Orquestación de flujos de datos con Airflow

Conocimientos necesarios

- Python
- Bash Basico
- SQL Basico

Temas

- Que es la orquestación de data pipelines o data workflows
- Por qué es un punto crucial en la ingeniería de datos
- Que es un DAG
- Que es Apache Airflow
- Cuáles son los componentes de Airflow
- Que es un Operador
- Que son los providers en Airflow
- Como orquestar procesos en GCP utilizando Airflow
- Que es un sensor
- Que es un branchOperator

Los Desafíos Iniciales en la Ingeniería de Datos

Datos, fuentes, arquitecturas, procesos y más datos

- Limpieza de datos
- Disponibilidad de los datos
- Estandarización de procesos
- Silos de información
- Cuellos de botella en el procesamiento
- Calidad de datos
- Integración de datos
- Integración de procesos

Nota: El 80% del esfuerzo de un equipo de data se enfoca en recopilar, limpiar y transformar los datos.
Sabemos que casi todos los procesos se pueden automatizar usando las habilidades adecuadas de planificación, diseño e ingeniería.

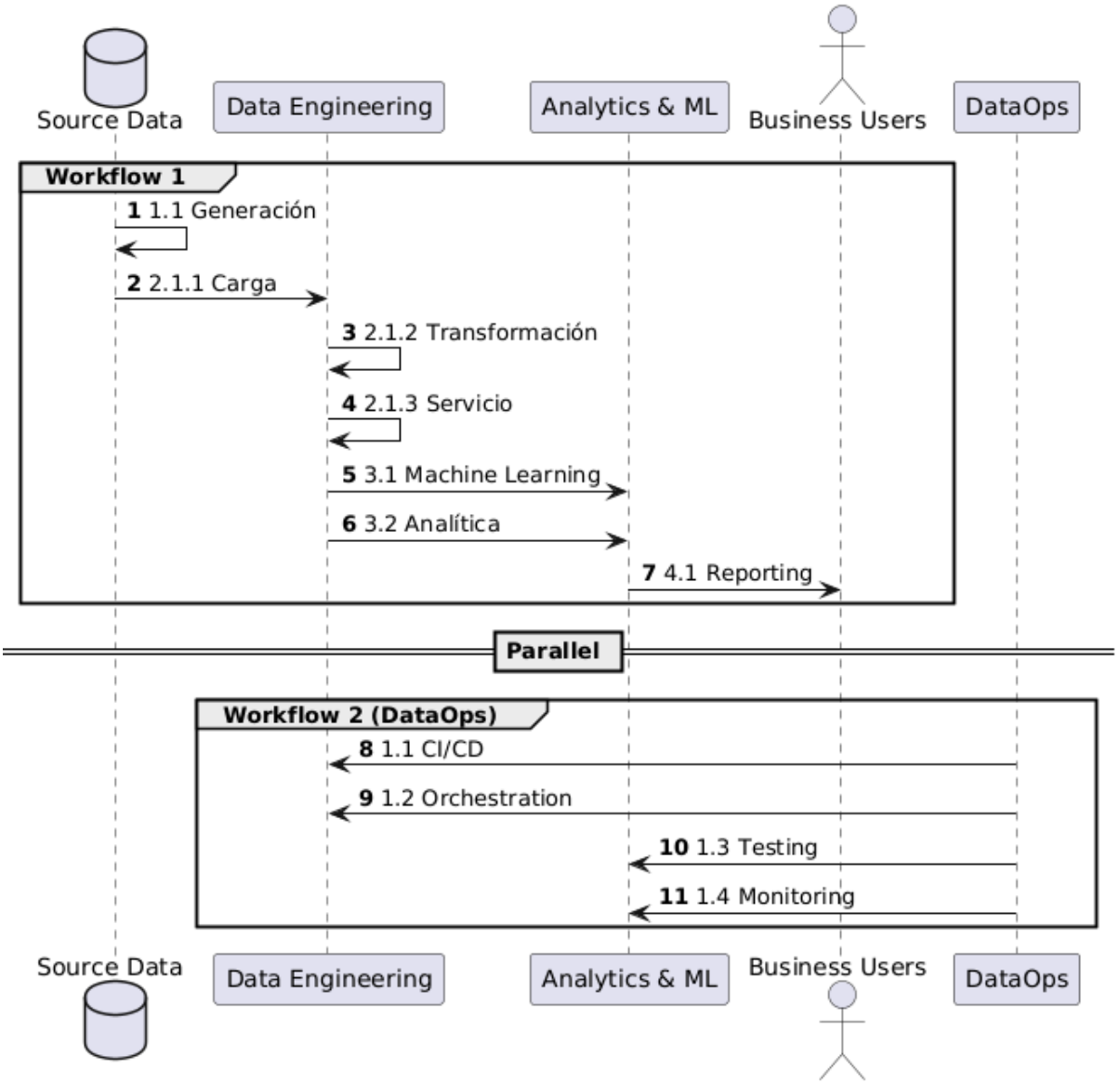
Flujo de vida de la ingenieria de datos

El proceso inicia con la **generación de datos desde las fuentes originales** (Source Data), que pueden incluir sensores, aplicaciones, transacciones u otros sistemas de captura. Una vez generados, los datos son canalizados hacia la etapa de **Data Ingestion y Data Engineering**, donde primero se almacenan de forma estructurada o semiestructurada. Esta etapa incluye subprocesos como la **carga** de datos al sistema de almacenamiento, la **transformación** de los mismos para adecuarlos a los modelos analíticos o de negocio, y su exposición a través de servicios o APIs que los disponibilizan a consumidores internos o externos.

Posteriormente, los datos procesados son utilizados por las áreas de **Data Analytics & Science**, que se encargan tanto de ejecutar modelos de **machine learning** como de realizar análisis descriptivos o predictivos que permitan generar conocimiento útil para la organización. Los resultados de estos

análisis son entregados a los **usuarios de negocio**, quienes acceden a ellos mediante herramientas de **reporting**, paneles de control o reportes automatizados, para facilitar la toma de decisiones.

En paralelo a todo este flujo, opera el dominio de **DataOps**, un conjunto de prácticas y herramientas orientadas a garantizar la confiabilidad, calidad y agilidad del proceso completo. DataOps interviene mediante mecanismos de **CI/CD** (integración y entrega continua), **orquestación** de pipelines de datos, **testing** automatizado para verificar la consistencia de los procesos, y **monitoreo** constante para detectar errores, anomalías o cuellos de botella en tiempo real. De este modo, DataOps actúa como un pilar transversal que respalda tanto al equipo de ingeniería como a los científicos de datos, asegurando la operación fluida y sostenible del sistema analítico.



Introducción a Apache Airflow

¿Qué es Apache Airflow?

Apache Airflow es un framework diseñado para la creación y monitoreo de data pipelines. Se destaca por su flexibilidad y su capacidad de programación en Python.

Historia de Apache Airflow

2015

Airbnb
Maxime Beauchemin creó una herramienta para orquestar procesos de datos en AirBnB llamada Airflow.

2016

Fundación Apache
Airflow se incorpora a la incubadora oficial de la Fundación Apache, marcando un hito significativo en su desarrollo y reconocimiento en la comunidad de código abierto.

2019

Máxima categoría
Airflow alcanza el estatus de proyecto de "máxima categoría" dentro de la Fundación Apache.

2020

Versión 2.0
La versión 2.0 se lanza al público, introduciendo nuevas funciones y mejoras significativas.

Funcionamiento

¿Para qué sirve?

Coordinación de Procesos de Datos: Airflow actúa como el conductor en el centro de sus procesos de datos. Coordina y programa la ejecución de tareas en sistemas distribuidos.

¿Para qué no sirve?

No es una herramienta de procesamiento: A diferencia de las herramientas de procesamiento de datos, Airflow no realiza la manipulación directa de los datos. En cambio, se encarga de orquestar los diversos componentes que procesan los datos en las canalizaciones.

Rol de Apache Airflow en la orquestación de canalizaciones de Datos

Características

Orquestación de componentes: Airflow coordina y programa componentes que son responsables del procesamiento de datos en las canalizaciones. Estos componentes pueden incluir extracción, transformación, carga (ETL), almacenamiento, y más.

Facilita la automatización: Al estar en el centro de la orquestación, Airflow permite la automatización de flujos de trabajo complejos, lo que ahorra tiempo y reduce la probabilidad de errores en la gestión de datos.

Apache Airflow NO es una herramienta de ETL, es una herramienta para la orquestación de flujos de datos.

Datapipelines como gráficos.

Características:

- Aspersión visual, permite comprender la lógica del proceso.
- Facilidad de la comprensión.
- Tomar decisiones informadas para optimizar nuestro flujo de trabajo.

