Clase 208 — Pipelines ETL/ELT con Airflow

Parte: 5 — Ingeniería de Datos · Fuente: Reis & Housley Fundamentals of Data Engineering (O'Reilly, 2022) cap. 8 + Airflow docs 2.10+. Duración estimada: 80 min.

Clase 208 — Pipelines ETL/ELT con Airflow

> *Parte: 5 — Ingeniería de Datos · Fuente: Reis & Housley Fundamentals of Data Engineering (O'Reilly, 2022) cap. 8 + Airflow docs 2.10+. Duración estimada: 80 min.*

Objetivo

Escribir DAGs de Airflow 2.x con la API moderna (TaskFlow + @dag/@task decorators), entender la diferencia entre ETL (transform antes de cargar) y ELT (cargar al warehouse y transformar ahí), y orquestar un pipeline extract → load → transform → notify con retries, SLAs y backfill.

Resultados de aprendizaje

Al finalizar, el estudiante podrá:

- Escribir un DAG con TaskFlow API (@dag, @task) y entender el grafo resultante.
- Diferenciar ETL (clásico) de ELT (moderno, warehouse-first) y elegir según contexto.
- Configurar retries, SLAs, trigger rules (all_success, one_failed, none_failed), y XComs.
- Hacer backfill (airflow dags backfill) para reprocesar fechas históricas sin duplicar.
- Diagnosticar Task stuck in queued, Worker died, DAG not appearing con airflow dags list, logs, y airflow.cfg.

Temas

#	Tema	Por qué importa
1	ETL vs ELT — cuándo cada uno	El warehouse moderno cambió el default.
2	DAG = grafo dirigido acíclico	Vocabulario fundacional.
3	TaskFlow API vs Operators clásicos	Código más limpio en Airflow 2.x.
4	XComs — pasar data entre tasks	Su uso correcto y sus límites (no para GBs
5	Schedule + catchup + backfill	Reprocesar histórico sin duplicar.
6	Sensors, hooks, providers	Esperar eventos externos, conectar a siste

Definiciones y características

- ETL (Extract-Transform-Load): transformás en Python/Spark antes de cargar al warehouse. Patrón clásico cuando el warehouse era caro.
- ELT (Extract-Load-Transform): cargás raw al warehouse y transformás con SQL ahí. Patrón moderno (BigQuery/Snowflake/DuckDB son baratos para compute SQL).
- DAG: grafo de tareas con dependencias. En Airflow se define con @dag decorator + @task per tarea.
- TaskFlow API: introducida en Airflow 2.0. Devuelve valores de @task que se vuelven inputs de otras @task — Airflow infiere XComs automático.
- XCom (Cross-Communication): mecanismo para pasar pequeños valores entre tasks. Default backend: metadata DB. No usar para GBs — pasá referencias (paths S3) en su lugar.
- Catchup: si un DAG con schedule='@daily' se prende un lunes, ¿corre todos los días desde su start_date? Si catchup=True: sí (reprocesa retroactivo). Si False: arranca desde hoy. Default True —

fuente de sorpresas caras.
- Backfill: reprocesar fechas específicas. airflow dags backfill --start-date 2026-06-01 --end-date 2026-06-15 my_dag.
- Sensor: tarea que espera un evento externo (archivo en S3, fila en DB) con polling. reschedule mode evita ocupar worker slot mientras espera.
- Hook: wrapper sobre un cliente externo (S3Hook, PostgresHook). Lee credenciales desde Connections UI.
- Provider: paquete con operators/hooks/sensors para una integración (apache-airflow-providers-amazon, apache-airflow-providers-google).

Dataset / recursos

- Stack ejemplo: Airflow 2.10+ con docker-compose oficial.
- Pipeline target: extrae CSV → carga a DuckDB → transforma con SQL → publica métricas a Slack.
- Librerías: apache-airflow>=2.10, duckdb, pandas.

Ejercicios

1. DAG mínimo TaskFlow: 3 tasks: extract (descarga CSV), transform (limpia con pandas), load (escribe a DuckDB). Ver el grafo en /graph.
2. Schedule + catchup: schedule='@daily', start_date=days_ago(7), catchup=True. Verificá que Airflow crea 7 runs históricos. Cambiar a catchup=False y observar diferencia.
3. Retries + SLA: agregá retries=3, retry_delay=timedelta(minutes=2), sla=timedelta(minutes=10) al transform. Simulá falla con raise Exception("flaky") y observá reintento.
4. XCom: extract devuelve un dict pequeño (filename + row count). transform lo recibe como argumento (TaskFlow autoinjecta). Verificá en la UI tab "XCom".
5. Backfill: airflow dags backfill --start-date 2026-06-01 --end-date 2026-06-05 my_dag. Confirmá que se ejecutan los 5 días sin duplicar (gracias a idempotencia con execution_date como key).

