Tu pipeline de datos funciona. ¿Pero está bien diseñado?

Flujo completo de datos: desde sistemas transaccionales hasta insights accionables. Por qué cada decisión arquitectónica importa.

CONTEXTO

El problema: dashboards que funcionan, métricas que mienten

La trampa silenciosa

Un pipeline puede estar "en verde", con flujo constante de datos, y estar mal diseñado. El problema aparece cuando las métricas son inconsistentes entre equipos y nadie lo nota. Un dashboard puede mostrar números técnicamente correctos pero conceptualmente equivocados. Y ahí tomás decisiones sobre datos que no sirven.

Una arquitectura de datos mal diseñada no se nota el día 1. Se paga después, y caro.

Caso real: empresa SaaS con PostgreSQL transaccional. Churn inconsistente, evolución de planes confusa, ingresos que no cierran. Los equipos pedían métricas, pero nadie confiaba en los números. Si tus dashboards funcionan pero las métricas no coinciden entre áreas, estás tomando decisiones sobre mentiras.

CONTEXTO

La solución: arquitectura medallion en Databricks

Stack tecnológico:

- **Origen:** PostgreSQL (sistema transaccional)
- **Plataforma de datos:** Databricks (Unity Catalog, Delta Lake, Jobs)
- **Consumo:** Databricks Apps

Arquitectura: Medallion (Bronze → Silver → Gold) con orquestación automatizada.

Este documento recorre la implementación completa: desde la extracción de PostgreSQL hasta el consumo en aplicaciones, con cada decisión arquitectónica explicada.

ROADMAP

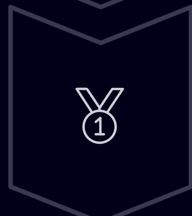
El recorrido completo: de transaccional a analítico

