: Optional JSON schema files uploaded to `schema/` folder for validation
Monitoring: Spark Streaming monitors `s3a://sensor-data/data/` every 10 seconds (configurable)
Trigger: File upload triggers immediate processing pipeline via `input_file_name()` detection
—

2.2 Stage 2: Data Storage and Access

MinIO Object Storage: S3-compatible storage running in Docker container
Bucket Structure:

- `data/` - Incoming files awaiting processing
- `schema/` - JSON schema definitions for validation
- `processed/` - Successfully processed files
- `quarantine/` - Files with validation errors
- `audit/` - Processing audit logs

2.3 Stage 3: Data Transformation Pipeline

Processing: Real-time batch processing using `foreachBatch` pattern
Validation Framework (from `helpers.py`):

- Key Fields: `sensor_id`, `timestamp`, `temperature_C` (null checks)
- Numeric Validation: Data type validation for `temperature_C`
- Range Validation: Temperature range -50°C to 50°C
- Heavy Null Detection: Rows with >50% null values flagged

Transformation Tasks:

- String trimming via `trim_all_strings()`
- All-null row removal via `drop_all_null_rows()`
- Metadata enrichment: `file_path`, `ingestion_ts`, `row_hash`
- Dynamic schema loading from MinIO schema files

Error Handling: Invalid records moved to `quarantine/` with detailed error reasons

2.4 Stage 4: Data Persistence

Database: PostgreSQL running in Docker container
Dynamic Table Creation: Tables created based on filename and format
Pattern: `{schema}.{format}_{filename}_transformed` **Example:** `public.csv_farm_data_transformed`
JDBC Configuration: Optimized with batch writes (batch size: 5000)

2.5 Stage 5: Aggregation Processing

Technology: Integrated within the same Spark job via `apply_aggregations()`
Source: Transformed data from the validation stage
Calculations:

- Statistical aggregates per `sensor_id`: min, max, avg, stddev
- Handles missing numeric columns gracefully
- Default grouping by `'unknown'` if `sensor_id` missing

Output Schema:

- `min_{column}`, `max_{column}`, `avg_{column}`, `stddev_{column}`
- Metadata: `data_source`, `file_name`, `ingestion_ts`

Storage: Separate aggregation tables `{format}_sensor_agg`

2.6 Stage 6: File Management and Audit

File Movement: Automatic file organization based on processing results

- Successful: `data/filename.csv` → `processed/filename.csv`
- Failed: `data/filename.csv` → `quarantine/filename.csv`

Audit Logging: Comprehensive processing logs stored in `audit/` folder

- Partitioned by `audit_date` for efficient querying
- Includes: timestamp, filename, format, row counts, status, messages

Quarantine Details:

- JSON format storage with error reasons
- Partitioned by `quarantine_date`
- MD5 hash-based folder organization

3 Installation

Install docker from https://docs.docker.com/desktop/setup/install/windows-install/

4 Version Control

It's advised to download the below version of the application for compatibility

1. spark 3.5.0
2. minio (latest)
3. hadoop-common version 3.3.4
4. hadoop-aws version 3.3.4
5. aws-java-sdk-bundle version 1.12.262

5 Scaling the Pipeline for Production

The local prototype uses Docker, PostgreSQL, and MinIO to demonstrate ingestion, processing, and storage. However, in a production-grade deployment, these components would be replaced with scalable and managed services to ensure high availability, fault tolerance, and enterprise-level reliability.

5.1 Containerization and Orchestration

- In production, **Docker Compose** would be replaced by orchestration and scheduling frameworks.
- The pipeline can be deployed on **cloud-managed platforms** (e.g., AWS EMR, GCP Dataproc, Azure HDInsight).

5.2 Data Ingestion

- **Apache Kafka:** Acts as a distributed messaging backbone for real-time data ingestion, ensuring durability and scalability.
- **Schema Registry:** Prevents schema drift and maintains consistency across producers and consumers.

5.3 Data Processing

- **Apache Spark:** Remains central to large-scale batch and stream processing. Running Spark on a distributed cluster (YARN, Kubernetes, or cloud-native services) allows horizontal scaling to handle terabytes of sensor data.
- **Apache Airflow:** Orchestrates Spark jobs, manages task dependencies, and provides monitoring, retries, and alerting capabilities. This ensures robustness of the pipeline without manual intervention.

5.4 Storage and Databases

- Instead of MinIO, production systems typically leverage **cloud object storage** such as Amazon S3, Google Cloud Storage, or Azure Blob Storage for durability and elasticity.
- For relational workloads, PostgreSQL can be replaced with a **cloud-managed relational database** (e.g., Amazon RDS, Cloud SQL, or Azure Database for PostgreSQL) or with distributed SQL solutions (e.g., CockroachDB, YugabyteDB) for horizontal scalability.

