

# Python: Análisis de Datos y Ecosistemas Modernos de Analítica

---

**PIPELINES Y  
ORQUESTRAMIENTO**



# Recordando: pandas

---

**pandas** para manipular datos tabulares.

Creación de columnas derivadas.

Agrupaciones y filtros.

Exportación en CSV y Parquet.



# Recordando: Duckdb

---

**DuckDB** como alternativa SQL embebida:

Consultas directas sobre CSV/Parquet.

Integración con pandas.



# Recordando: pandas

---

Pandas nos permite representar una **serie de pasos encadenados**, un **pipeline de datos**:

```
import pandas as pd

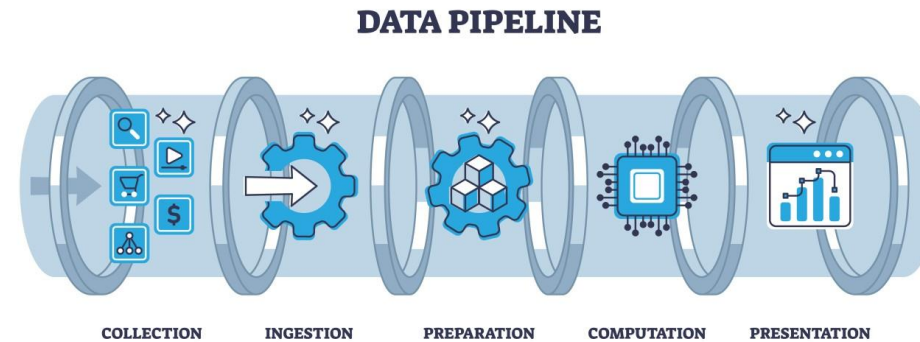
df = pd.read_csv("ventas.csv")
df["total"] = df["precio"] * df["cantidad"]
resumen = df.groupby("categoria", as_index=False)["total"].sum()
resumen.to_parquet("resumen.parquet", index=False)
```



# Pipeline de datos

---

Las funciones individuales pueden combinarse en operaciones continuas, actuando como **un flujo organizado de datos**.



Una **secuencia organizada de pasos** que procesan datos de manera reproducible.



# Pipeline de datos

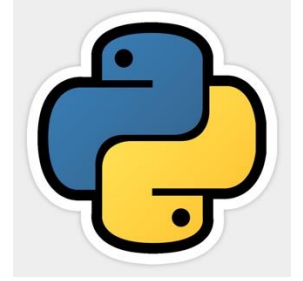
---

Pipelines → flujo estructurado, reutilizable y automatizable.

Ejemplo conceptual en pandas: **Ingesta → Transformación → Exportación.**

```
import pandas as pd

(
    pd.read_csv("ventas.csv")
    .assign(precio_total=lambda d: d["precio"] * d["cantidad"])
    .query("precio_total > 0")
    .groupby("categoria", as_index=False)
    .agg(total_ventas=("precio_total", "sum"))
    .to_parquet("resumen.parquet", index=False)
)
```



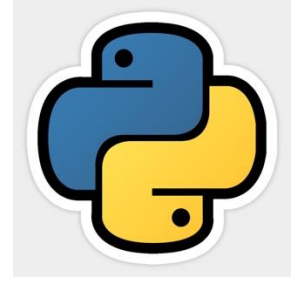
# Complejidad

---

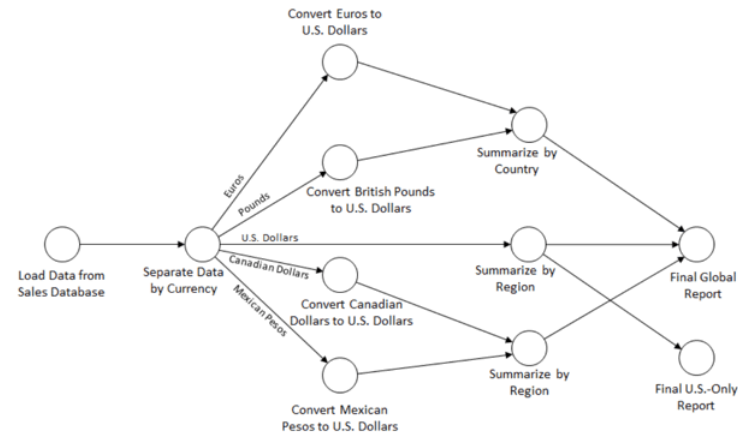
Cuando el análisis crece, incrementa la complejidad del proceso.

- **Múltiples fuentes** (varios CSV, bases de datos, APIs).
- **Normalización de columnas** y formatos distintos.
- **Consolidación** en un único dataset.
- **Validaciones adicionales** (duplicados, nulos, rangos).

# Complejidad



Eventualmente, las pipelines dejan de ser puramente lineales. Se convierten en **procesos con múltiples entradas, pasos y salidas**.



Organizar el flujo en **funciones claras** permite mantener el control a medida que crece la complejidad.



# Encuesta

La pipeline funciona correctamente.

Pero, ¿qué problemas genera esta forma de implementarla?