PostgreSQL

Sistema transaccional

Extracción

Decisiones de ingesta

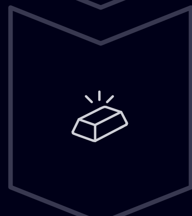


Bronze

Preservación sin transformar

Silver

Limpieza y consistencia

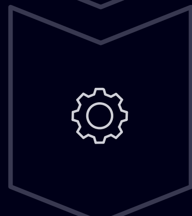


Gold

Modelo analítico

Consumo

Apps y visualización



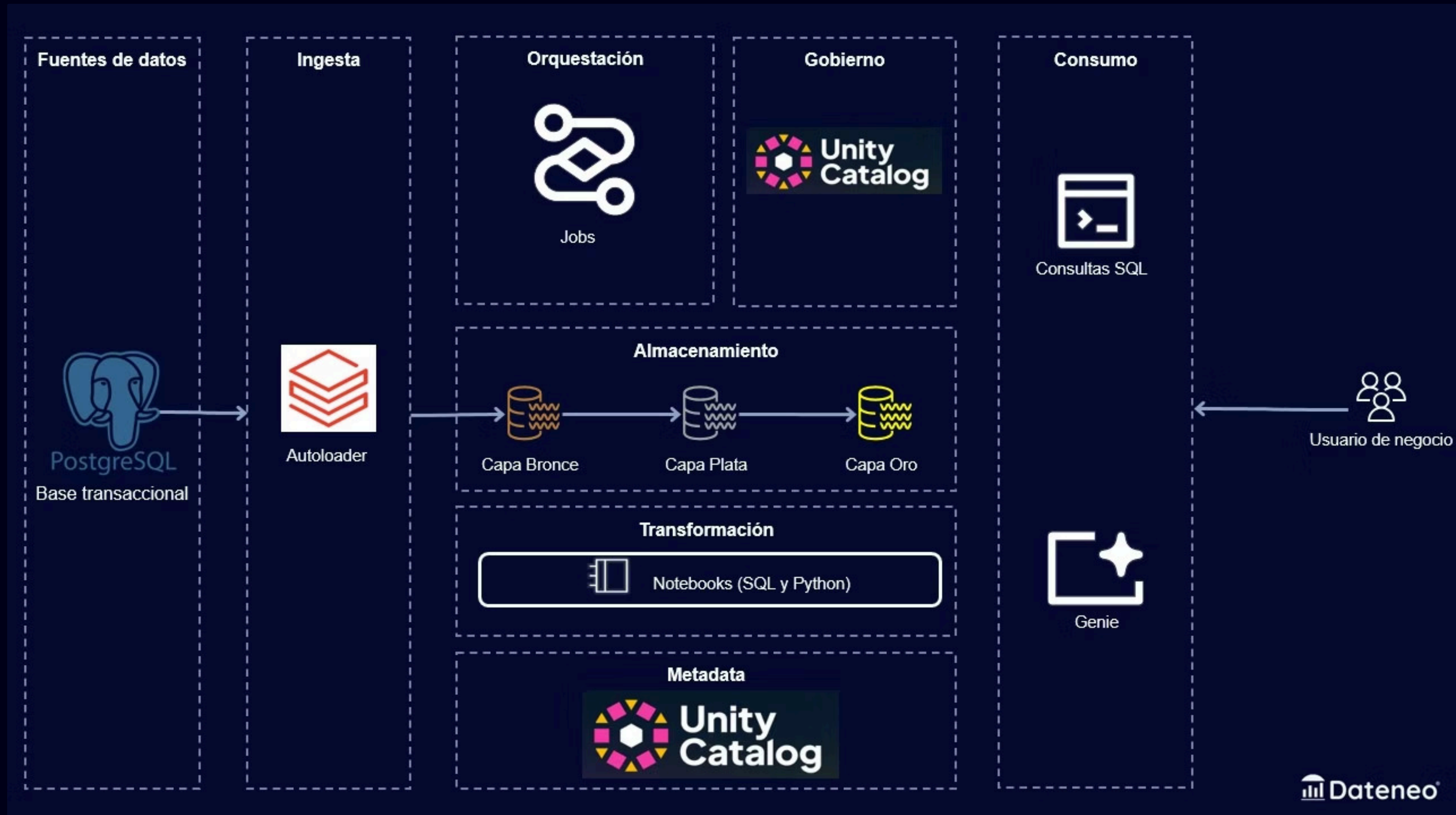
Orquestación

Automatización confiable

Cada capa tiene un propósito. Saltarse una es una decisión arquitectónica, no un atajo.

ROADMAP

El caso: arquitectura implementada



INGESTA

Punto de partida: decisiones desde la extracción

Las decisiones que tomes acá sobre cómo extraer y cargar datos impactan todo lo que viene después. Una estrategia de ingesta mal pensada te genera cuellos de botella, datos inconsistentes o costos que explotan más adelante.

Full refresh vía JDBC

Para tablas core con volumen moderado. Conexión directa, extracción completa, simplicidad operativa.

CSV + Auto Loader

Para tabla específica con alto volumen. Procesamiento incremental, detección automática de esquema.

Error común: Usar la misma estrategia de ingesta para todas las tablas.

No todo se mueve igual. Las decisiones de arquitectura empiezan desde la ingesta: diferente volumen, diferente frecuencia, diferente estrategia.

```
# Extracción completa vía JDBC
df = spark.read.jdbc(url=jdbc_url, table=table, properties=properties)
# Sobrescritura con partición por fecha
(df_strings.write
    .mode("overwrite")
    .option("replaceWhere", f"fecha_extraccion = '{today}'")
    .saveAsTable(target))
```

```
# Procesamiento incremental con Auto Loader
query = (spark.readStream
  .format("cloudFiles")
  .option("cloudFiles.format", "csv")
  .option("cloudFiles.schemaLocation", checkpoint_path)
  .load(f"{VOLUMEN_LANDING}/{TABLA}/")
  .withColumn("fecha_carga", current_date())
  .writeStream
  .option("checkpointLocation", checkpoint_path)
  .trigger(availableNow=True)
  .toTable(f"{CATALOGO}.datavision.{TABLA}"))
```

Diferencias clave:

- **JDBC Full Refresh:** Conexión directa a PostgreSQL, extracción batch completa, sobrescritura controlada por partición. Ideal para tablas con volumen moderado.
- **Auto Loader Incremental:** Streaming incremental desde archivos CSV, detección automática de esquema, checkpoint para procesamiento exactamente-una-vez. Adecuado para tablas de alto volumen y actualizaciones continuas.

