BSG Institute	**Programa Certified AI Data Engineer** **Curso Procesos ETL para Workloads de AI** **Sesión 3 – 10/14/2025** **Instructor: Andres Felipe Rojas Parra**

Sesión 3 – Laboratorio Práctico ETL con Apache Airflow y Apache Spark

Este documento guía paso a paso la implementación práctica de los conceptos vistos en la Sesión 3 del curso 'Procesos ETL para Workloads de AI'. Se trabajará con Apache Airflow para la orquestación de pipelines y con Apache Spark para el procesamiento distribuido de datos.

Conceptos clave

- Airflow es un **sistema de orquestación de workflows** que permite programar, monitorear y escalar pipelines ETL.
- Basado en **DAGs (Directed Acyclic Graphs)** → cada nodo representa una tarea y las aristas definen dependencias.
- Ideal para **automatizar procesos de AI/ML pipelines**, como:
  - Ingesta de datos (desde APIs, S3, BigQuery, etc.)
  - Preprocesamiento (Pandas, Spark)
  - Entrenamiento de modelos ML
  - Despliegue y monitoreo

Módulo	Objetivo/Tema	Archivo(s)	Rol principal	Comentarios de uso
Módulo 1	Instalación y configuración de **Apache Airflow**	scripts/init_airflow.sh	**Bootstrap** de Airflow	Crea venv, instala reqs, inicializa DB, crea usuario y levanta webserver/scheduler.
Módulo 1	Instalación y configuración de **Apache Airflow**	airflow/requirements.txt	Dependencias	Versiona Airflow y providers (incluye apache-airflow-providers-apache-spark).
Módulo 3	Diseño de **DAGs** en Airflow	airflow/dags/etl_pipeline_demo.py	**DAG** de referencia	Define tareas extract → spark_transform → load, dependencias y schedule.
Módulo 4	Programación y ejecución de tareas en Airflow	airflow/dags/etl_pipeline_demo.py	**Scheduling/operadores**	Se muestra trigger, retries, catchup, logs; integra SparkSubmitOperator.
Módulo 5	Introducción/instalación de **Apache Spark**	configs/spark-env.sh	**Entorno Spark**	Variables de entorno para Spark standalone (cores/memoria). Opcional en local.
Módulo 5	Introducción/instalación de **Apache Spark**	scripts/run_spark_local.sh	**Smoke test** Spark	Ejecuta spark-submit en local para validar instalación sin Airflow.
Módulo 6	Transformación con **Spark SQL**	spark_jobs/transform_job.py	**Job PySpark**	Lógica de transformación (leer CSV, limpiar, agrupar, escribir Parquet).
Módulo 6	Transformación con **Spark SQL**	configs/spark-defaults.conf	**Tuning por defecto**	Particiones de shuffle, memoria, app name; se aplica con --properties-file.
Módulo 7	**Integración Airflow + Spark**	airflow/dags/etl_pipeline_demo.py	**Orquestación**	Tarea spark_transform llama a transform_job.py con rutas/args.
Módulo 7	**Integración Airflow + Spark** (opcional con contenedores)	docker/docker-compose.yml	**Stack en Docker**	Levanta Spark (master/worker) + Airflow; monta airflow/ y spark_jobs/.

Revisar este procedimiento una vez se tenga toda la estructura y todos los archivos configurados:

Flujo de arranque rápido (local)

1. **Instala dependencias de Airflow** (en venv):

cd etl-ai-lab

bash scripts/init_airflow.sh

2. **Prepara datos de prueba**:

Coloca sales_data.csv en airflow/data/ (con columnas: region,revenue, etc.)

3. **Prueba Spark local**:

bash scripts/run_spark_local.sh

4. **Orquesta con Airflow**:

Abre http://localhost:8080, habilita el DAG etl_pipeline_demo y ejecútalo.
Verifica logs y salida en airflow/data/out_parquet/.