- A) Difícil de **validar** cada paso de manera aislada.
- B) Duplica lógica si se necesita en otro pipeline.
- C) Cambiar una fuente obliga a modificar **todo el script**.
- D) No se recomienda Pandas y Duckdb en conjunto.
- E) Rastrear errores es complejo, todo está en un mismo bloque.

```
...

# 1) Ingesta ventas (pandas)
df = pd.read_csv(sales_path)
df.columns = [c.strip().lower() for c in df.columns]

# Normalización mínima de tipos
for col in ("precio", "cantidad"):
    if col in df.columns:
        df[col] = pd.to_numeric(df[col], errors="coerce")

# 2) Ingesta catálogo (DuckDB con SQL, puede ser .parquet o .csv)
con = duckdb.connect()
cat = con.execute(f"SELECT * FROM '{catalog_path}'").df()
cat.columns = [c.strip().lower() for c in cat.columns]

# Asegurar columnas esperadas para el join/enriquecimiento
for col in ("producto", "categoria_norm", "impuesto"):
    if col not in cat.columns:
        cat[col] = pd.NA

# 3) Transformar ventas + enriquecer con catálogo
df["precio_total"] = df["precio"] * df["cantidad"]
df = df[df["precio_total"] > 0]
df = df.merge(cat[["producto", "categoria_norm", "impuesto"]], on="producto", how="left")

# 4) Ajustes con impuesto (si existe; por defecto 0)
df["impuesto"] = df["impuesto"].fillna(0)
df["precio_total_con_impuesto"] = df["precio_total"] * (1 + df["impuesto"])

# 5) Resumen por categoría
resumen = (
    df.groupby("categoria_norm", as_index=False)
    .agg(
        total_ventas=("precio_total_con_impuesto", "sum"),
        n_items=("cantidad", "sum"),
        precio_promedio=("precio", "mean")
    )
    .sort_values("total_ventas", ascending=False)
)

# 6) Exportar resultado
resumen.to_parquet(out_path, index=False)

...
```

# Encuesta

La pipeline funciona correctamente.

Pero, ¿qué problemas genera esta forma de implementarla?

- A) Difícil de validar cada paso de manera aislada.
- B) Duplica lógica si se necesita en otro pipeline.
- C) Cambiar una fuente obliga a modificar todo el script.
- D) No se recomienda Pandas y Duckdb en conjunto.
- E) Rastrear errores es complejo, todo está en un mismo bloque.

```
...  
  
# 1) Ingesta ventas (pandas)  
df = pd.read_csv(sales_path)  
df.columns = [c.strip().lower() for c in df.columns]  
  
# Normalización mínima de tipos  
for col in ("precio", "cantidad"):  
    if col in df.columns:  
        df[col] = pd.to_numeric(df[col], errors="coerce")  
  
# 2) Ingesta catálogo (DuckDB con SQL, puede ser .parquet o .csv)  
con = duckdb.connect()  
cat = con.execute(f"SELECT * FROM '{catalog_path}'").df()  
cat.columns = [c.strip().lower() for c in cat.columns]  
  
# Asegurar columnas esperadas para el join/enriquecimiento  
for col in ("producto", "categoria_norm", "impuesto"):  
    if col not in cat.columns:  
        cat[col] = pd.NA  
  
# 3) Transformar ventas + enriquecer con catálogo  
df["precio_total"] = df["precio"] * df["cantidad"]  
df = df[df["precio_total"] > 0]  
df = df.merge(cat[["producto", "categoria_norm", "impuesto"]], on="producto", how="left")  
  
# 4) Ajustes con impuesto (si existe; por defecto 0)  
df["impuesto"] = df["impuesto"].fillna(0)  
df["precio_total_con_impuesto"] = df["precio_total"] * (1 + df["impuesto"])  
  
# 5) Resumen por categoría  
resumen = (  
    df.groupby("categoria_norm", as_index=False)  
        .agg(  
            total_ventas=("precio_total_con_impuesto", "sum"),  
            n_items=("cantidad", "sum"),  
            precio_promedio=("precio", "mean")  
        )  
        .sort_values("total_ventas", ascending=False)  
)  
  
# 6) Exportar resultado  
resumen.to_parquet(out_path, index=False)  
  
...
```

# Complejidad

---

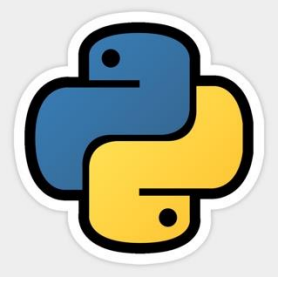


Organizar el flujo en **funciones claras** permite mantener el control a medida que crece la complejidad.

- Módulos y funciones separadas.
- Reutilización de lógica en distintos flujos.
- Testeo de unidad e integración.
- Cambiar una fuente o regla no exige tocar todo el pipeline.

# Complejidad

---



```
import pandas as pd

def limpiar(df: pd.DataFrame) -> pd.DataFrame:
    return (df
            .assign(
                precio_total=lambda d: d["precio"] * d["cantidad"],
                categoria_norm=lambda d: d["categoria"].str.strip().str.lower()
            )
            .query("precio_total > 0"))

def resumir_por_categoria(df: pd.DataFrame) -> pd.DataFrame:
    return (df
            .groupby("categoria_norm", as_index=False)
            .agg(total_ventas=("precio_total", "sum")))

(
    pd.read_csv("ventas.csv")
    .pipe(limpiar)
    .pipe(resumir_por_categoria)
    .to_parquet("resumen.parquet", index=False)
)
```



# De Pipelines a ETL

---

Lo que hicimos con **pandas + DuckDB** ya es un mini **pipeline de ETL**:

- **Extract (E)** → leer archivos CSV, Parquet, bases de datos.
- **Transform (T)** → limpiar, crear columnas, unir catálogos, agrupar.
- **Load (L)** → exportar a Parquet, CSV, bases de datos o dashboards.

**ETL = sistematizar** el proceso de llevar datos crudos a datos listos para análisis o consumo.



# Automatizar ETL

---

Ejecutar un proyecto ETL “a mano” puede funcionar en pruebas, pero en la práctica los datos cambian **cada día, cada hora, incluso cada minuto**.

Dependiendo de lal Proyecto, se necesita:

- **Ejecuciones recurrentes** (ej. cada mañana 8am).
- **Monitoreo de fallas**.
- **Escalabilidad** cuando aumentan las fuentes o las reglas.