Decisión clave: Volumen y frecuencia determinan la estrategia, no la herramienta disponible.

BRONCE

Bronce: preservar sin interpretar

Por qué existe Bronze

Reprocesabilidad. Si algo falla en silver u oro, puedes volver a bronze sin reextraer del origen.

Bronze es tu seguro de vida

La filosofía de bronce

En bronze no modelo, no agrego lógica de negocio, no optimizo. Solo persisto exactamente lo que llegó desde PostgreSQL.

Si paro acá no tengo limpieza ni modelo analítico, pero tengo algo crítico: reprocesabilidad completa.

```
CREATE OR REPLACE TABLE
{CATALOGO}.datavision.account_premium_features (
    account_feature_id STRING,
    account_id STRING COMMENT 'Referencia a la cuenta (FK
a cuentas)',
    feature_id STRING COMMENT 'Referencia a la
funcionalidad premium (FK a funcionalidades_premium)',
    purchase_date STRING COMMENT 'Fecha en que se
adquirió la funcionalidad',
    amount_paid COMMENT 'Monto pagado por la
funcionalidad (puede diferir del precio base)',
    fecha_carga COMMENT 'Fecha de carga en la capa bronce
desde la base origen',
    _rescued_data STRING COMMENT 'garantiza que las
columnas que no coinciden con el esquema se rescaten en
lugar de eliminarse'
)
```

Error común: Agregar lógica de negocio en bronze 'porque ya estoy acá'. Esto rompe la reprocesabilidad.

Datos append-only

Sin transformaciones

Copia fiel del origen

Preservación total

Qué pasa si no existe Bronze:
- No podés auditar: ¿Qué dato llegó exactamente del origen?
- No podés reconstruir estados: Cada error requiere volver al OLTP
- Dependés del sistema transaccional para cada fallo: Eso no escala y genera presión operativa
- Perdés la capacidad de reprocesar: Si cambia la lógica de negocio, tenés que reextraer todo

TRANSICIÓN

De bronce a plata: del caos a la consistencia

Problema en bronze

Duplicados potenciales, formatos inconsistentes, estructura OLTP. No puedo medir churn así. Ejemplo real: ¿cómo mido churn si tengo duplicados y fechas en formato string?

Solución en silver

Necesito consistencia antes de modelar. Datos limpios y confiables. Tipos correctos, deduplicación, validaciones. Ahora los datos son confiables.

```
CREATE OR REPLACE TABLE {catalogo}.core_negocio.cuentas_funcionalidades_premium ( id_cuenta_funcionalidad BIGINT COMMENT 'Identificador único de la relación cuenta-funcionalidad (PK)', id_cuenta BIGINT COMMENT 'Referencia a la cuenta (FK a cuentas)', id_funcionalidad BIGINT COMMENT 'Referencia a la funcionalidad premium (FK a funcionalidades_premium)', fecha_compra DATE COMMENT 'Fecha en que se adquirió la funcionalidad', monto_pagado DECIMAL(10,2) COMMENT 'Monto pagado por la funcionalidad (puede diferir del precio base)', fecha_carga_bronce DATE COMMENT 'Fecha de carga en la capa bronce desde la base origen', fecha_carga_plata DATE COMMENT 'Fecha de carga en la capa plata desde bronce', CONSTRAINT cuenta_funcionalidades_premium_pk PRIMARY KEY(id_cuenta_funcionalidad), CONSTRAINT fk_cuentas_cuenta_funcionalidades_premium FOREIGN KEY(id_cuenta) REFERENCES {catalogo}.core_negocio.cuentas(id_cuenta), CONSTRAINT fk_funcionalidades FOREIGN KEY(id_funcionalidad) REFERENCES {catalogo}.core_negocio.funcionalidades_premium(id_funcionalidad) ) USING DELTA CLUSTER BY (id_cuenta_funcionalidad) COMMENT 'Registro de funcionalidades premium que los usuarios han adquirido individualmente.'
```

Silver no responde preguntas de negocio todavía. Pero hace posible que oro lo haga.