Módulo 1: Instalación y Configuración de Apache Airflow

Apache Airflow es una plataforma de orquestación de flujos de trabajo (workflows) que permite planificar, monitorear y ejecutar pipelines ETL. Cada pipeline se define como un DAG (Directed Acyclic Graph), donde cada nodo representa una tarea y las aristas definen las dependencias.

Paso 1. Crear entorno virtual y activar

```
python3 -m venv airflow_env
source airflow_env/bin/activate
```

Paso 2. Instalar Apache Airflow

```
pip install "apache-airflow==2.10.1" --constraint \
"https://raw.githubusercontent.com/apache/airflow/constraints-2.10.1/constraints-3.12.txt"
```

Paso 3. Inicializar base de datos y crear usuario

```
airflow db init
airflow users create --username admin --firstname Andres --lastname Rojas --role Admin --email admin@triskel.ai --password admin123
```

Paso 4. Iniciar servicios

```
airflow webserver --port 8080 &
airflow scheduler &
```

Luego abra la interfaz web en http://localhost:8080 con el usuario y contraseña definidos.

Paso 5. Ejecutar Script (scripts/init_airflow.sh):

```
#!/usr/bin/env bash
set -e
cd "$(dirname "$0")/.."

python3 -m venv .venv
```

```
source .venv/bin/activate
pip install -r airflow/requirements.txt

export AIRFLOW_HOME="$(pwd)/airflow"
airflow db init

airflow users create \
  --username admin \
  --firstname Andres \
  --lastname Rojas \
  --role Admin \
  --email admin@triskel.ai \
  --password admin123

# arrancar
airflow webserver --port 8080 &
sleep 5
airflow scheduler &
```

Que hace este archivo?

Qué es:
Script de **bootstrap** para dejar **Airflow** listo y corriendo rápido en local.

Qué hace:

- Crea/activa **virtualenv**, instala requirements.
- Define AIRFLOW_HOME, **inicializa DB**, crea **usuario Admin**.
- Levanta **webserver** y **scheduler**.

Cuándo corre:
Al inicio del entorno (primera vez o cuando reinstales).

Qué personalizar:

- **Puertos**, usuario/contraseña, requirements.txt.
- AIRFLOW_HOME (si quieres otro path).
- Puedes añadir la creación de **Connections** (e.g., spark_default) por CLI.

Errores comunes:

- No tener Python/venv o compilar paquetes nativos (necesitas toolchain).
- AIRFLOW_HOME apuntando a un path sin permisos.

BSG Institute	**Programa Certified AI Data Engineer** **Curso Procesos ETL para Workloads de AI** **Sesión 3 – 10/14/2025** **Instructor: Andres Felipe Rojas Parra**

Modulo 2: Creación de la estructura

```
etl-ai-lab/
├─ airflow/                    # AIRFLOW_HOME
│  ├─ dags/
│  │  └─ etl_pipeline_demo.py
│  ├─ plugins/
│  ├─ include/                 # assets: SQL, Jinja, helpers
│  ├─ data/                    # data de entrada (pequeños CSV, etc.)
│  ├─ logs/
│  ├─ tests/                   # pruebas unitarias de DAGs
│  ├─ airflow.env              # vars de entorno (conn strings, etc.)
│  └─ requirements.txt
├─ spark_jobs/
│  ├─ transform_job.py
│  ├─ utils/                   # helpers PySpark
│  └─ data/                    # muestras grandes (si aplica)
├─ configs/
│  ├─ spark-defaults.conf
│  └─ spark-env.sh
├─ docker/
│  ├─ docker-compose.yml       # (opcional) stack Airflow + Spark
│  └─ .env
├─ scripts/
│  ├─ init_airflow.sh
│  └─ run_spark_local.sh
└─ README.md
```

Crear la estructura en Linux/Ubuntu

mkdir -p etl-ai-lab/{airflow/{dags,plugins,include,data,logs,tests},spark_jobs/utils,configs,docker,scripts}
touch etl-ai-lab/airflow/{airflow.env,requirements.txt}
touch etl-ai-lab/spark_jobs/transform_job.py
touch etl-ai-lab/configs/{spark-defaults.conf,spark-env.sh}
touch etl-ai-lab/docker/docker-compose.yml
touch etl-ai-lab/README.md