# CRON: Linux / macOS

---

**Servicio del sistema** que ejecuta tareas en horarios definidos.

Cada usuario puede definir un archivo, mediante un lenguaje de horarios: **crontab**.

**Ejemplo:** correr un script cada día a las 7:30

```
30 7 * * * /usr/bin/python3 /home/usuario/proyecto/run_pipeline.py
```



# CRON: Linux / macOS

---

## Limitaciones de cron

- No sabe de **dependencias** entre tareas (todas corren aisladas).
- **Sin reintentos automáticos**: si algo falla, no se vuelve a ejecutar hasta el siguiente horario.
- **Monitoreo limitado**: solo logs locales por defecto.

Difícil de escalar cuando hay muchos scripts o flujos interdependientes.





# Task Scheduler (Windows)

---

Equivalente a **cron**, pero en Windows.

- Herramienta integrada para programar tareas recurrentes.
- Interfaz gráfica → elegir programa, frecuencia y condiciones.
- También se puede usar por línea de comandos (schtasks).

Limitaciones similares a las de **cron**.



# Automatización local

---

Aceptable para:

- Scripts utilitarios (enviar correos, backups simples).
- Pipelines locales simples (ejecutar un ETL una vez al día).
- Pipelines locales complejas pero no optimizadas.

Útiles cuando **no se necesitan garantías adicionales** como:

- Reintentos automáticos.
- Manejo de dependencias entre múltiples pasos.
- Escalado a varios nodos o entornos distribuidos.

# Encuesta

---

Un pipeline tiene 3 pasos. Cada paso se agenda **POR SEPARADO** con cron.

1. Descargar ventas de una API cada mañana.
2. Limpiar y validar los datos.
3. Generar un reporte y enviarlo por correo.

¿Cuál es el mayor riesgo en este flujo?

# Encuesta

---

Un pipeline tiene 3 pasos. Cada paso se agenda **POR SEPARADO** con cron.

1. Descargar ventas de una API cada mañana.
2. Limpiar y validar los datos.
3. Generar un reporte y enviarlo por correo.

¿Cuál es el mayor riesgo en este flujo?

Que los pasos se ejecuten de forma inconsistente.

# Un gran problema

---

Cuando se usan sistemas de automatización sin gestión de dependencias, cada paso se ejecuta de forma aislada, lo que genera el riesgo de producir datos incompletos o inválidos.

De automatización a orquestación

**Orquestadores** existen para gestionar pipelines **complejas, críticas y de forma escalable**.

# ¿Qué es un orquestador?

---

Herramienta que **coordina la ejecución de pipelines** de datos.

- Manejo de **dependencias** entre pasos.
- **Reintentos automáticos**.
- **Monitoreo centralizado** y alertas.
- Escalabilidad: ejecución en clústeres o nube.

# La orquestación es crítica

---

La orquestación es el paso natural después de la automatización básica.

- **ETL en masa:** pipelines que integran múltiples fuentes.
- **Streaming + batch:** cuando conviven flujos de datos en tiempo real y procesos programados.
- **Machine Learning:** entrenar modelos con datos frescos, validar.
- **BI / dashboards:** asegurar que los reportes se alimenten con datos actualizados y correctos.

# Orquestadores

---

Existen orquestadores independientes de un lenguaje

Ejemplo: **Argo Workflows**, corre sobre Kubernetes.

Define pipelines como **manifiestos YAML**. No está atado a un lenguaje: se orquestan contenedores con cualquier stack.

**Desventaja:**

Complejidad operacional





# Orquestadores

---

Existen orquestadores ligados a un lenguaje, como Python.

Ejemplo: **Dagster**.

Los pipelines se definen como código Python.

Ofrece integración natural con librerías de datos (pandas, duckdb, soda, etc).

**Desventaja:**

Menos flexible si el stack es heterogéneo (Java, R, Go, etc.).



# Dagster

---



Orquestador de datos centrado en **Python**.

- Modela el pipeline en términos de “assets” (datasets/artefactos) y sus dependencias.
- Incluye checks (validaciones) y UI para visualizar ejecuciones, estados y dependencias.

<https://docs.dagster.io/>

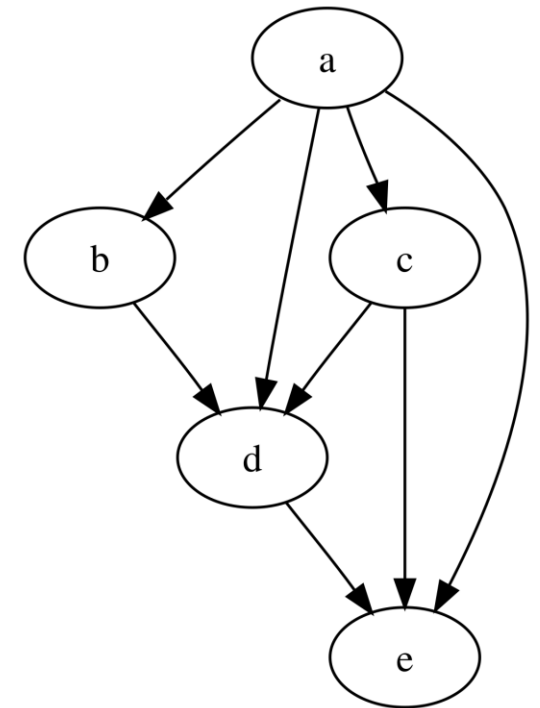
# DAG



Concepto clave en los orquestadores:

Directed Acyclic Graph (*Grafo Dirigido Acíclico*).

- **Directed:** cada flecha indica el **flujo de datos** o el **orden de ejecución**
- **Acyclic:** no hay ciclos, un paso no puede depender de sí mismo
- **Graph:** nodos (datasets, cálculos) y aristas (dependencias).