Homework verificable

Repo con docker-compose Airflow + DAG que:

1. Corre cada hora (schedule='@hourly').
2. Extrae data de una API pública (ej. CoinGecko BTC price), la carga a DuckDB.
3. Transforma con SQL (run_sql task usando DuckDBHook).
4. Calcula métrica (avg price 24h) y la postea a Slack via SlackWebhookOperator.
5. SLA de 5 min en extract, retries=3.
6. README con docker-compose up, comandos kubectl, captura de la UI.

Criterio de aceptación: el DAG corre 24 veces seguidas (1 día) sin fallar, los datos en DuckDB son idempotentes (re-run para misma hora no duplica), Slack recibe el mensaje cada hora.

Errores comunes

Síntoma / mensaje	Causa y cómo arreglar
DAG no aparece en la UI	Sintaxis error en el archivo. Fix: python
Catchup explota: 365 runs creados al arran	catchup=True (default) con start_date de h

Task stuck in queued forever	Worker no levantó, o pool agotado. Fix: ai
XCom serialization error con DataFrame	XCom default es JSON; un DataFrame no es J
start_date en el futuro hace que no corra	El scheduler ignora DAGs con start_date >
Variables de entorno no llegan al worker	airflow.cfg o Docker compose con env vars

Preguntas frecuentes

¿Airflow, Prefect, Dagster, Mage — cuál elijo en 2026?

- Airflow: estándar de la industria, max madurez, syntax verbosa.
- Prefect 3: API Python moderna, hybrid execution (Clase 209).
- Dagster: data-aware (assets), mejor lineage.
- Mage: notebook-first.

Para empresas grandes / equipos data engineering serios: Airflow. Para equipos chicos / ML-focused: Prefect o Dagster son más rápidos.

¿ETL o ELT?

ELT cuando: warehouse moderno (BigQuery/Snowflake), data analyst con SQL, transformaciones cambian seguido. ETL cuando: data muy sensible (no podés cargar raw al warehouse), transformaciones complejas (geo, ML inference), warehouse caro. La industria converge a ELT con dbt para SQL transforms post-load.

¿XCom para pasar un DataFrame de 500 MB?

No. XCom default está en la metadata DB (Postgres/MySQL) — vas a saturarla. Pasá un path S3 / GCS por XCom, y cada task lee/escribe del storage. O configurá Custom XCom Backend con S3.

¿Tengo que aprender los Operators clásicos (PythonOperator, BashOperator)?

Para mantener DAGs viejos: sí. Para nuevos: TaskFlow API es mejor. Pero algunos casos (operators provider-specific como BigQueryInsertJobOperator) siguen siendo más limpios sin TaskFlow.

¿Cómo testeo un DAG?

Tres niveles: (1) import test: python my_dag.py no rompe. (2) unit test de tasks: extraer la lógica a funciones puras + tests. (3) integration: airflow dags test my_dag 2026-06-01 corre el DAG sin scheduler.

¿Airflow corre el Python de mi DAG en cada heartbeat?

Sí — el scheduler parsea todos los DAGs cada min_file_process_interval segundos. No pongas código pesado al top level (requests.get(...) en import time es un anti-pattern). Todo lo costoso va dentro de tasks.

Referencias

- Reis, J. & Housley, M. Fundamentals of Data Engineering (O'Reilly, 2022), cap. 8 — Queries, Modeling, and Transformation.
- Airflow docs 2.x — TaskFlow API.
- Awesome Apache Airflow.
- dbt-core — el complemento natural para ELT.
- Designing Data-Intensive Applications (Kleppmann, 2017) — fundamentos.

Siguiente clase

Clase 209 — Pipelines con Prefect o Dagster

Apéndice: notebook (primer bloque)

Notebook declarativo. Genera el DAG y el docker-compose. Para correrlo en vivo: docker-compose up y abrir la UI en localhost:8080.

```
import os, tempfile, shutil
from pathlib import Path
WORK = Path(tempfile.gettempdir()) / 'airflow_demo'
if WORK.exists(): shutil.rmtree(WORK)
(WORK / 'dags').mkdir(parents=True)
os.chdir(WORK)
print('cwd:', Path.cwd())
```

Archivos complementarios

- notebook.ipynb