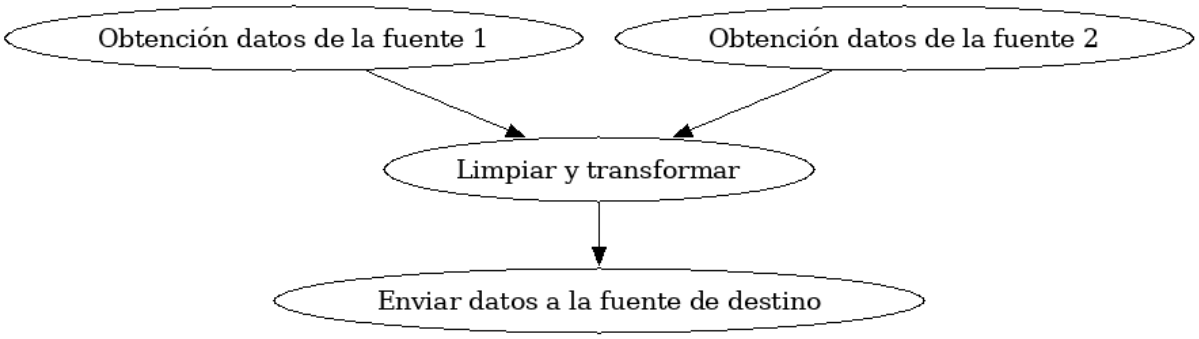
Los data pipelines consisten en varias tareas o acciones que deben ejecutarse para lograr un resultado esperado.

Ejemplo: Supongamos que queremos crear un pequeño dashboard de ventas que nos diga como se ven las ventas de la semana.

Pasos a seguir:

1. Obtener los datos de las ventas desde sus fuentes
2. Limpiar y/o transformar los datos para que se ajusten a nuestro propósito aplicando reglas de negocio
3. Enviar los datos transformados a una fuente donde la pueda leer el dashboard.

Gráficos Dirigidos



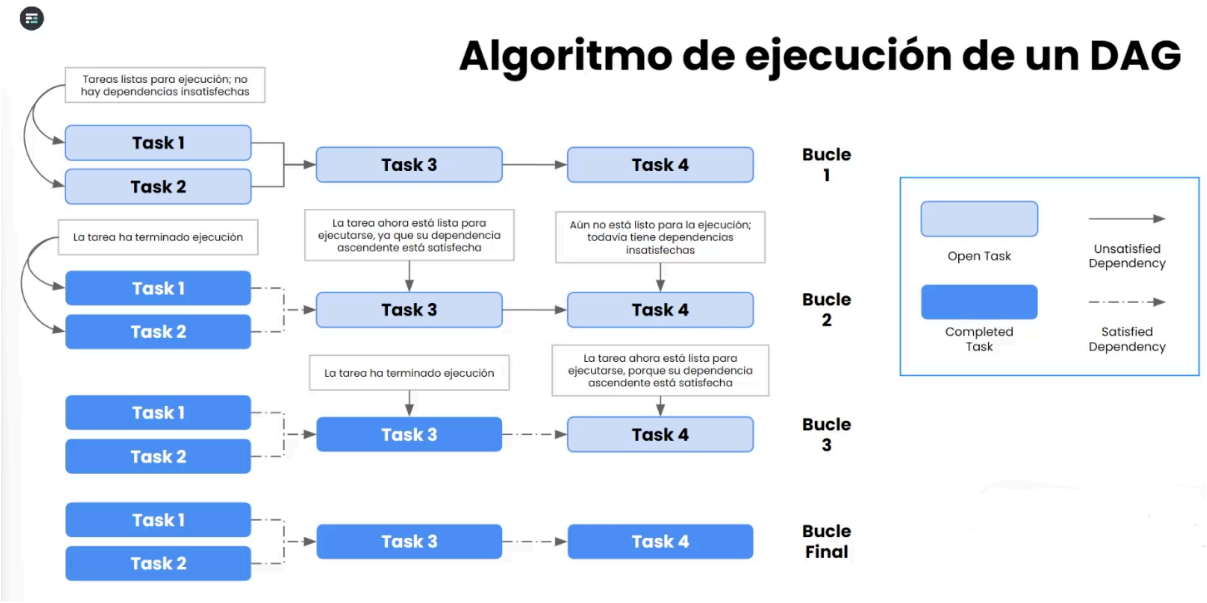
- Obtención datos de la fuente 1
- Obtención datos de la fuente 2
- Limpiar y transformar
- Enviar datos a la fuente de destino

Las flechas significan **task dependency.**

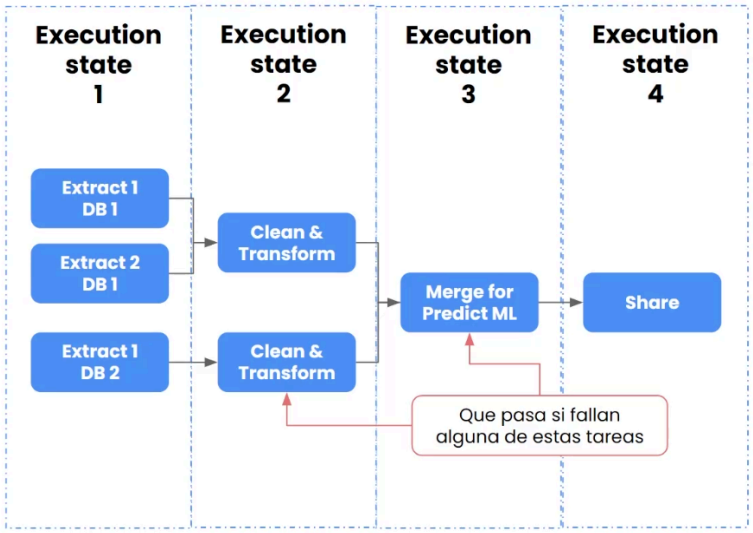
DAG

Gráfico acíclico dirigido

Un gráfico acíclico dirigido (DAG) es un gráfico que contiene bordes dirigidos y no tiene ciclos ni bucles, lo que significa que no hay dependencias circulares entre las tareas. Esta propiedad de ser acíclico es fundamental, ya que evita situaciones en las que una tarea depende de otra y viceversa, lo que se conoce como una dependencia circular.



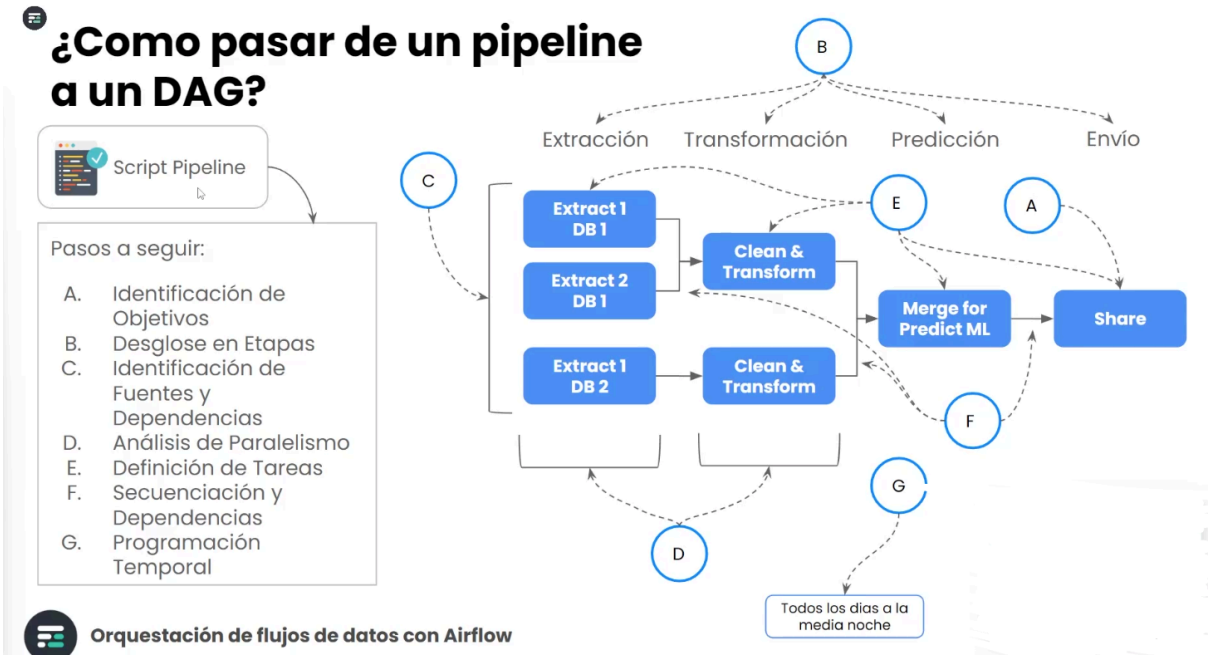
Caso de uso: Implementación de aprendizaje automático (ML)
Imaginemos un escenario en el que el propietario de una empresa busca utilizar el aprendizaje automático (ML) para mejorar la eficiencia de su operación. El objetivo es utilizar un modelo ML que correlacione las ventas de la empresa con los patrones de tráfico de personas en todas sus tiendas y así mejorar las promociones para vender más.



