
Universidad de
Castilla-La Mancha

# Proyecto Big Data

**_SmartPool_: Sistema de monitorización y análisis de datos en piscinas públicas**

**Autor:** Iván Vicente Hernández García de Mora

**Estudios:** Máster Universitario en Ingeniería Informática

**Curso:** 2025/26

**Asignatura:** Arquitectura de Sistemas Big Data | Procesamiento Masivo de Datos

**Fecha:** 29 de enero del 2026

# 1. Introducción y objetivos

El proyecto *SmartPool* se centra en construir una arquitectura Medallion (Bronze/Silver/Gold) capaz de integrar tres tipos de ingesta: datos estructurados en batch, datos semiestructurados en batch y datos en streaming.

La idea principal es disponer de un flujo completo que cubra la captura, limpieza, enriquecimiento y analítica operativa de piscinas: calidad del agua, mantenimiento y consumo energético.

Se comenzó trabajando con notebooks (`notebooks/01..07_*.ipynb`) para validar cada fase y, una vez estabilizados los resultados, se transformaron en scripts Python ejecutables desde Airflow (`spark-apps/`).

El resultado es un sistema reproducible que automatiza ingestas, mantiene histórico en Delta y combina información batch con eventos en tiempo real.

# 2. Diseño Medallion y configuración

La persistencia se organiza en MinIO con rutas S3A para las capas Bronze/Silver/Gold y una zona `_state` para checkpoints de ingesta incremental.

La configuración de Spark se centraliza en `spark-apps/smartpool_config.py` para evitar modificar múltiples archivos, incluyendo parámetros S3A, extensiones de Delta Lake, zona horaria y conexión JDBC a SQL Server. Además, se fijan valores por defecto mediante variables de entorno para facilitar despliegues consistentes.

Para mejorar la estabilidad de ejecución, se ajustaron límites de recursos y paralelismo de los jobs lanzados desde Airflow en `docker-compose.yml`. Esto fue clave para evitar bloqueos por consumo de memoria y para mantener la ejecución concurrente de Airflow y Spark sin saturar el entorno.

# 3. Pipeline 1: Ingesta estructurada batch (SQL Server)

- **Bronze**: `02_ingest_smartpool.py` lee vía JDBC e implementa ingesta incremental con un checkpoint Delta en `s3a://spark/medallion/_state`.
- **Silver**: `03_silver_smartpool.py` limpia y consolida la última versión por clave usando `updated_at`.
- **Gold**: `04_gold_smartpool.py` enriquece tareas de mantenimiento con datos de piscinas y calcula una estimación de costes.

Las tablas origen son `pools_dim` y `maintenance_events`, con las que se construye el historial operativo y de mantenimiento sobre el que se apoyan las capas Silver y Gold.

# 4. Pipeline 2: Ingesta semiestructurada batch (CSV electricidad)

Los CSV se generan con el productor `producer_electricity_csv.py` y se guardan en MinIO en `landing/electricity_prices/date=YYYY-MM-DD`. El pipeline:

- **Bronze/Silver**: `05_ingest_electricity_csv.py` normaliza tipos, preserva `ingest_date` y particiona por `date`.
- **Gold**: `06_gold_electricity_enrichment.py` genera estadísticas diarias y horas pico.

Este flujo se centra en disponer de series diarias de precios fiables para cruzarlas después con el consumo estimado de cada piscina.

# 5. Pipeline 3: Ingesta streaming (Kafka)

El productor `producer_smartpool_sensors.py` envía eventos JSON al tópico `smartpool-sensors`. El job `07_kafka_smartpool_sensors.py` aplica:

- Lectura de Kafka con `startingOffsets=latest`.
- Parseo de JSON a schema, **watermark** y ventanas de 1 minuto.
- Escritos en Bronze/Silver y agregados Gold por ventana.
- Enriquecimiento con `pools_dim` (batch) para contexto.

El streaming se orienta a disponer de señales casi en tiempo real para generar agregados operativos y alertas tempranas.