# DAG



---

En Dagster una pipeline se modela como un DAG

Nodos → pasos del pipeline: *input datasets, transformaciones, dataset final*.

Aristas → dependencias entre pasos.

Flujo siempre **avanza hacia adelante**, nunca vuelve atrás.

# DAG



- 
- **Claridad:** describe explícitamente qué depende de qué.
  - **Confiabilidad:** asegura que un paso no corra antes de sus dependencias.
  - **Paralelismo:** pasos independientes pueden ejecutarse al mismo tiempo.
  - **Monitoreo:** fácil de visualizar el estado de cada nodo (éxito, fallo, pendiente).

# Encuesta

---



¿Qué es un DAG?

# Encuesta

---

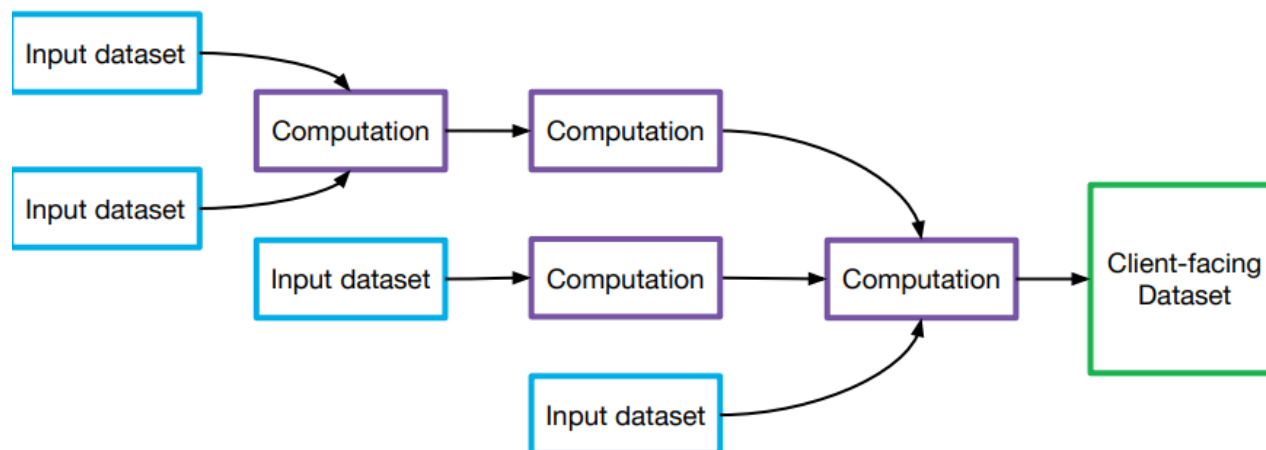


¿Qué es un DAG?

Un **grafo** que representa dependencias entre pasos de una pipeline.

# Modelo de ejecución

---

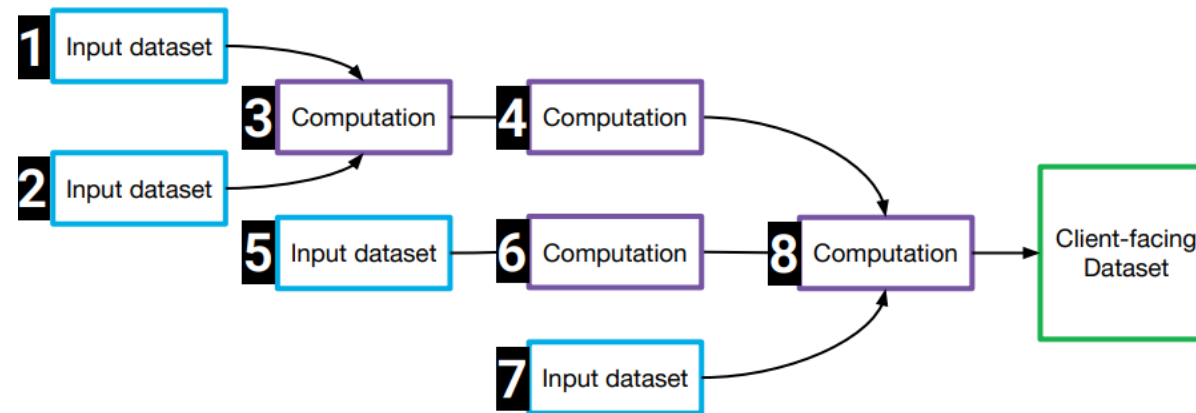




# Sin Orquestación



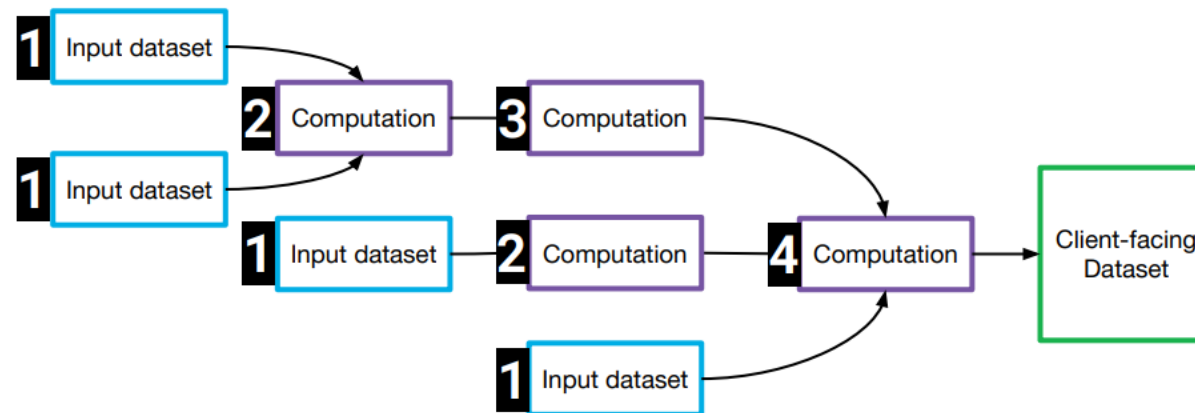
La única forma de garantizar el resultado correcto:



# Con Orquestación



Se optimiza la ejecución y puntos de paralelización



# Dagster



---

Arquitectura modular que ofrece:

**Desarrollo local** → definir y experimentar pipelines como código.

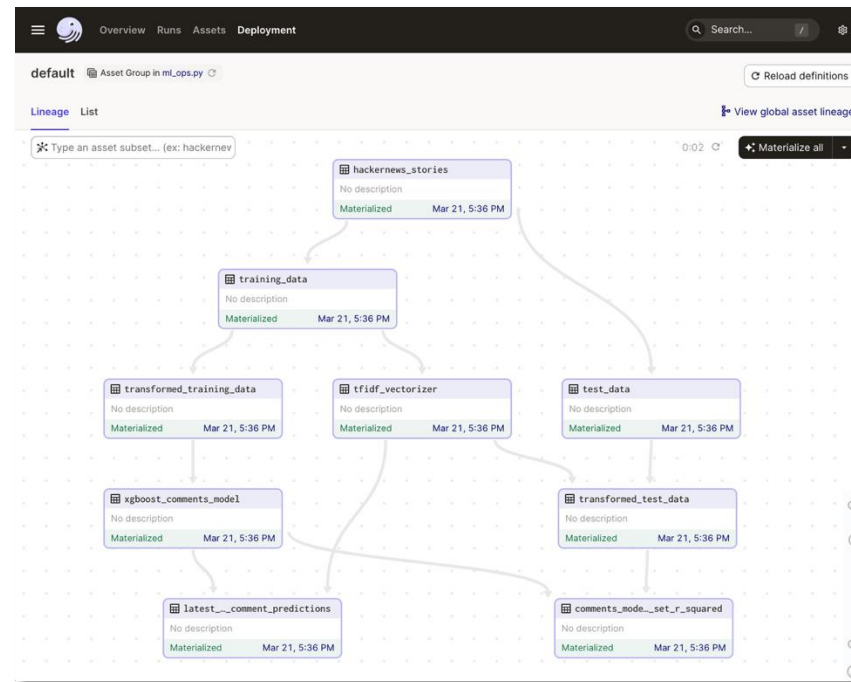
**Ejecución en producción** → horarios automatizados con monitoreo.

Todo se organiza en torno a **assets**, que representan los datasets y productos intermedios.

# Dagster: UI



Dagster ofrece un **entorno de gestión y monitoreo desde el primer momento** a través de su UI.



# Dagster: Asset

---



Un *asset* es un dataset o artefacto producido por un paso del pipeline.

**Son declarativos:** describen qué producen y de qué dependen.

Ejemplo: un archivo Parquet, una tabla en DuckDB, un DataFrame en memoria.

# Asset: Usando funciones



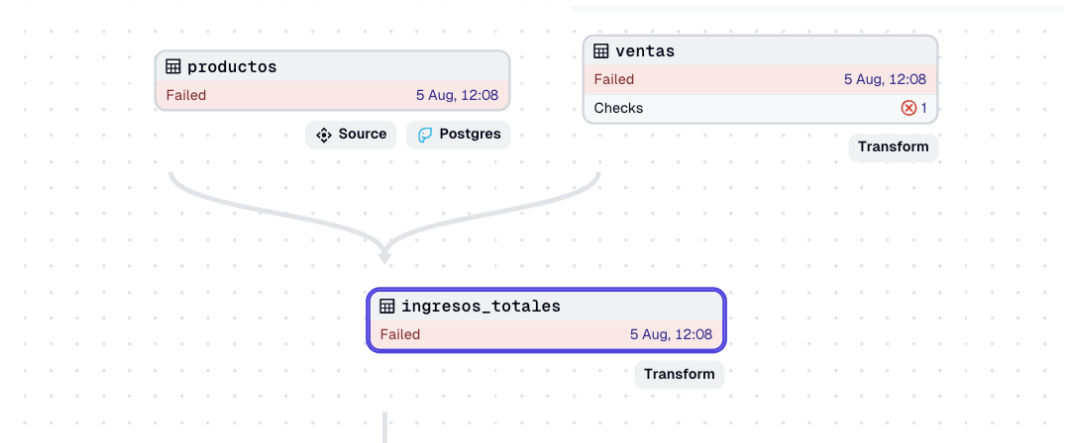
Dagster permite declarar qué función produce un asset. Mediante **introspección automática**, Dagster detecta las **relaciones de dependencia**.

```
from dagster import asset
import pandas as pd

@asset
def productos() -> pd.DataFrame:
    ...

@asset
def ventas() -> pd.DataFrame:
    ...

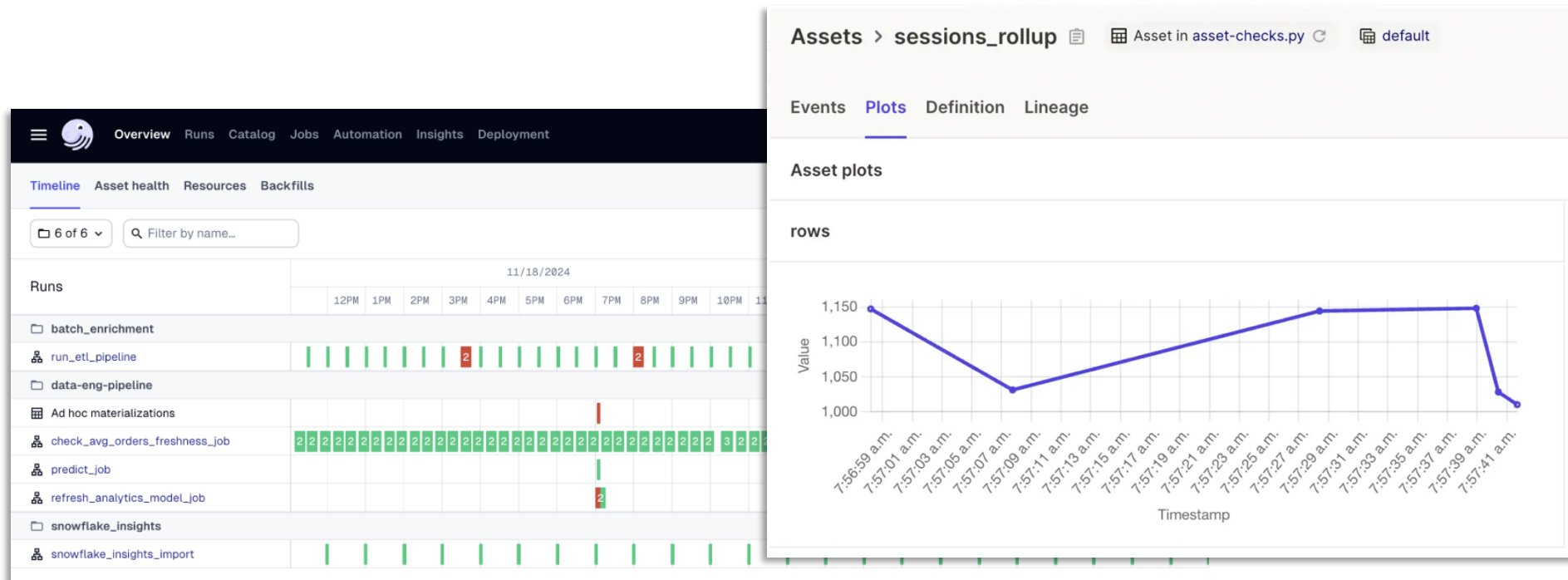
@asset
def ingresos_totales(productos, ventas) -> pd.DataFrame:
    # Une productos con ventas y calcula ingresos
    ...
```