PLATA

Plata: consistencia y control

Por qué existe Silver

Silver garantiza consistencia antes de modelar. Sin ella, pasa esto:

- Cada analista limpia los datos a su manera: 5 personas, 5 versiones de la verdad
- Los dashboards se rompen cuando cambia el origen: No hay capa de protección
- No hay validaciones centralizadas: Los errores se propagan hasta oro y consumo
- El debugging es imposible: ¿El problema está en la extracción, la transformación o la lógica de negocio?

Validación

Tipos de datos, formatos consistentes

Normalización

Fechas, nulos, deduplicación

Calidad

Reglas de validación aplicadas

Decisión arquitectural clave

Separamos cada entidad en **tabla vigente + histórica** con snapshots diarios. **No usamos SCD Type 2**.

Los snapshots simplifican la reconstrucción temporal y reducen la complejidad en joins históricos. Acceder a datos en cualquier punto del tiempo es más simple.

⚠ **Error común:** Mezclar limpieza de datos con lógica de negocio. Silver limpia y valida. Gold modela y responde preguntas de negocio.

Silver todavía no responde preguntas de negocio, pero los datos ya son confiables.

```
-- Validación de valores nulos en columnas críticas WITH datos_origen_validacion AS ( SELECT CAST(amount_paid AS DECIMAL(10,2)) AS monto_pagado FROM datavision.account_premium_features ) validacion_nulos AS ( SELECT COUNT(*) AS total_registros, SUM(CASE WHEN monto_pagado IS NULL THEN 1 ELSE 0 END) AS nulos_monto_pagado FROM datos_origen_validacion ), resumen AS ( SELECT total_registros, nulos_monto_pagado, CASE WHEN nulos_monto_pagado > 0 THEN 'ERROR: Valores nulos en monto_pagado' ELSE 'OK' END AS estado_validacion FROM validacion_nulos ) SELECT total_registros, estado_validacion, CASE WHEN estado_validacion != 'OK' THEN RAISE_ERROR(estado_validacion) ELSE 'Validación de nulos exitosa' END AS resultado FROM resumen
```

TRANSICIÓN

De plata a oro: del dato al insight

Problema

Si negocio consulta silver directamente, cada dashboard calcula churn distinto porque cada uno implementa su propia lógica. Se duplica lógica, se pierde gobierno.

Solución

Oro centraliza las reglas de negocio. Una métrica, una definición, una fuente de verdad.

Oro no es opcional si querés gobierno de datos.

ORO

Oro: modelo orientado a preguntas

Por qué existe Gold

Para responder preguntas de negocio con reglas centralizadas. Silver tiene datos limpios, oro tiene significado.

Acá el diseño cambia completamente. No replico PostgreSQL ni plata.

Diseño para responder preguntas de negocio.

⚠ **Error común:** Replicar la estructura de silver en oro. Oro no es staging, es modelo analítico.

✓ **Buena práctica:** Diseñar oro desde las preguntas del negocio, no desde las tablas de silver.

Dimensiones

Cuentas, planes, tiempo. Contexto para el análisis.

Hechos

Suscripciones, eventos, transacciones con métricas.

Métricas de churn

Detección de upgrades, downgrades, cancelaciones.

Window functions

Comportamiento histórico, tendencias temporales.

¿Qué pasa si no existe Oro?