Crear la estructura con powershell

New-Item -ItemType Directory etl-ai-lab | Out-Null

```
New-Item -ItemType Directory etl-ai-lab\airflow,dags,plugins,include,data,logs,tests -Force
| Out-Null
New-Item -ItemType Directory etl-ai-lab\spark_jobs,utils -Force | Out-Null
New-Item -ItemType Directory etl-ai-lab\configs -Force | Out-Null
New-Item -ItemType Directory etl-ai-lab\docker -Force | Out-Null
New-Item -ItemType Directory etl-ai-lab\scripts -Force | Out-Null

New-Item etl-ai-lab\airflow\airflow.env -ItemType File | Out-Null
New-Item etl-ai-lab\airflow\requirements.txt -ItemType File | Out-Null
New-Item etl-ai-lab\spark_jobs\transform_job.py -ItemType File | Out-Null
New-Item etl-ai-lab\configs\spark-defaults.conf -ItemType File | Out-Null
New-Item etl-ai-lab\configs\spark-env.sh -ItemType File | Out-Null
New-Item etl-ai-lab\docker\docker-compose.yml -ItemType File | Out-Null
New-Item etl-ai-lab\README.md -ItemType File | Out-Null
```

Contenido mínimo de archivos clave

airflow/requirements.txt

```
apache-airflow==2.10.1
apache-airflow-providers-common-sql
apache-airflow-providers-http
apache-airflow-providers-apache-spark
pyspark==3.5.2
```

airflow/airflow.env (variables de entorno útiles)

```
AIRFLOW__CORE__LOAD_EXAMPLES=False
AIRFLOW__CORE__DAGS_ARE_PAUSED_AT_CREATION=True
AIRFLOW__CORE__EXECUTOR=LocalExecutor
AIRFLOW__CORE__FERNET_KEY=
AIRFLOW__CORE__SQL_ALCHEMY_CONN=sqlite:////absolute/path/to/etl-ai-
lab/airflow/airflow.db

# Conexión lógica a Spark (usada por SparkSubmitOperator)
AIRFLOW_CONN_SPARK_DEFAULT=spark://spark-master:7077
```

Módulo 3: Diseño de DAGs en Airflow

Un DAG (Directed Acyclic Graph) representa un flujo de tareas ETL donde cada tarea se ejecuta de acuerdo con dependencias definidas. Airflow utiliza Python para definir estos DAGs.

Ejemplo de DAG ETL simple (airflow/dags/etl_pipeline_demo.py)

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from airflow.providers.apache.spark.operators.spark_submit import SparkSubmitOperator
from datetime import datetime, timedelta
```

```
import os

BASE_DIR = os.path.abspath(os.path.join(os.path.dirname(__file__), ".."))
SPARK_JOB = os.path.abspath(os.path.join(BASE_DIR, "..", "spark_jobs", "transform_job.py"))

def extract():
    # Ejemplo: aquí podrías descargar un CSV o consultar un API
    print("Extrayendo datos... (placeholder)")

def load():
    # Ejemplo: subir a DW/S3/Parquet
    print("Cargando datos... (placeholder)")

default_args = {
    "owner": "andres",
    "retries": 1,
    "retry_delay": timedelta(minutes=2),
    "start_date": datetime(2025, 10, 14),
}

with DAG(
    dag_id="etl_pipeline_demo",
    default_args=default_args,
    schedule_interval="@daily",
    catchup=False,
    description="Pipeline ETL Airflow + Spark (demo)"
) as dag:

    t_extract = PythonOperator(task_id="extract", python_callable=extract)

    t_transform = SparkSubmitOperator(
        task_id="spark_transform",
        application=SPARK_JOB,
        # Si no usas conexión, puedes forzar master local:
        conn_id=None,
        application_args=[
            "--input", os.path.join(BASE_DIR, "data", "sales_data.csv"),
            "--output", os.path.join(BASE_DIR, "data", "out_parquet")
        ],
        packages="", # extra jars si fueran necesarios
        verbose=True
    )