Ejemplo: Implementación de Aprendizaje Automático (ML)

¿Cómo pasar de un pipeline a un DAG?

Pasos a seguir:
A. Identificación de objetivos
B. Desglose en etapas
C. Identificación de fuentes y dependencias
D. Análisis de paralelismo
E. Definición de tareas
F. Secuenciación y dependencias
G. Programación temporal



Creación de DAGs

Caso: Sistema de recomendación de películas

La empresa "CineFlow" está desarrollando un sistema de recomendación personalizada de películas.

El proceso se inicia con la recopilación de datos, que provienen de dos fuentes principales: la base interna de registros de comportamiento de usuarios dentro de la plataforma y los datos extraídos de una API pública que proporciona críticas de películas y otros metadatos relevantes.

Una vez recopilados, estos datos se procesan mediante un modelo de aprendizaje automático alojado en la nube que se consume desde una API. Este ***modelo analiza los patrones de comportamiento de los usuarios, las preferencias de visualización y otros datos relevantes para generar recomendaciones personalizadas para cada usuario.***

Las recomendaciones resultantes se almacenan en una base de datos centralizada, que se actualiza mensualmente.

Por último, se implementa un sistema de notificación por correo electrónico para comunicar las recomendaciones a los usuarios.

Metodología
A. Identificación de objetivos
B. Desglose en etapas
C. Identificación de fuentes y dependencias
D. Análisis de paralelismo
E. Definición de tareas
F. Secuenciación y dependencias
G. Programación temporal

Identificación de tareas

Recopilación de datos

- DB interna **(paralela)**
- API RESTful **(paralela)**

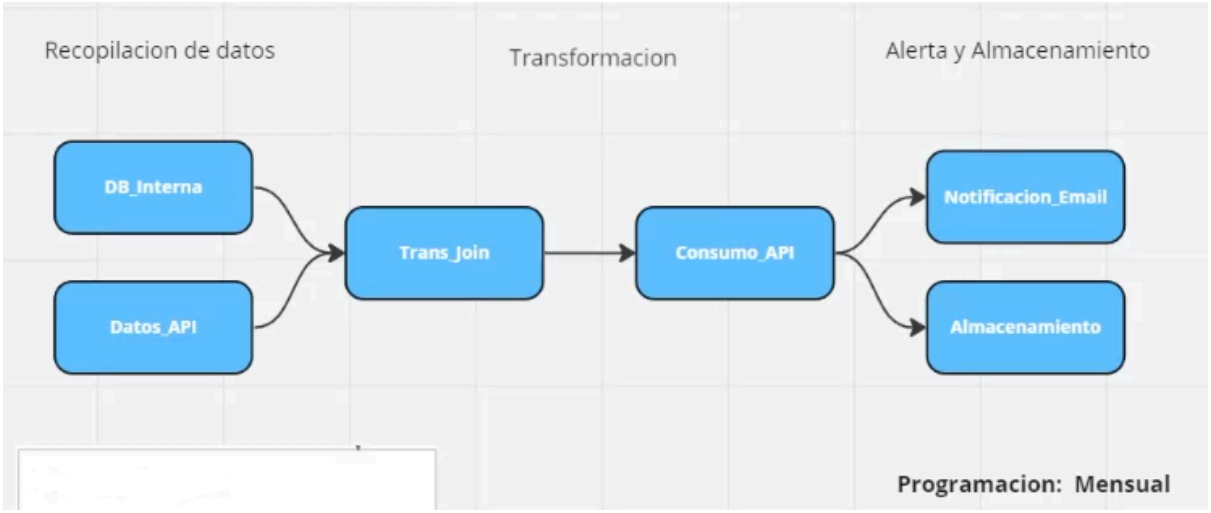
Transformación de datos

- Limpieza de los datos
- Consolidación de las bases de datos [TRANS_JOIN]
- Consumo de APIs

Alerta y almacenamiento

- Notificación de email **(paralela)**
- Almacenamiento de datos **(paralela)**

Programación: Mensual



Caso: Análisis de redes sociales para una marca de moda.

La reconocida marca de moda "ChicTrends" busca mejorar su presencia en redes sociales mediante un proceso estructurado de análisis y mejora continua. El flujo de datos comienza con la **extracción** de métricas de múltiples plataformas sociales, como Instagram, Twitter y Facebook. Estas métricas incluyen datos de interacciones de usuarios, como likes, comentarios, shares y otros indicadores relevantes para evaluar el rendimiento de las publicaciones y la percepción del público hacia la marca.

Posteriormente, se lleva a cabo un ***análisis de sentimientos utilizando un modelo de procesamiento de lenguaje natural (NLP) previamente entrenado***, que se encuentra alojado en el servidor de la empresa.

Los resultados de este análisis se almacenan en una base de datos centralizada, donde se realiza la actualización automática de tableros de métricas para el equipo ejecutivo, permitiéndoles monitorear de manera continua el desempeño de la marca en las redes sociales y tomar decisiones estratégicas basadas en datos.

NOTA: Este caso no tiene calendarización.

Metodología
A. Identificación de objetivos
B. Desglose en etapas
C. Identificación de fuentes y dependencias
D. Análisis de paralelismo
E. Definición de tareas
F. Secuenciación y dependencias
G. Programación temporal

Identificación de tareas

Extracción

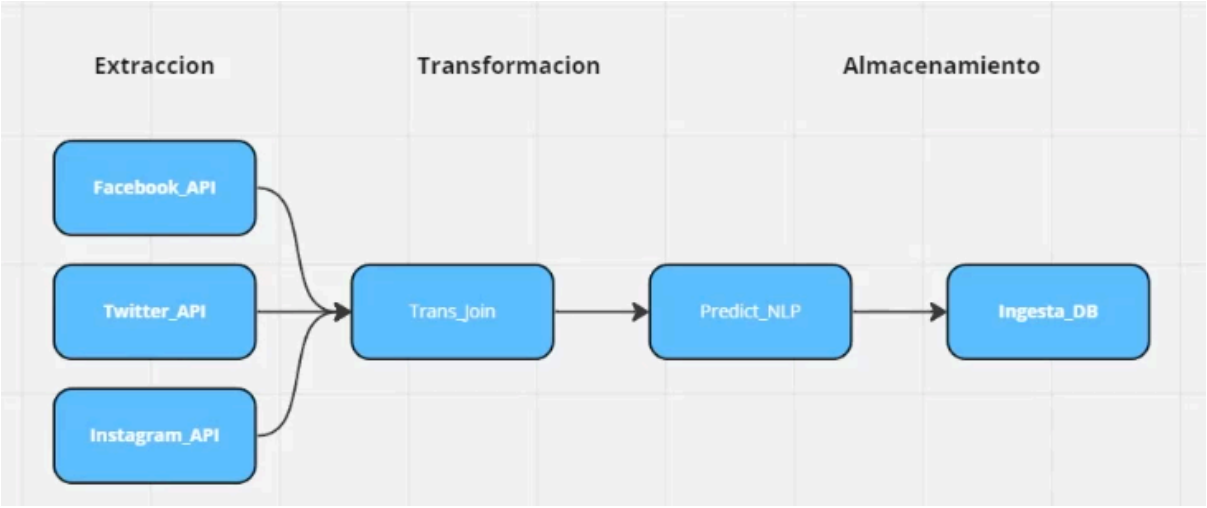
- Facebook API
- Twitter API
- Instagram API

Transformación

- Limpieza de los datos
- Consolidación de las bases de datos [TRANS_JOIN]
- Predict_NPL

Almacenamiento

- Ingesta de bases de datos



Programación: Por definir

Caso: Sistema de seguimiento de inventario para una tienda en línea.

La tienda en línea "E-Shopper" está buscando mejorar la gestión de su inventario para garantizar la disponibilidad de productos y optimizar sus operaciones.