- Cada analista implementa su propia definición de churn: Marketing dice 15%, Producto dice 22%
- Las métricas cambian según quién las calcula: Nadie confía en los números
- No hay gobierno de datos: Imposible auditar o certificar métricas
- Las decisiones de negocio se toman sobre arena: Cada área tiene su propia versión de la realidad

Oro no es staging, es **modelo analítico** con reglas de negocio centralizadas.

```
INSERT OVERWRITE fact_suscripciones_cuentas WITH base AS ( SELECT id_cuenta, fecha_fin, LEAD(id_suscripcion) OVER (
PARTITION BY id_cuenta ORDER BY fecha_inicio ) AS id_suscripcion_siguiente FROM core_negocio.cuentas_suscripciones )
SELECT id_cuenta, CASE WHEN fecha_fin IS NOT NULL AND id_suscripcion_siguiente IS NULL THEN fecha_fin ELSE NULL END
AS fecha_churn FROM base;
```

Oro es donde la arquitectura se convierte en valor de negocio.

CONSUMO

Consumo: el negocio interactúa con aplicaciones

Acá el negocio consume insights a través de Databricks Apps, no consulta tablas directamente.

Por qué existe esta capa

Desacoplar el consumo del modelo. El negocio interactúa con aplicaciones, no con tablas.

Decisión arquitectónica: Databricks Apps sobre Unity Catalog: seguridad, gobierno y experiencia de usuario en una sola capa.

Métricas de Churn y Análisis Dinámico

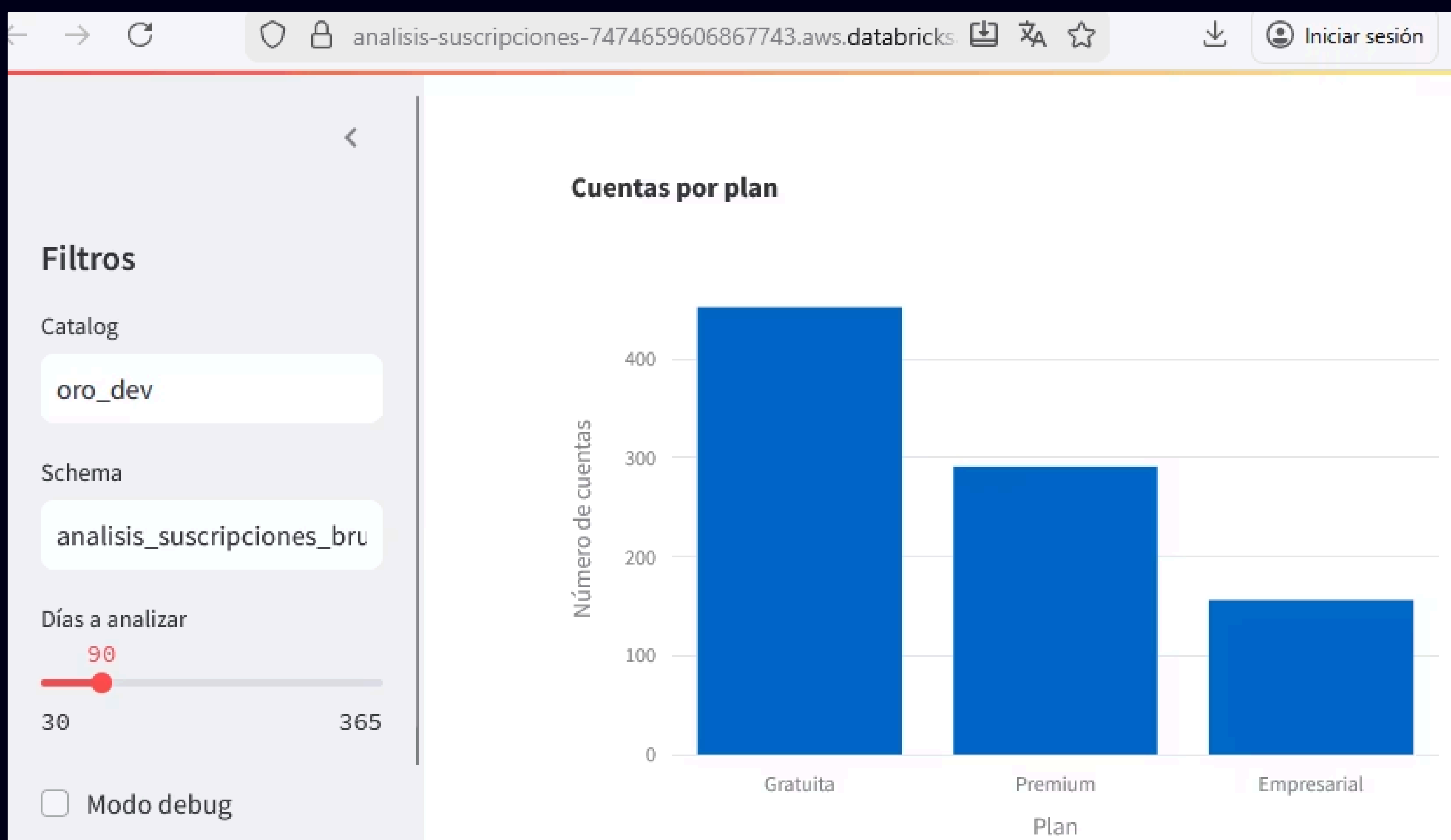
Visualización interactiva de la retención y pérdida de clientes, con filtros dinámicos para explorar los datos.