    t_load = PythonOperator(task_id="load", python_callable=load)
```

t_extract >> t_transform >> t_load

Que hace este script?

Qué es:
El **DAG** (Directed Acyclic Graph) principal de Airflow que **orquesta** el pipeline ETL.

Qué hace:
- Define 3 tareas en orden:
  1. extract (placeholder: preparar/descargar datos)
  2. spark_transform (**ejecuta Spark** vía SparkSubmitOperator)
  3. load (placeholder: subir resultados a DW/S3/Parquet)
- Programa la ejecución (e.g., @daily) y establece políticas de reintento, start_date, etc.

Cuándo corre:
Cuando habilitas el DAG en la UI de Airflow o lo disparas manualmente (Trigger).

Qué personalizar:
- **schedule_interval**: frecuencia real de tu ETL.
- **Rutas** de SPARK_JOB, --input, --output.
- **Dependencias**/tareas reales (ingesta desde API/DB, validaciones, calidad de datos).
- **Conexión Spark** (si usas conn_id o modo local).

Errores comunes:
- Airflow no encuentra el archivo del job Spark (rutas relativas vs absolutas).
- Falta de permisos/paquetes en el entorno de Airflow.

Que hace el flujo?

Etapa	Descripción	Elementos técnicos
1. Definición de DAG	Se crea un grafo acíclico con tareas dependientes.	DAG() con dag_id, default_args, schedule_interval.
2. Funciones de tarea (extract, load)	Funciones Python sencillas que Airflow ejecutará como tareas independientes.	PythonOperator(python_callable=extract)
3. Tarea de transformación con Spark	Ejecuta un job distribuido de PySpark.	SparkSubmitOperator(application=...)
4. Flujo de dependencias	Se indica el orden de ejecución.	extract_task >> spark_task >> load_task
5. Programación (Scheduling)	Controla cuándo y cada cuánto se ejecuta el DAG.	@daily, @hourly, o cron (0 8 * * *)
6. Monitoreo	Airflow registra el estado y logs de cada tarea.	Web UI → Graph View / Tree View / Logs

Módulo 4: Programación y Ejecución de Tareas

Airflow permite programar tareas con expresiones cron o presets como @daily, @hourly, etc. Además, la interfaz gráfica permite visualizar dependencias y logs en tiempo real.

Integración con Spark

BSG Institute	**Programa Certified AI Data Engineer** **Curso Procesos ETL para Workloads de AI** **Sesión 3 – 10/14/2025** **Instructor: Andres Felipe Rojas Parra**

Conceptos clave

¿Qué es una "tarea" en Airflow?
Una **tarea (task)** representa **una unidad de trabajo atómica** dentro de un pipeline ETL.

Puede ser:
- Una función Python (PythonOperator)
- Un script SQL (SQLExecuteOperator)
- Un job Spark (SparkSubmitOperator)
- Una llamada API (SimpleHttpOperator)
- Un contenedor Docker (DockerOperator)

Cada tarea:
- Tiene un **task_id** único.
- Puede **depender** de otras tareas.
- Se ejecuta en un orden definido en el **DAG**.
- Genera **logs** y **estado** (success, failed, skipped, retrying).

¿Qué es un scheduler?
El **scheduler** de Airflow es el proceso que:
- Lee los DAGs definidos en airflow/dags/.
- Calcula qué tareas deben ejecutarse según la **frecuencia programada**.
- Coordina la ejecución con el **executor** (LocalExecutor, Celery, Kubernetes, etc.).
- Marca los estados de ejecución en la base de datos interna (metastore).

Por eso siempre necesitas tener corriendo:
- airflow webserver &
- airflow scheduler &

Tipos de programación:

Frecuencia	Sintaxis	Ejemplo	Descripción
Cada hora	@hourly	Ejecuta cada hora completa	Ideal para tareas de monitoreo
Diario	@daily	00:00 UTC cada día	Típico para ETL batch
Semanal	@weekly	Lunes 00:00	Consolidación semanal
Cron	"0 8 * * *"	8:00 am cada día	Control preciso

Reintentos y alertas:

default_args = {