# Explorando metadatos



Como parte de la orquestación, **Dagster** almacena información descriptiva de cada ejecución.



# Dagster: Job

---



Un **Job** es la unidad principal de ejecución.

Permite **materializar** (ejecutar) un subconjunto del DAG de assets.

Dagster:

- **Resuelve automáticamente el orden de ejecución** a partir de las dependencias entre assets.
- **Gestiona la computación:** paraleliza pasos independientes (cuanto sea possible)



# Dagster: Schedule

---



Un **Schedule** automatiza un job en intervalos definidos.

```
from dagster import define_asset_job, ScheduleDefinition

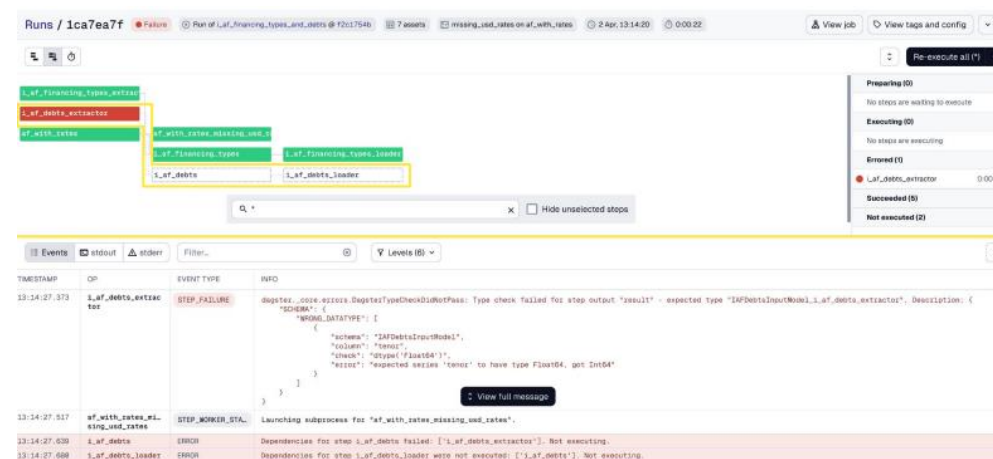
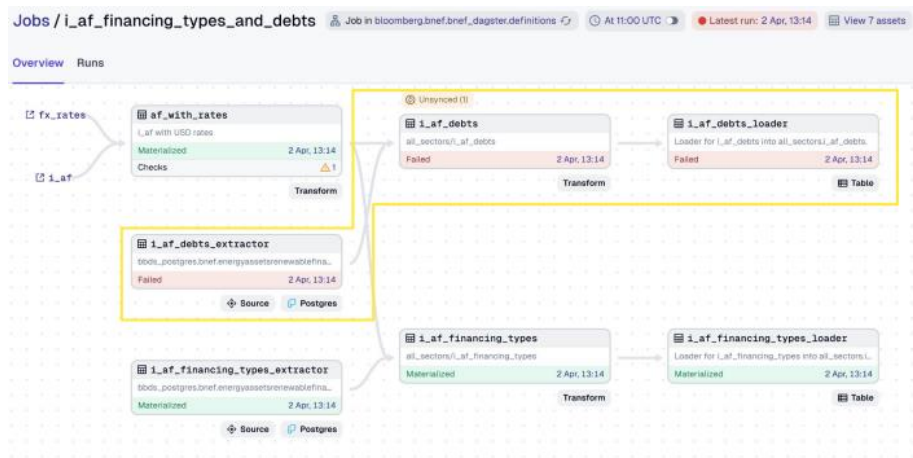
# Definimos un Job que materializa ingresos_totales
ingresos_totales_job = define_asset_job(
    "ingresos_totales_diario",
    # Incluye ingresos_totales y todas sus dependencias (ventas, productos)
    selection=["*ingresos_totales"]
)

# Definimos un Schedule: ejecutar todos los días a las 07:30 AM
daily_schedule = ScheduleDefinition(
    job=ingresos_totales_job,
    cron_schedule="30 7 * * *"
)
```

# Investigando



La UI permite identificar de forma **rápida y efectiva** dónde falló la ejecución.