Se lleva a cabo un proceso de reconciliación de inventario, donde se contrastan las ventas realizadas con las reposiciones y nuevas llegadas de productos para actualizar el estado actual del inventario. Durante este proceso, se utiliza un algoritmo de aprendizaje automático para ***predecir cuándo es***

probable que se agoten los productos disponibles en función de las tendencias de ventas y el ritmo de reposición.

Los resultados de este proceso se actualizan en una base de datos en la nube, lo que proporciona a "E-Shopper" una visión del estado de su inventario y de posibles problemas de escasez que puedan surgir.

Además, se implementa un sistema de alertas automáticas por correo electrónico para notificar al equipo de logística y gestión de inventario en caso de niveles críticos de existencias.

Metodología

A. Identificación de objetivos
B. Desglose en etapas
C. Identificación de fuentes y dependencias
D. Análisis de paralelismo
E. Definición de tareas
F. Secuenciación y dependencias
G. Programación temporal

Identificación de tareas:

Extracción

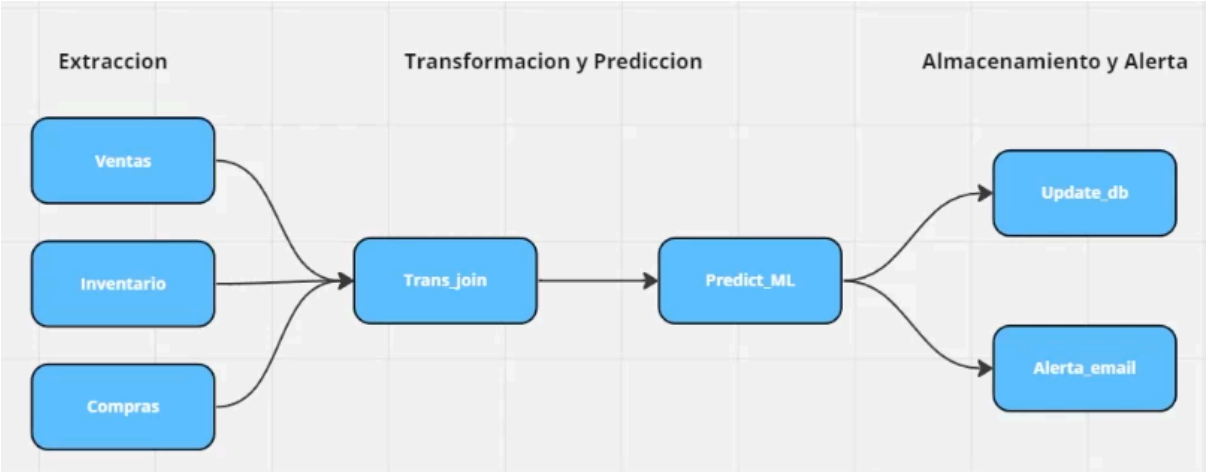
- Ventas
- Inventario
- Compras

Transformación

- Consolidación de las bases de datos [TRANS_JOIN]
- Predicción

Almacenamiento y alertas

- Actualizar base de datos.
- Alerta tipo email



Programación: Por definir

Caso: Monitoreo de sensores en una planta de manufactura.

En una planta de manufactura moderna, se implementa un sistema de monitoreo basado en sensores para supervisar el rendimiento de los equipos de producción. El flujo de datos comienza con la extracción de datos de los sensores distribuidos en la planta, los cuales se recopilan cada hora y se transmiten al sistema central para su procesamiento.

Una vez recopilados, los datos se someten a un ***análisis de calidad para identificar posibles problemas de producción o cualquier otra anomalía que pueda afectar la calidad o eficiencia del proceso de manufactura.***

Los resultados de este análisis se resumen automáticamente y se envían a los supervisores y al equipo de mantenimiento a través de correos electrónicos.

Además, se actualizan tableros específicos para los departamentos de producción y mantenimiento, donde se visualizan de manera clara y concisa los datos recopilados, los hallazgos del análisis de calidad y el estado actual de los equipos.

Metodología

A. Identificación de objetivos
B. Desglose en etapas
C. Identificación de fuentes y dependencias
D. Análisis de paralelismo
E. Definición de tareas
F. Secuenciación y dependencias
G. Programación temporal

Identificación de tareas:

Extracción

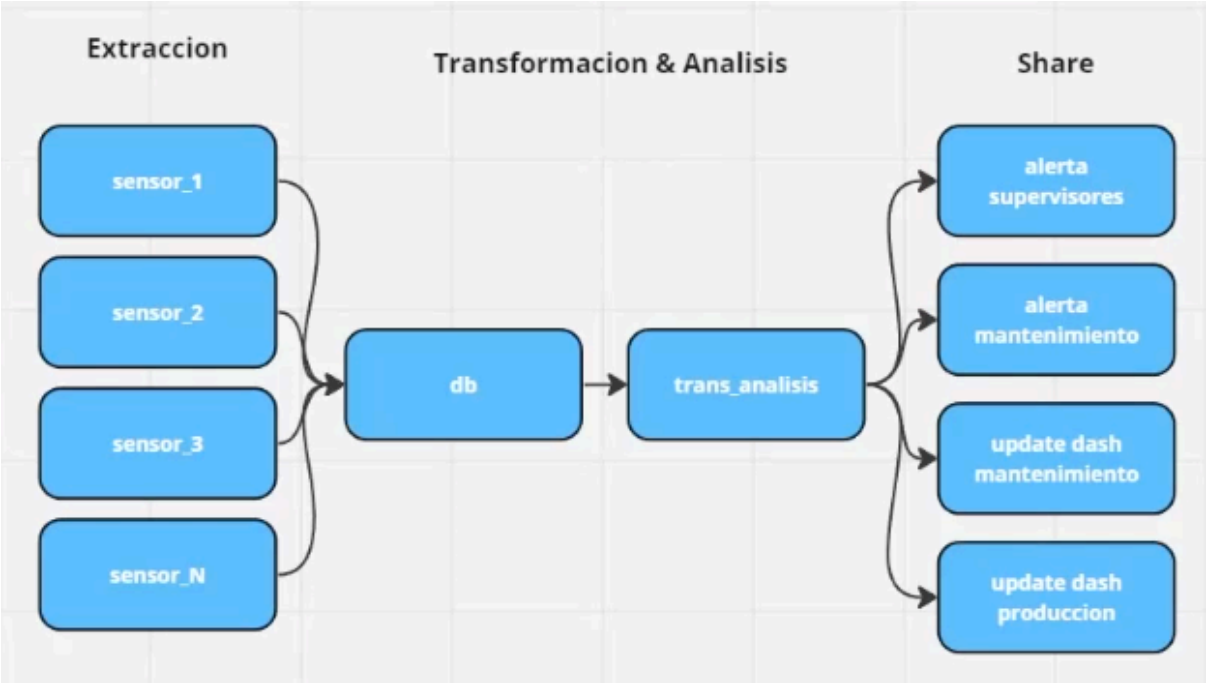
- Sensores

Transformación / Análisis

- Consolidación de las bases de datos [TRANS_JOIN]
- Análisis de calidad (basado en Business Rules)

Data sharing

- Alerta tipo email (Supervisores / Mantenimiento)
- Actualización de dashboards (Producción / Mantenimiento)



Programación: Cada hora

Componentes de Airflow – Docker compose

- airflow-scheduler
- airflow-webserver
- airflow-worker
- airflow-triggerer
- airflow-init
- postgres
- redis

Documentation:

- https://airflow.apache.org/docs/apache-airflow/stable/installation/index.html
- https://airflow.apache.org/docs/apache-airflow/stable/howto/docker-compose/index.html