```
    'owner': 'andres',
    'retries': 3,
    'retry_delay': timedelta(minutes=5),
    'email': ['andres@triskel.ai'],
    'email_on_failure': True
}
```

Esto enviará correos si alguna tarea falla.

Integración con otros operadores

En este punto, el DAG puede extenderse fácilmente con:

Tipo de tarea	Operador	Uso típico
HTTP	SimpleHttpOperator	Llamar APIs REST o endpoints JSON
Bash	BashOperator	Ejecutar scripts shell
SQL	PostgresOperator	Ejecutar queries sobre una base de datos
Docker	DockerOperator	Correr contenedores específicos
Sensors	FileSensor, HttpSensor	Esperar eventos antes de continuar

Como ejemplo, tenemos:

```
from airflow.operators.bash import BashOperator

notify = BashOperator(
    task_id='notify_completion',
    bash_command='echo "Pipeline completado exitosamente"'
)

load_task >> notify
```

Módulo 5: Instalación y Uso de Apache Spark

Apache Spark es un motor de procesamiento distribuido en memoria, diseñado para analizar grandes volúmenes de datos. Funciona en modo local o en clústeres (YARN, Kubernetes, etc.).

Instalación básica (modo local)

```
sudo apt install openjdk-17-jdk -y
wget https://downloads.apache.org/spark/spark-3.5.2/spark-3.5.2-bin-hadoop3.tgz
tar -xvf spark-3.5.2-bin-hadoop3.tgz
sudo mv spark-3.5.2-bin-hadoop3 /opt/spark

echo 'export SPARK_HOME=/opt/spark' >> ~/.bashrc
```

```
echo 'export PATH=$SPARK_HOME/bin:$PATH' >> ~/.bashrc
source ~/.bashrc
spark-shell
```

Ejecutar:

configs/spark-env.sh (hay que elevar permisos con chmod +x)

```
export SPARK_WORKER_CORES=2
export SPARK_WORKER_MEMORY=3g
```

Que hace este archivo?

Qué es:
Script de **variables de entorno** de Spark (solo si usas modo standalone/cluster propio).

Qué hace:

- Ajusta recursos del **worker** (cores, memoria) y variables del entorno del cluster.

Cuándo aplica:
Cuando levantas un cluster Spark standalone y fuentes este archivo en los nodos.

Qué personalizar:

- SPARK_WORKER_CORES, SPARK_WORKER_MEMORY.
- Rutas de Java (JAVA_HOME), SPARK_HOME, etc.

Errores comunes:

- Olvidar dar permisos de ejecución (chmod +x).
- Inconsistencia entre lo declarado aquí y lo que pides en spark-submit.

scripts/run_spark_local.sh

```
#!/usr/bin/env bash
set -e
cd "$(dirname "$0")/.."

INPUT="airflow/data/sales_data.csv"
OUTPUT="airflow/data/out_parquet"

spark-submit \
  --conf spark.sql.shuffle.partitions=4 \
  --properties-file configs/spark-defaults.conf \
  spark_jobs/transform_job.py --input "$INPUT" --output "$OUTPUT"
```

Que hace este archivo?

Qué es:
Script rápido para **probar Spark** de forma **local** (sin Airflow).

Qué hace:
- Lanza spark-submit contra spark_jobs/transform_job.py con --input y --output.
- Aplica configs/spark-defaults.conf y spark.sql.shuffle.partitions.

Cuándo corre:
Para validar el job Spark **antes** de orquestarlo con Airflow.

Qué personalizar:
- Rutas de entrada/salida.
- Configs adicionales (--packages, --jars, --master local[*] o cluster).

Errores comunes:
- Dataset inexistente.
- Conflictos de versión de PySpark vs Spark instalado.

Módulo 6: Procesamiento con Spark SQL

Spark SQL permite transformar y consultar grandes datasets usando una API tipo SQL. Es ideal para limpiezas, agregaciones y análisis distribuidos.

Ejemplo práctico