# Chequeos

---



Un **Asset Check** es una tarea asociada a un asset que ejecuta **validaciones** definidas por el usuario.

Los Asset Checks están **integrados con la orquestación** y pueden **bloquear la ejecución de assets dependientes** si las validaciones fallan.

# Chequeos



Las validaciones se definen totalmente en código, aprovechando toda la potencia de Python y sus librerías.

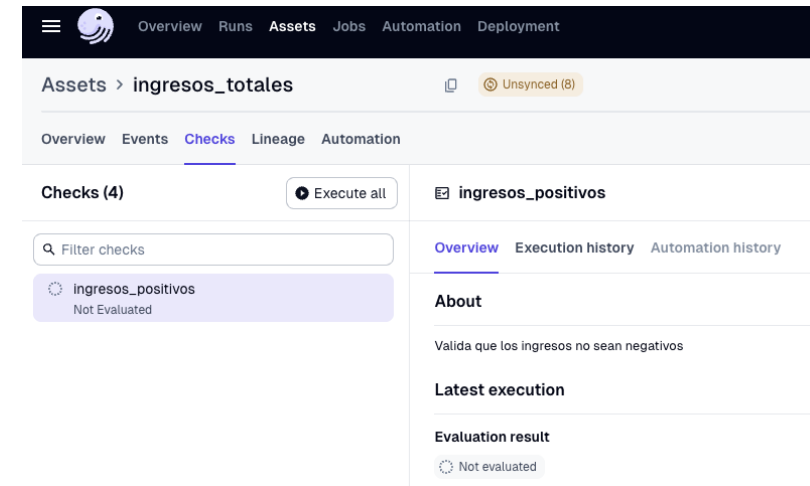
```
import pandas as pd
from dagster import asset, asset_check, AssetCheckResult

@asset
def productos():
    ...

@asset
def ventas():
    ...

@asset
def ingresos_totales(productos, ventas):
    ...

# -----
# Asset Check: validar que no existan ingresos negativos
# -----
@asset_check(asset="ingresos_totales")
def check_ingresos_positivos(_, ingresos_totales: pd.DataFrame):
    negativos = (ingresos_totales["ingreso"] < 0).sum()
    return AssetCheckResult(
        passed=(negativos == 0),
        metadata={"registros_negativos": negativos}
    )
```



The screenshot displays the Dagster web interface. The top navigation bar includes links for Overview, Runs, Assets, Jobs, Automation, and Deployment. The main content area is titled 'Assets > ingresos\_totales' and shows a list of checks under the 'Checks (4)' tab. A search bar is available to filter checks. The 'ingresos\_positivos' check is highlighted, showing a status of 'Not Evaluated'. The right sidebar provides details for the selected check, including an 'About' section stating 'Valida que los ingresos no sean negativos' and a 'Latest execution' section showing the 'Evaluation result' as 'Not evaluated'.

# Recursos Externos

---



Un **Resource** es un componente **configurable y reutilizable**.

Sirve para interactuar con **dependencias externas**:

- Bases de datos.

- APIs externas.

- Sistemas de archivos o almacenamiento en la nube.

# Recursos Externos

---



Los **assets** pueden depender de resources para realizar sus tareas.

Los resources se pueden **configurar por trabajo (Job)**, adaptándose a distintos entornos.

Beneficio:

Separar la lógica de datos (assets) de la infraestructura (resources).

# Recursos Externos

---



```
from dagster import asset, resource, Definitions

@resource
def conexion_db():
    # Aquí podrías abrir una conexión a PostgreSQL, DuckDB, etc.
    import duckdb
    return duckdb.connect()

@asset
def leer_clientes(conexion_db):
    return conexion_db.execute("SELECT * FROM clientes").df()

defs = Definitions(assets=[leer_clientes], resources={"conexion_db": conexion_db})
```

# Datos entre Assets

---



Un **IO Manager** define cómo se almacenan y recuperan los datos entre ejecuciones de assets.

Separa la **lógica de procesamiento de datos** (assets) del **código de lectura y escritura** (persistencia).  
Permite que el mismo pipeline use diferentes backends y facilite la distribución de la computación.

Por defecto, Dagster usa el **disco local**.



# Datos entre Assets

---



En ejecución **local** o en un **servidor único**, el disco duro es suficiente.

Pero en un context distribuido, ¿como acceden otros nodos al resultado de un asset?

Un **IO Manager** que permita gestionar un almacenamiento accesible a todos los nodos.

- **S3 / GCS / Azure Blob** (almacenamiento en la nube).
- **Bases de datos** (ej. Postgres, Snowflake, BigQuery).
- **Sistemas de archivos distribuidos** (ej. NFS, HDFS).