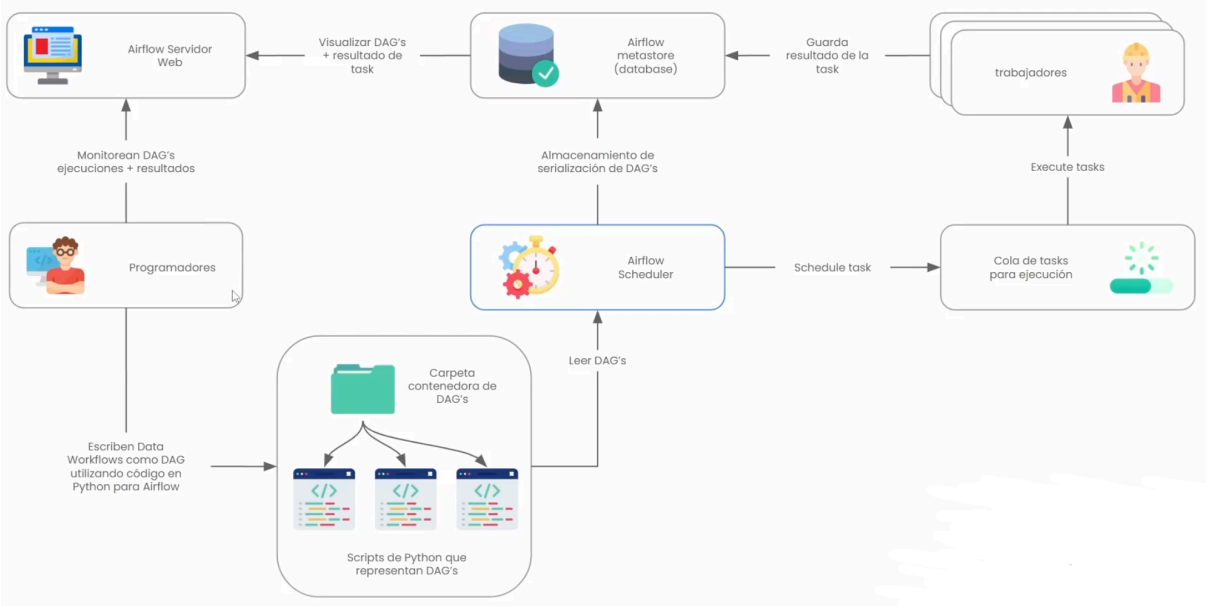
Commands

```
cd
mkdir airflow_codigofacilito
cd airflow_codigofacilito
curl -LfO
'https://airflow.apache.org/docs/apache-airflow/2.8.3/docker-compose.yaml'
mkdir -p ./dags ./logs ./plugins ./config
echo -e "AIRFLOW_UID=S(id -u)" > .env
cat .env
docker compose up airflow-init
docker compose up
```

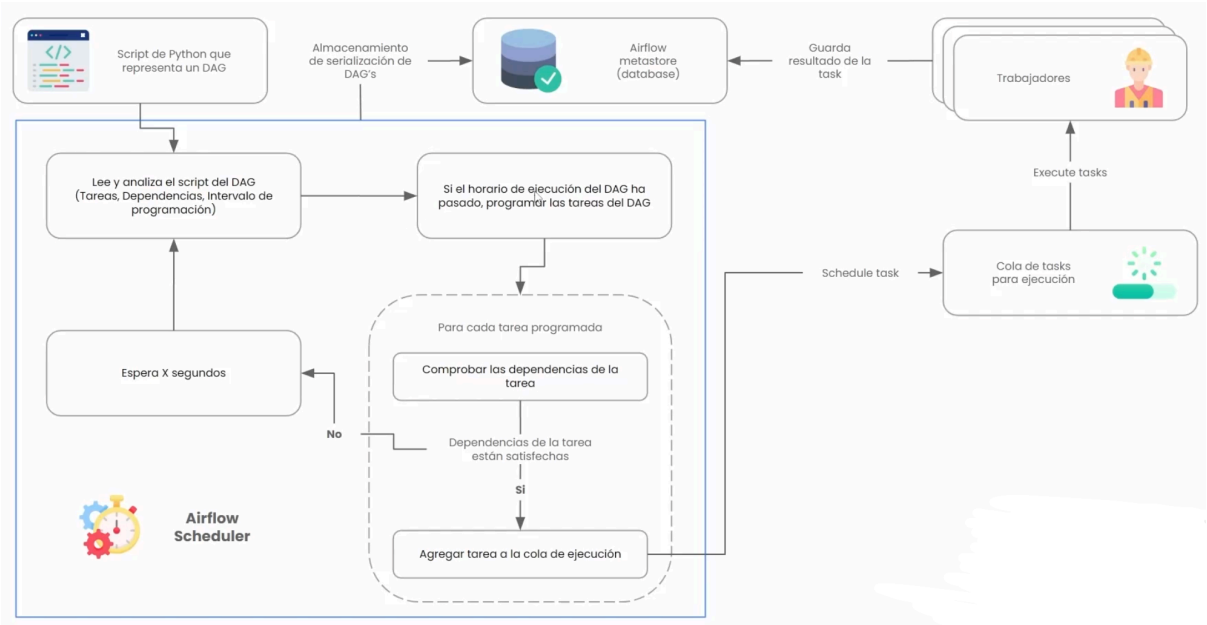
Programación y ejecución de DAGs en Airflow

Para comprender cómo Airflow lleva a cabo la ejecución de los DAG, es esencial examinar el proceso general que involucra el desarrollo y la ejecución en Airflow.
A un nivel más alto, Airflow se organiza en tres componentes fundamentales:

- **Programador de Airflow:** Verifica los intervalos de programación de los DAGs creados y guarda las tareas a gestionar en base de datos.
- **Trabajadores de Airflow:** Recogen las tareas programadas y las ejecutan.
- **Servidor web de Airflow:** Visualización de DAGs



DAG Management



Consideraciones y razones para elegir Airflow

- Flexibilidad en la implementación
- Ampliación e integración
- Semántica de programación enriquecida
- Funciones avanzadas como backfilling
- Interfaz web
- Código abierto y soluciones gestionadas

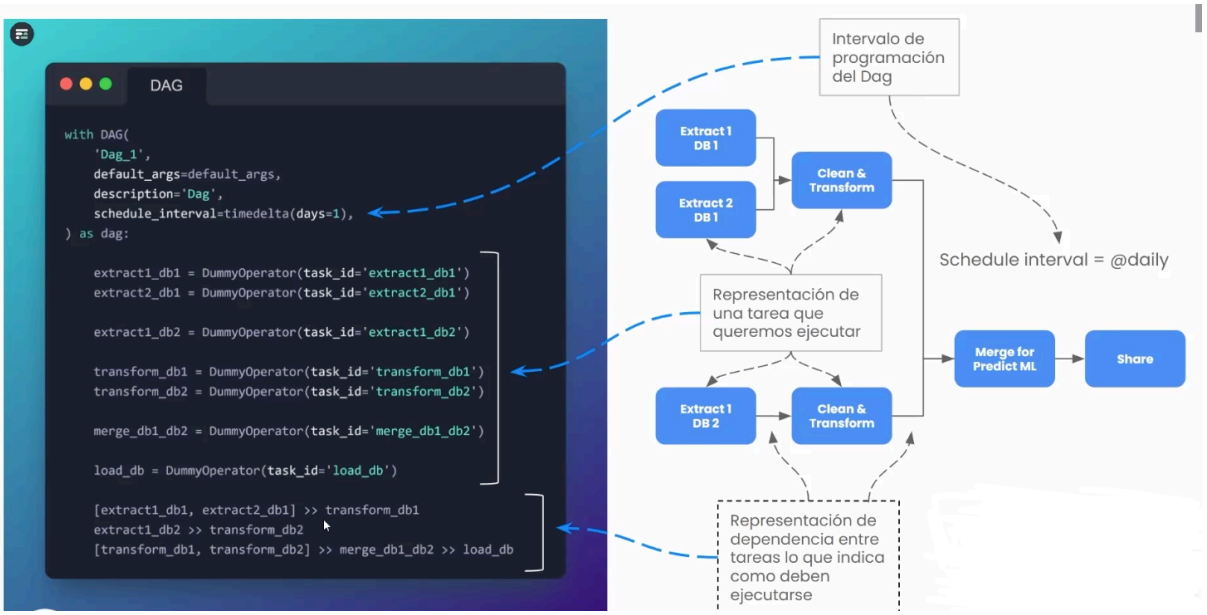
Consideraciones y razones para no elegir Airflow

- Cargas de trabajo de transmisión
- Canalizaciones altamente dinámicas
- Poca experiencia en programación (Python)
- Complejidad para grandes casos de uso
- Limitaciones en funcionalidades más amplias

Anatomía y desarrollo de un DAG con Python

```
from datetime import timedelta
from airflow import DAG
from airflow.operators.dummy import DummyOperator
from datetime import datetime

default_args = {
    'owner': 'airflow',
    'start_date': datetime(2024, 1, 1),
    'retries': 1,
    'retry_delay': timedelta(minutes=5),
}

with DAG(
    dag_id='dag_example',
    description='Dag',
    default_args=default_args,
    schedule_interval=timedelta(days=1), #Intervalo de programación del DAG, schedule
interval @daily
    catchup=False,
) as dag:

# Inicio: Representación de tareas que se desea ejecutar
    extract1_db1 = DummyOperator(task_id='extract1_db1') # Tarea extracción 1 [ a db1 ]
    extract2_db1 = DummyOperator(task_id='extract2_db1')  # Tarea extracción 2 [ a db1 ]
    extract1_db2 = DummyOperator(task_id='extract1_db2') # Tarea extracción 3 [ a db2 ]

    transform_db1 = DummyOperator(task_id='transform_db1') # Tarea transformación 1 [
sobre datos de  db1 ]
    transform_db2 = DummyOperator(task_id='transform_db2') # Tarea transformación 2 [
sobre datos de  db2 ]

    merge_db1_db2 = DummyOperator(task_id='merge_db1_db2') # Tarea de fusión de la
información entre dbs
    load_db = DummyOperator(task_id='load_db') # Tarea de carga de datos
# Final: Representación de tareas que se desea ejecutar
```