5.5 Monitoring and Reliability

- **Airflow Web UI:** Provides real-time monitoring, failure alerts, and retry mechanisms for ETL workflows.
- **Spark Monitoring (Spark UI / History Server):** Used to analyze performance bottlenecks in streaming jobs.

5.6 Security and Compliance

- Authentication and authorization via **cloud IAM systems** or enterprise-grade solutions like Keycloak.
- End-to-end encryption (TLS for data in transit, server-side encryption for data at rest).
- Audit logging for compliance with GDPR or industry-specific regulations.

5.7 CI/CD and Automation

- **Apache Airflow + CI/CD Pipelines (GitHub Actions, GitLab CI):** Ensure continuous integration, automated testing, and safe deployment of DAGs and Spark jobs.
- Infrastructure managed via **Terraform or Ansible** for reproducibility.

5.8 Scalability Roadmap

The transition from local prototype to production would follow these stages:

1. Replace Docker Compose with cloud-managed services.
2. Use Kafka for ingestion and S3/GCS/Azure Blob for storage instead of MinIO.
3. Scale Spark jobs across distributed clusters for batch and streaming.
4. Orchestrate workflows with Airflow to ensure resilience and observability.
5. Enhance monitoring, logging, and security with enterprise frameworks.

This setup ensures that the pipeline can handle high-frequency sensor streams, scale elastically with demand, and recover automatically from failures, making it production-ready for enterprise environments.

6 Instructions for Setting Up and Running the Pipeline Locally

This section provides step-by-step instructions to set up and run the **Real-Time Data Pipeline** using Docker. The pipeline leverages **Apache Spark** for data processing, **PostgreSQL** for storage, and **MinIO** for object storage.

6.1 Prerequisites

Ensure the following dependencies are installed on your local machine:

- Docker (version $\geq$ 20.x) Install Guide
- Docker Compose (version $\geq$ 1.29.x)
- Git (optional, if cloning the repository)
- Python 3.9 or higher (for testing utilities and schema tools)

Verify installation:

```
docker --version
docker-compose --version
python --version
```

6.2 Clone the Repository

```
https://github.com/Sudhanshu132/Advanced-Real-Time-Data-Pipeline-and-Analytical-Processing.git
```

6.3 Configure Environment

Update the `.env` file with the following credentials:

```
# MinIO Config
MINIO_ENDPOINT=http://minio:9000
MINIO_ACCESS_KEY=admin
MINIO_SECRET_KEY=Goodgame25
BUCKET_NAME=sensor-data

# PostgreSQL Config
POSTGRES_USER=admin
POSTGRES_PASSWORD=Goodgame25
POSTGRES_DB=farmingdb
POSTGRES_PORT=5432
```

6.4 Build Docker Images

Run the setup script:

- On Windows:

  ```
  start.bat
  ```

- On Linux/Mac:

  ```
  start.sh
  ```

This will:

1. Build the Spark Docker image (`sdp-spark:1.0.0`).
2. Start containers using `docker-compose`.
3. Initialize MinIO buckets and PostgreSQL schema.

6.5 Verify Services

Check running containers:

```
docker ps
```

You should see containers for Spark, PostgreSQL, MinIO, and Spark Worker. Access UIs:

- MinIO: `http://localhost:9001`
- PostgreSQL (via pgAdmin/DBeaver): Host = `localhost`, Port = `5432`
- Spark UI: `http://localhost:8080`

6.6 Upload Input Data

Upload CSV/JSON files under the `data/` folder and run below bat file
NOTE :- Only run after Start.bat file.

```
data_schema_upload.bat
```

6.7 Run the Pipeline

The pipeline runs automatically when new files arrive in MinIO.
To trigger manually:

```
docker exec spark spark-submit /home/spark/Main.py
```

6.8 Check Processed Data

Processed data is available in:

- MinIO: `processed/` folder
- PostgreSQL: Table `smart_farming_crop_yield_2024_transformed`

  Example query:

```
docker exec -it postgres psql -U admin -d farmingdb \
  -c "SELECT * FROM smart_farming_crop_yield_2024_transformed LIMIT 10;"
```

6.9 Recovery and Error Handling

- Invalid files → moved to `quarantine/` in MinIO.
- Temporary PostgreSQL unavailability → automatic retries.
- Errors are logged in container logs (`docker logs spark`).

6.10 Stopping the Pipeline

Stop all containers:

```
docker-compose down
```

To reset everything:

```
docker system prune -a
OR
stop.bat
```