# Procesos Avanzados

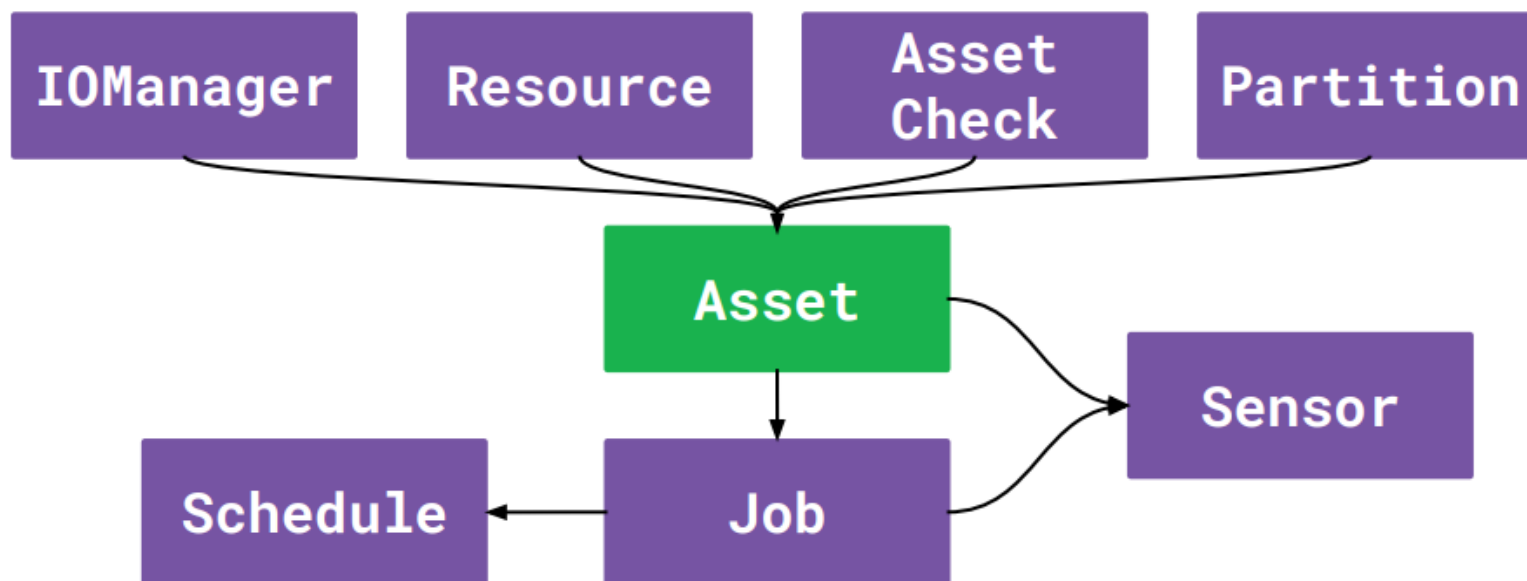
---



Dagster permite construir pipelines **más dinámicos, reactivos y escalables**.

- **Partitioning:** dividir un asset en partes (tiempo, categorías, ejes) para procesar datasets grandes en lotes pequeños.
- **Sensors:** reaccionar a eventos (archivo nuevo, corrida completada) para disparar jobs o materializar assets.
- **Ejecución distribuida:** usar colas de trabajo (ej. Celery) para escalar la ejecución en múltiples workers.

# Dagster



<https://docs.dagster.io/getting-started/concepts>

# El Beneficio

---



Dagster es excelente para **equipos Python** que buscan claridad y validación integrada, pero puede presentar desafíos en **adopción, despliegue y ecosistemas heterogéneos**.

# Limitaciones

---



El modelo basado en *assets* y *checks* es poderoso, pero puede resultar complejo al inicio.

- Requiere entender bien la arquitectura (assets, ops, jobs, schedules, IO managers).
- El uso en local es sencillo con `dagster dev`, pero en producción, configurar workers distribuidos (Celery, Kubernetes) puede ser complejo. Requiere experiencia en infraestructura.
- Menos cantidad de ejemplos y tutoriales disponibles.

# Instalando Dagster

---



Componentes clave (instalados vía pip/uv):

- dagster-web-server**: visualización, ejecución manual, logs.
- dagster** : gestiona *schedules* y *sensors*, *el corazón del orquestrador*
- create-dagster**: *opcional, crea el esqueleto para un Proyecto.*

**Base de datos de metadatos:**

Si no configuras una, **usa SQLite local por defecto.**

Suficiente para la clase y desarrollo local.

<https://docs.dagster.io/etl-pipeline-tutorial>

# Operando Dagster

---



La interfaz del servidor web permite directamente ejecutar o manejar operaciones. Sin embargo, la CLI permite ejecutar todos los procesos de manera programática.

<https://docs.dagster.io/api/clis/cli>

# Encuesta

---



Tienes un pipeline en Dagster con cinco elementos:

- productos (lee un catálogo desde CSV)
- ventas (lee datos de ventas desde otro CSV)
- ingresos\_totales (combina ambos y calcula ingresos)
- check\_ingresos (valida que no haya ingresos negativos)
- publicar\_reporte (publica los datos de ingresos\_totales)

El job está programado con un **horario diario a las 07:30 AM**.

¿Qué sucederá si mañana a las 07:30 AM ocurre un error generando **ventas**?



# Encuesta

---



Tienes un pipeline en Dagster con cinco elementos:

- productos (lee un catálogo desde CSV)
- ventas (lee datos de ventas desde otro CSV)
- ingresos\_totales (combina ambos y calcula ingresos)
- check\_ingresos (valida que no haya ingresos negativos)
- publicar\_reporte (publica los datos de ingresos\_totales)

El job está programado con un **horario diario a las 07:30 AM**.

¿Qué sucederá si mañana a las 07:30 AM ocurre un error generando **ventas**?

**Dagster reporta el error, y bloquea el flujo a los siguientes pasos.**

# Tutorial

---

- Repositorio aislado en Codespaces
- Definición de dependencias y requisitos
- Instalación de Dagster
- Recorrido por la interfaz
- Creación de un asset de ejemplo con pandas
- Configuración de un schedule de ejemplo
- Definición de un chequeo de ejemplo