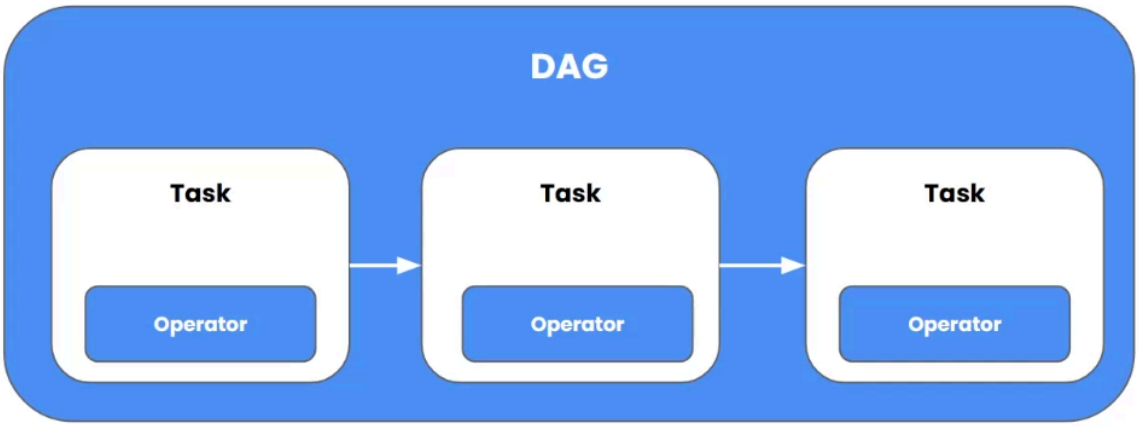
```
#Representación de dependencias entre tareas lo que indican como deben ejecutarse
    # Definir las dependencias
    [extract1_db1, extract2_db1] >> transform_db1
    extract1_db2 >> transform_db2
    [transform_db1, transform_db2] >> merge_db1_db2 >> load_db
```



Diccionario de DAG

```
default_args = {
    "owner": "airflow",  # Dueño del DAG
    "depends_on_past": True,  # Espera a que la ejecución anterior termine
exitosamente
    "email": ["airflow@example.com"],  # Lista de correos para notificaciones
    "email_on_failure": False,  # No enviar correos en caso de fallo
    "email_on_retry": False,  # No enviar correos en caso de reintento
    "retries": 1,  # Número de reintentos en caso de fallo
    "retry_delay": timedelta(minutes=2),  # Tiempo de espera entre reintentos
    "on_failure_callback": task_fail_slack_alert,  # Función callback si la
tarea falla
    "on_retry_callback": task_retry_slack_alert,  # Función callback si la tarea
es reintentada
    "on_success_callback": task_success_slack_alert,  # Función callback si la
tarea tiene éxito
    "tags": ["CF"]  # Etiquetas para el DAG
}
```

¿Qué es un operador en Apache Airflow?



- **Task**: Es una **unidad de trabajo** dentro de un DAG. Cada task ejecuta una operación específica (por ejemplo, extraer datos, transformarlos, cargarlos, etc.).

- **Operator**: Es una **plantilla que define qué hace un task**. Airflow incluye operadores como:

    - **PythonOperator:** ejecuta funciones Python.
    - **BashOperator:** ejecuta comandos de Bash.
    - **DummyOperator:** crea tareas vacías (para estructurar el flujo).
- Existen también operadores específicos.

Claro, aquí tienes ejemplos breves en Python de tres operadores comunes en **Apache Airflow**: PythonOperator, BashOperator y DummyOperator.

🐍 1. PythonOperator

Ejecuta una función Python.

```
from airflow import DAG
from airflow.operators.python import PythonOperator
from datetime import datetime

def say_hello():
    print("Hello from PythonOperator!")

with DAG(
    dag_id="python_operator_example",
    start_date=datetime(2023, 1, 1),
    schedule_interval=None,
    catchup=False
) as dag:

    python_task = PythonOperator(
```

```
        task_id="say_hello_task",
        python_callable=say_hello
    )
```

💻 2. BashOperator

Ejecuta un comando de shell o terminal.

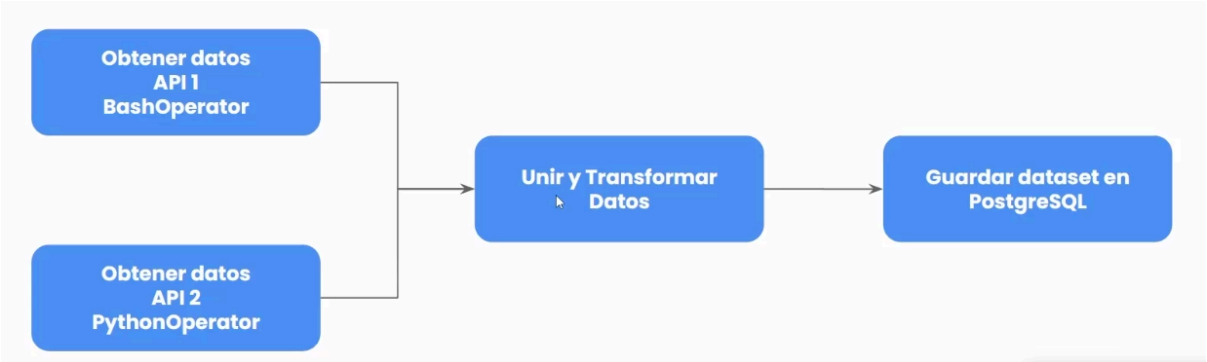
```
from airflow import DAG
from airflow.operators.bash import BashOperator
from datetime import datetime

with DAG(
    dag_id="bash_operator_example",
    start_date=datetime(2023, 1, 1),
    schedule_interval=None,
    catchup=False
) as dag:

    bash_task = BashOperator(
        task_id="print_date_task",
        bash_command="date"
    )
```

🪄 3. DummyOperator (Airflow < 2.5) or EmptyOperator (Airflow 2.5+)

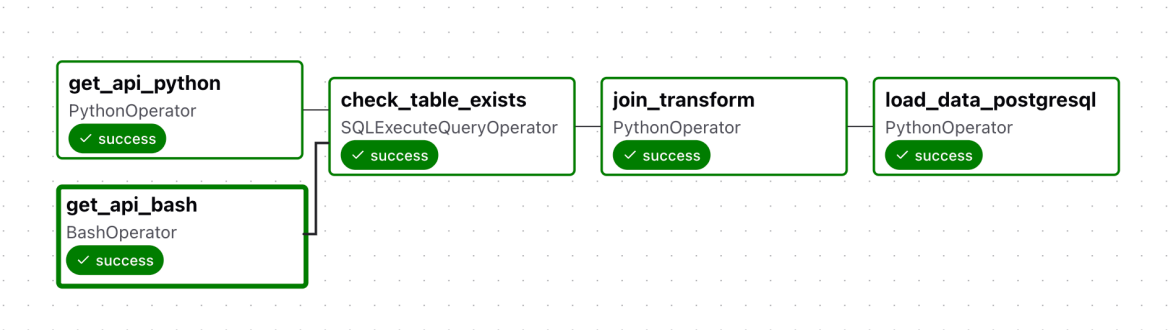
Se usa para marcar puntos en el flujo o para pruebas.

```
from airflow import DAG
from airflow.operators.empty import EmptyOperator  # Antes: DummyOperator
from datetime import datetime

with DAG(
    dag_id="dummy_operator_example",
    start_date=datetime(2023, 1, 1),
    schedule_interval=None,
    catchup=False
) as dag:

    dummy_task = EmptyOperator(
        task_id="placeholder_task"
    )
```