```
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg

spark = SparkSession.builder.appName("ETL Spark SQL").getOrCreate()
df = spark.read.csv("data/sales_data.csv", header=True, inferSchema=True)
df_clean = df.filter(col("revenue") > 0)
avg_sales = df_clean.groupBy("region").agg(avg("revenue").alias("avg_revenue"))
avg_sales.show()
avg_sales.write.mode("overwrite").parquet("output/avg_sales_by_region.parquet")
```

Ejemplo (spark_jobs/transform_job.py (job PySpark mínimo))

```
import argparse
from pyspark.sql import SparkSession
from pyspark.sql.functions import col, avg

def main(input_path: str, output_path: str):
    spark = SparkSession.builder.appName("ETL Spark SQL").getOrCreate()
    df = spark.read.csv(input_path, header=True, inferSchema=True)
    df_clean = df.filter(col("revenue") > 0)
    avg_sales = df_clean.groupBy("region").agg(avg("revenue").alias("avg_revenue"))
```

```
        avg_sales.show()
        #avg_sales.write.mode("overwrite").parquet("output/avg_sales_by_region.parquet"
)
        sales.coalesce(1).write.mode("overwrite").parquet(output_path)
        spark.stop()

if __name__ == "__main__":
   parser = argparse.ArgumentParser()
   parser.add_argument("--input", required=True)
   parser.add_argument("--output", required=True)
   args = parser.parse_args()
   main(args.input, args.output)
```

Que hace este script?

Qué es:
El **job PySpark** que realiza el **procesamiento distribuido** (la "T" del ETL).

Qué hace:
- Recibe parámetros --input y --output.
- Lee un CSV, filtra filas (revenue > 0), agrupa por region y calcula avg(revenue).
- Escribe salida en **Parquet** (particionado), con coalesce(1) para un archivo grande.

Cuándo corre:
Cuando lo lanza Airflow (tarea spark_transform) o manualmente con spark-submit.

Qué personalizar:
- **Transformaciones**: columnas, filtros, joins, funciones (UDFs) según tu caso.
- **Fuente/Destino**: CSV/JSON/Parquet, lectura de S3, Lakehouse, etc.
- **Optimización**: repartition, cache, broadcast, spark.sql.shuffle.partitions.

Errores comunes:
- Esquema/columnas no coinciden con el dataset real.
- Rutas de entrada/salida no existen o no tienen permisos.

configs/spark-defaults.conf

```
spark.app.name=etl-ai-lab
spark.sql.shuffle.partitions=4
spark.driver.memory=2g
spark.executor.memory=2g
```

Que hace este archivo?

Qué es:
Archivo de **configuración por defecto** para Spark.

Qué hace:
- Define parámetros globales para ejecutores/driver y SQL:
  - spark.app.name
  - spark.sql.shuffle.partitions (particiones de shuffle)
  - spark.driver.memory, spark.executor.memory, etc.

Cuándo aplica:
Al ejecutar spark-submit con --properties-file configs/spark-defaults.conf (o via env).

Qué personalizar:
- **Memoria** y **paralelismo** según tamaño de datos/nodo.
- Integraciones (S3/HDFS), spark.serializer, dynamicAllocation, etc.

Errores comunes:
- Subdimensionar memoria → OOM.
- Poner demasiadas particiones para datasets pequeños (overhead).

BSG Institute	**Programa Certified AI Data Engineer** **Curso Procesos ETL para Workloads de AI** **Sesión 3 – 10/14/2025** **Instructor: Andres Felipe Rojas Parra**

Módulo 7: Integración Airflow + Spark

Combinando ambas herramientas, Airflow puede orquestar la ejecución de tareas Spark, logrando un pipeline ETL completamente automatizado y escalable. Un flujo típico incluye:
1. Airflow ejecuta la tarea de ingesta.
2. Spark procesa y transforma los datos.
3. Airflow carga los resultados a un Data Warehouse o S3.

Beneficios:

- Automatización total del pipeline.
- Escalabilidad horizontal para Big Data.
- Integración directa con AI/ML pipelines.
- Monitoreo centralizado.

docker/docker-compose.yml (opcional)