# 6. Orquestación en Airflow

Se han definido tres DAGs:

- **dag_10_smartpool_structured_batch**: ingesta SQL Server y transformaciones Silver/Gold. Se programa con cron `0 0 * * *` y lanza el DAG de electricidad con `TriggerDagRunOperator`.
- **dag_20_electricity_semi_batch**: pipeline de CSV. Incluye bifurcación con `BranchPythonOperator` y usa **XCom** (task `pick_ingest_date`) para pasar la fecha a los jobs Spark.
- **dag_30_sensors_streaming**: streaming Kafka. Usa `@hourly` como macro temporal y un `ExternalTaskSensor` para esperar la disponibilidad de datos batch.

El encadenamiento entre DAGs y el uso de sensores permite coordinar la llegada de datos batch antes de ejecutar el streaming enriquecido.

# 7. Planificación temporal

La ejecución batch estructurada se programa diariamente con cron (cada día a las 00:00h). El pipeline de electricidad se activa desde el DAG estructurado para mantener coherencia en la fecha de ingesta. El streaming se ejecuta de forma periódica (macro `@hourly`) con ventanas de 1 minuto y watermark de 2 minutos, dejando los agregados listos para análisis operativo y alertas.

## 8. Retos encontrados y soluciones

Algunos de los problemas encontrados son:

- **Precisión de timestamps**: en SQL Server los tiempos se almacenaban con 7 decimales, mientras que en Delta se leían con 6, lo que hacía que algunas filas quedaran fuera en filtros incrementales. Se ajustaron conversiones y comparaciones para evitar pérdidas.
- **Ingesta incremental**: la tabla `pools_dim` no tenía inicialmente `updated_at`, lo que impedía la incrementalidad. Se incorporó ese campo y se habilitó el checkpoint en `_state`.
- **Particiones y configuración**: se revisaron particiones y parámetros de Spark para equilibrar rendimiento y consumo de recursos, evitando bloqueos cuando Airflow y Spark coincidían en ejecución.
- **Dependencias Spark (packages/jars)**: al usar `configure_spark_with_delta_pip` no se cargaban correctamente los extra jars (por ejemplo el token provider de Kafka), lo que llevó a intentar resolverlo con Ivy. Finalmente se solucionó centralizando jars locales y configuraciones.
- **Recursos en Docker**: se sufrieron caídas y lentitud al ejecutar varios procesos a la vez. Para solucionarlo, se ajustó el paralelismo de Airflow (`AIRFLOW__CORE__PARALLELISM`, `MAX_ACTIVE_TASKS_PER_DAG`, `MAX_ACTIVE_RUNS_PER_DAG`) y se limitaron recursos de Spark desde `docker-compose.yml` (`SPARK_EXECUTOR_MEMORY`, `SPARK_EXECUTOR_CORES`, `SPARK_EXECUTOR_INSTANCES`, `SPARK_DRIVER_MEMORY`, `SPARK_CORES_MAX`).
- **Sincronización entre DAGs**: ajustes en sensores para alinear fechas lógicas y garantizar ejecuciones encadenadas.

Estas correcciones permitieron estabilizar los DAGs y asegurar que los tres pipelines funcionasen de forma natural.

## 9. Cumplimiento de requisitos

- **1.1 Macro temporal**: `dag_30` usa `@hourly`.
- **1.3 Cron**: `dag_10` usa `0 0 * * *`.
- **2.1 Bifurcación**: `BranchPythonOperator` en `dag_20`.
- **2.3 XCom/Taskflow**: `pick_ingest_date` pasa `ds` a tareas Spark.
- **3.1 Sensor**: `ExternalTaskSensor` en `dag_30`.
- **3.2 Push/Pull entre DAGs**: `TriggerDagRunOperator` de `dag_10` a `dag_20`.
- **PMD Medallion**: tres pipelines (batch estructurado, batch CSV y streaming Kafka) con Delta en Bronze/Silver/Gold.

## 10. Trabajo futuro

Como líneas de evolución, se plantea integrar fuentes externas de contexto (AEMET u Open-Meteo) para correlacionar clima y calidad del agua, ampliar el modelo energético con tarifas reales y añadir paneles de visualización y alertas.

También se podría incorporar control de calidad automático y modelos predictivos para anticipar incidencias en piscinas con mayor carga de uso.