Ejemplo de DAG



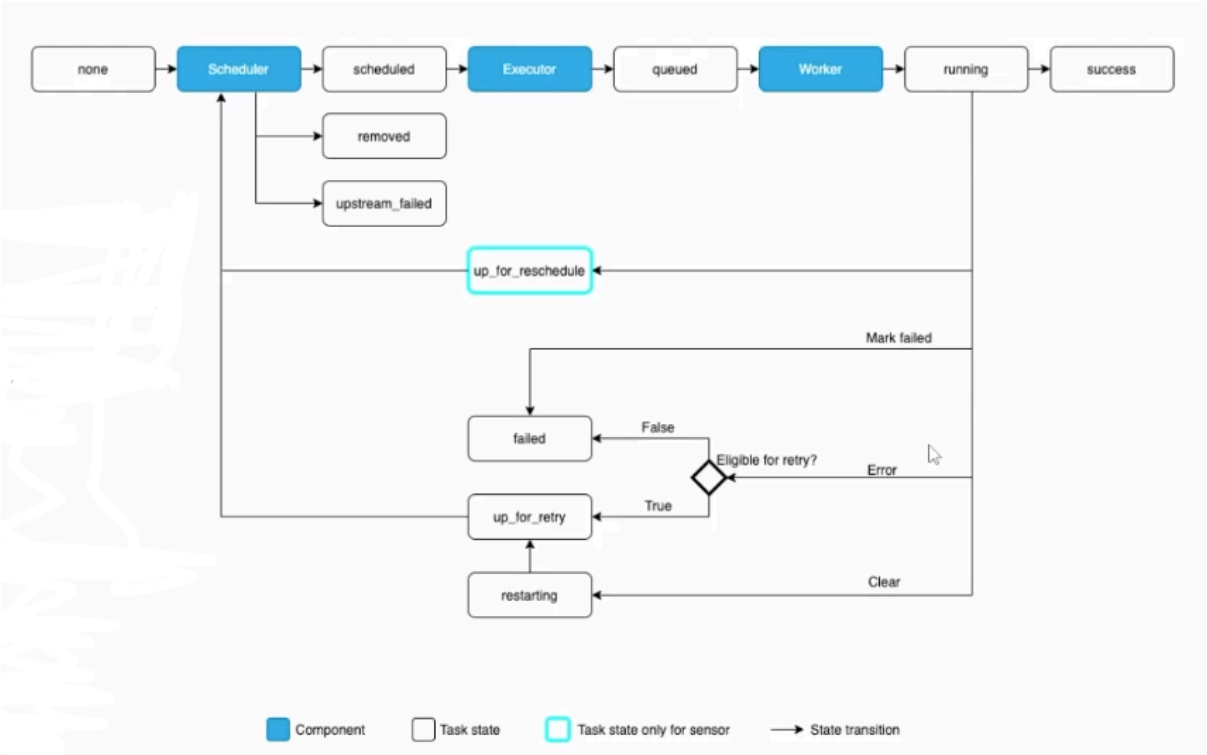
Evidencia de realización en Airflow:



¿Qué pasa si una tarea falla?

Estados de una tarea

- **none:** La tarea aún no se ha puesto en cola para su ejecución (sus dependencias aún no se cumplen).
- **scheduled:** El programador ha determinado que se cumplen las dependencias de la tarea y debe ejecutarse.
- **queued:** La tarea se ha asignado a un ejecutor y está esperando a un trabajador.
- **running:** La tarea se está ejecutando en un trabajador (o en un ejecutor local/síncrono).
- **success:** La tarea terminó de ejecutarse sin errores.
- **failed:** La tarea tuvo un error durante la ejecución y no pudo ejecutarse.
- **skipped:** La tarea se omitió debido a ramificaciones, LatestOnly o similar.
- **upstream_failed:** Una tarea anterior falló y el Trigger Rule dice que la necesitábamos.
- **up_for_retry:** La tarea falló, pero le quedan intentos de reintento y se volverá a programar.
- **up_for_reschedule:** La tarea es un Sensor que está en modo schedule.



Trigger Rules

- **all_success:** (predeterminado) Todos los padres lo han logrado.
- **all_failed:** Todos los padres están en un estado failed o upstream_failed.
- **all_done:** Todos los padres han terminado con su ejecución.
- **one_failed:** Se activa tan pronto como al menos uno de los padres falla, no espera a que todos los padres hayan terminado.
- **one_success:** Se activa tan pronto como al menos uno de los padres tiene éxito, no espera a que todos los padres terminen.
- **none_failed:** Todos los padres no han fallado (failed o upstream_failed), es decir, todos los padres han tenido éxito o se han omitido.
- **none_failed_or_skipped:** todos los padres no han fallado (failed o upstream_failed) y al menos uno de los padres ha tenido éxito.
- **none_skipped:** Ningún padre está en un estado skipped, es decir, todos los padres están en un estado success, failed o upstream_failed.
- **dummy:** Las dependencias son sólo para mostrar, se activan a voluntad.

Providers:
Para la ampliación de funcionalidades de Airflow.
Brindan la incorporación de nuevas funcionalidades dentro de Airflow, ejemplo de providers: OpenAI, Google Cloud, AWS, Azure, dbt, snowflake

GCP

Tipos de operadores que podríamos utilizar
Operadores de Google Cloud Storage (GCS):

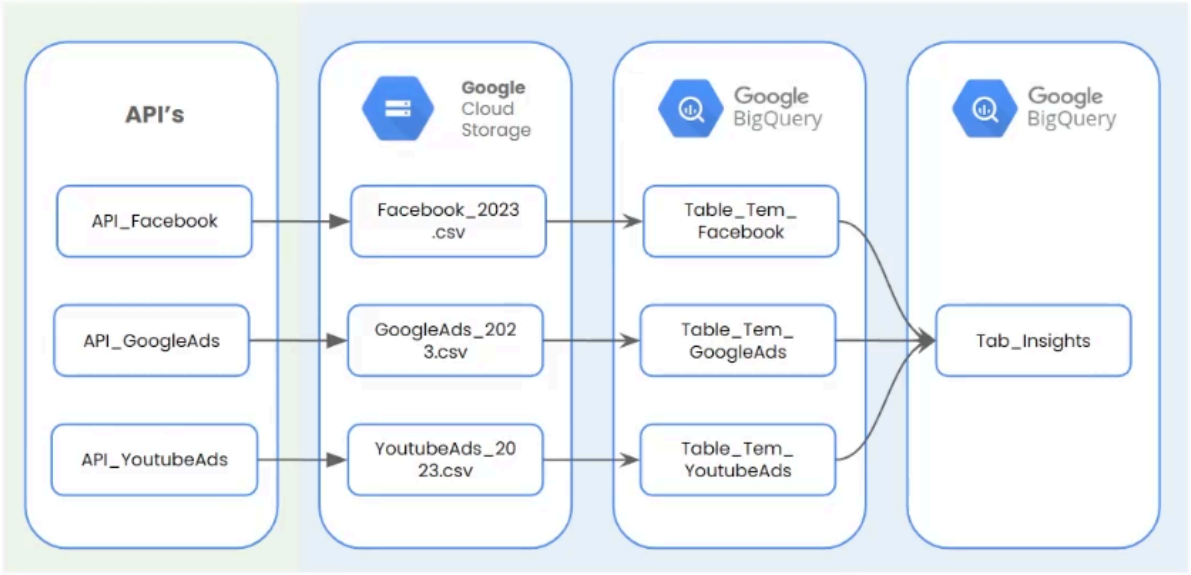
- **GCSToBigQueryOperator:** Transfiere datos desde Google Cloud Storage a BigQuery.
- **BigQueryToGCSOperator:** Transfiere datos desde BigQuery a Google Cloud Storage.

- **GCSTOGCSOperator:** Transfiere datos entre dos ubicaciones de Google Cloud Storage.
- **LocalFilesystemToGCSOperator:** Transfiere datos desde Local PC a Google Cloud Storage.

Estos operadores están diseñados para interactuar específicamente con servicios de Google Cloud Platform (GCP) y son útiles cuando trabajas con datos almacenados en GCS o BigQuery.

Caso: Automatización de Integración y Análisis de Datos de Campañas de Marketing

La agencia digital "Elevate Marketing" ha ejecutado con éxito ocho campañas de marketing a lo largo del último año en tres plataformas principales: Facebook, Google Ads y YouTube Ads. Cada campaña ha generado archivos de datos específicos, nombrados de acuerdo con la plataforma y el año, como Facebook_2023.csv, GoogleAds_2023.csv y YoutubeAds_2023.csv. La agencia busca consolidar y analizar estos datos para obtener insights valiosos cada mes.



Automatización Programada con Apache Airflow DAGS

El argumento `schedule_interval` al inicializar un DAG tiene, por defecto, un valor de `None`. En consecuencia, el DAG no se programará automáticamente y solo se ejecutará cuando se active manualmente desde la interfaz de usuario o la API. No obstante, al establecer la frecuencia de ejecución con la macro `@daily`, se generarán ejecuciones diarias a medianoche.
Otro aspecto importante a tener en cuenta es la fecha de inicio, ya que desempeña un papel crucial en la determinación de la primera ejecución y los intervalos subsiguientes.