```
services:
 spark-master:
  image: bitnami/spark:3.5
  environment:
   - SPARK_MODE=master
  ports: ["7077:7077","8081:8080"]

 spark-worker:
  image: bitnami/spark:3.5
  environment:
   - SPARK_MODE=worker
   - SPARK_MASTER_URL=spark://spark-master:7077
  depends_on: [spark-master]

 airflow:
  image: apache/airflow:2.10.1
  environment:
   - AIRFLOW__CORE__LOAD_EXAMPLES=False
   - AIRFLOW__CORE__EXECUTOR=LocalExecutor
   - AIRFLOW_CONN_SPARK_DEFAULT=spark://spark-master:7077
  volumes:
   - ../airflow:/opt/airflow
   - ../spark_jobs:/opt/spark_jobs
  working_dir: /opt/airflow
  command: bash -lc "pip install -r requirements.txt && airflow db init && \
            airflow users create --username admin --firstname Andres --lastname Rojas \
            --role Admin --email admin@triskel.ai --password admin123 && \
            airflow webserver -p 8080 & airflow scheduler"
  ports: ["8080:8080"]
```

depends_on: [spark-master, spark-worker]

Qué es:
Stack **Docker** para levantar **Spark (master/worker)** y **Airflow** orquestando sobre ese cluster.

Qué hace:

- Servicio spark-master y spark-worker (standalone).
- Servicio airflow con variables clave y **montajes** de directorios:
  - Monta ../airflow (DAGs, logs, requirements) y ../spark_jobs.
  - Ejecuta: pip install -r requirements.txt, airflow db init, crea usuario, y levanta webserver+scheduler.
- Exporta puertos (p. ej., 8080 para Airflow, 7077 para Spark master).

Cuándo corre:
Cuando quieres un entorno reproducible con **todo en contenedores**.

Qué personalizar:

- **Versiones de imágenes** (Airflow/Spark).
- **Volúmenes** (paths locales).
- **Conexión** AIRFLOW_CONN_SPARK_DEFAULT → spark://spark-master:7077.
- Recursos (memoria/CPU) en Docker Desktop.

Errores comunes:

- Paths relativos mal montados (monta **desde** la raíz del proyecto).
- No exponer puertos necesarios.
- Diferencias de UID/GID que provoquen permisos en carpetas montadas (logs/dags).

BSG Institute	**Programa Certified AI Data Engineer** **Curso Procesos ETL para Workloads de AI** **Sesión 3 – 10/14/2025** **Instructor: Andres Felipe Rojas Parra**

Cómo se conecta todo el pipeline o el proceso?

1. **Airflow (DAG)** etl_pipeline_demo.py
   → define extract → **spark_transform** (llama spark-submit con transform_job.py) → load.
2. **Spark job** transform_job.py
   → lee sales_data.csv → limpia/agrega → escribe **Parquet**.
3. **Configs Spark** (spark-defaults.conf, spark-env.sh)
   → control de recursos y parámetros por defecto (local o cluster).
4. **Scripts**

- init_airflow.sh levanta y prepara Airflow.
- run_spark_local.sh prueba Spark sin Airflow (debug rápido).

5. **Docker Compose** (opcional)
   → te da **Airflow + Spark** en contenedores; el DAG llama al cluster Spark.

Cierre del Laboratorio

En este laboratorio el estudiante implementará un pipeline ETL completo: desde la instalación de Airflow y Spark hasta la ejecución distribuida de transformaciones con Spark SQL. El enfoque combina teoría, práctica y automatización real para entornos de Inteligencia Artificial.