Intervalos de programación Cron

La sintaxis cron consta de cinco componentes (minuto, hora, día del mes, mes y día de la semana) y permite una representación detallada del momento en que debe ejecutarse un trabajo cron. Los asteriscos (*) se utilizan para definir campos sin restricciones, lo que significa que no se tiene en cuenta el valor de ese campo.

Ejemplos de expresiones cron y sus significados incluyen:

- `0****`: Cada hora, ejecutando en la hora.
- `00***`: Todos los días, ejecutando a la medianoche.
- `00**0`: Semanal, ejecutando a la medianoche del domingo.

La sintaxis cron también permite definir expresiones más complejas, como:

- `001**`: La medianoche del primero de cada mes.
- `45 23 SAT`: A las 23:45 todos los sábados.

Además, las expresiones cron permiten definir colecciones de valores mediante comas (,) para listas de valores y guiones (-) para rangos de valores. Ejemplos incluyen:

- `00** MON, WED, FRI`: Correr todos los lunes, miércoles, y viernes a la medianoche.
- `00** MON-FRI`: Ejecutar todos los días de la semana a la medianoche.
- 0 0,12***: Correr todos los días a las 00:00 y 12:00.

Para generación de Crontab: www.crontab.guru

Intervalos de programación basados en frecuencia

```
with DAG(
   dag_id="time_delta",
   schedule_interval=dt.timedelta(days=3),
   start_date=dt.datetime(year=2019, month=1, day=1),
   end_date=dt.datetime(year=2019, month=1, day=5),
) as dag:
```

El DAG se ejecutará cada tres días después de la fecha de inicio, es decir, en los días 4, 7, 10 y así sucesivamente de enero de 2019. Este enfoque proporciona una solución sencilla y flexible para definir intervalos de tiempo basados en la frecuencia sin preocuparse por las limitaciones de las expresiones cron.

Adicionalmente, esta técnica también es aplicable para programar DAGs en intervalos más cortos o largos, como ejecutar el DAG cada 10 minutos (timedelta (minutes=10)) o cada dos horas (timedelta(hours=2)).

Uso de Plantillas en la Definición de Tareas con Operadores en Airflow

```
from airflow import DAG
from airflow.operators.bash_operator import BashOperator
from datetime import datetime

dag = DAG(
   'mi_dag_bash',
    schedule_interval='@daily',
    start_date=datetime(2023, 1, 1)
)
```

```
#Utilizando plantillas en el operador Bosh
tarea_bash BashOperator (
    task_id="tarea_bash",
    bash_command='echo {{ params.mensaje }}',
    params={'mensaje': '¡Hola, Airflow!"},
    dag=dag
)
```

Para tener en cuenta

Aplicación de kwargs

Mockaroo

```
schema "marketing_campaing"
set "Date", min: params['start_date'], max: params['end date']
generate 1000
```

API:

```
from datetime import datetime, timedelta
import requests
```

```
def _extract_data(platform, ti, **kwargs):
  # API Fecha: M/D/Y
  # Airflow: Y/M/D
  start_date = kwargs['data_interval_start'].strftime('%-m/%-d/%Y')
  end_date = kwargs['data_interval_end'].strftime('%-m/%-d/%Y')
  url =
f'https://my.api.mockaroo.com/marketing_campaing.json?start_date={start_date}&end_date={
end_date}'
  headers = {'X-API-Key': 'd67d54e0'}
  response = requests.get(url, headers=headers)
  tmp_file =
'marketing_stats_{platform}_{kwargs['ds_nodash']}_{kwargs['next_ds_nodash']}.csv'
  tmp_path = f'/tmp{tmp_file}'
  with
open(f'/tmp/marketing_stats_{platform}_{kwargs['ds_nodash']}_{kwargs['next_ds_nodash']}.
csv', 'wb') as file:
    file.write(response.content)
    file.close()
  ti.xcom_push(key='tmp_file_{platform}', value=tmp_file)
  ti.xcom_push(key='tmp_path_{platform}', value=tmp_path)
```

Operadores

```
facebook_extract = PythonOperator(
      task_id = 'facebook_extract',
      python_callable = _extract_Data,
      op_kwargs = {'platform', 'facebook_ADS''}
)
```

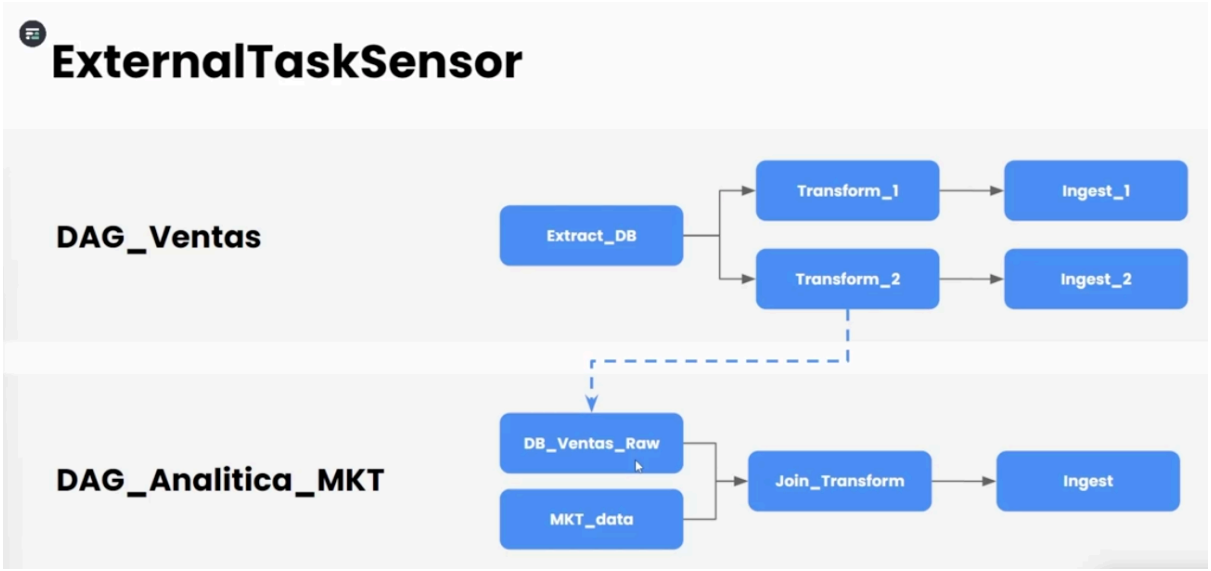
```
transf_facebook = LocalFilesystemToGCSOperator(
  task_id='transf_facebook',
  src='{{ ti.xcom_pull(key="tmp_path_facebook_ADS") }}',
  dst=f'marketing_data/{{ti.xcom_pull(key="tmp_path_facebook_ADS")}}',
  bucket='mkt-cf-elt',
  gcp_conn_id='google_cloud_conn
)
```

```
yads_bigquery = GCSToBigQueryOperator(
  task_id='yads_bigquery',
  bucket='mkt-cf-elt',
  source_objects=[f'marketing_data/{{ ti.xcom_pull(key="tmp_file_yads") }}'],
  destination_project_dataset_table='marketing.yads',
  schema_fields=schema_fields,
  write_disposition='WRITE_APPEND',
  skip_leading_rows=1,
  gcp_conn_id='google_cloud_conn'
)
```

Sensores en Airflow

los **sensores** son un tipo especial de operador que **esperan a que se cumpla una condición** antes de continuar con la ejecución del DAG. Es decir, **pausan** el flujo hasta que un evento específico suceda.

Ejemplo de aplicación



BranchPythonOperator

Es un operador en Apache Airflow que permite tomar decisiones en tiempo de ejecución dentro de un DAG (Grafo Acíclico Dirigido). Este operador se utiliza para crear ramificaciones en el flujo de tareas basándose en el resultado de una función Python. Dependiendo del resultado de esta función, el flujo del DAG puede dirigirse por diferentes caminos.

Ejemplo

```
tarea_si_existe = DummyOperator( task_id = "tarea_si_existe', dag = dag)
tarea_si_no_existe = DummyOperator( task id = tarea_si_no_existe",
dag=dag)

def decidir tarea(**kwargs):
 if kwargs['til state success:
   return "tarea si_existe
 else:
   return "tarea_si_no_existe

branching = BranchPythonOperator(
  task_id = 'branching',
  python_callable = decidir_tarea,
  provide_context=True,
  dag=dag
)

gcs_sensor >> branching
branching >> tarea_si_existe
branching >> tarea_si_no_existe
```