Gobernanza

Seguridad y control de acceso garantizados con Unity Catalog.

La complejidad vive en oro. El consumo es simple.

La lógica compleja reside en la capa Oro, y las aplicaciones Databricks exponen esos resultados de manera intuitiva y visual, escalando el acceso a los datos de forma consistente.

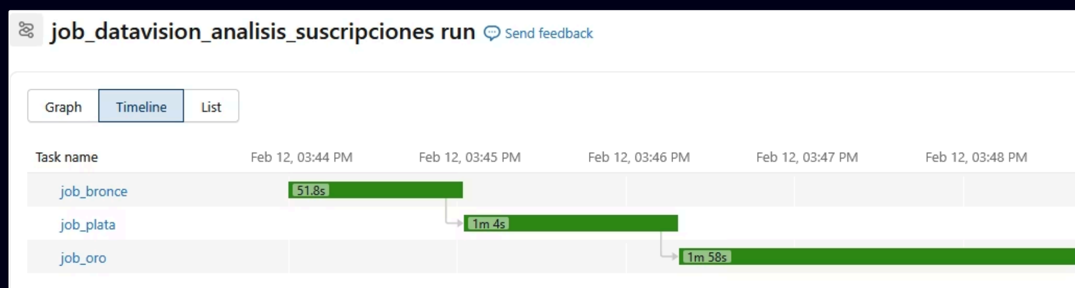


ORQUESTACIÓN

Automatización: de frágil a confiable

Por qué existe orquestación

Sin automatización, todo es manual. Sin idempotencia, todo es frágil.



Idempotencia

Si reejecutamos una fecha, el resultado es consistente.

Job parameters

catalogo_bronce	bronce_dev
catalogo_oro	oro_dev
catalogo_plata	plata_dev
fecha_proceso	{{job.start_time.year}}-{{job.start_time.month}}-{{job.start_time.day}}

Error común: Scripts que funcionan una vez pero fallan o duplican datos en reejecución.

01

Workflows orquestados

Bronce → Plata → Oro con dependencias

02

Parametrización por fecha

Reprocesamiento consistente

03

MERGE para idempotencia

Sin duplicados en reejecuciones

04

Validaciones entre capas

Calidad garantizada

Buena práctica: MERGE para idempotencia. Parametrización por fecha. Validaciones entre capas.

La diferencia entre un pipeline que funciona y uno confiable está en la orquestación.

CIERRE

La diferencia entre usar herramientas y entender arquitectura

Si estás construyendo pipelines sin pensar en estas decisiones, estás acumulando deuda técnica.

Un pipeline bien pensado

No solo "funciona", es confiable. Cada decisión, desde la extracción, tiene un propósito claro según el tipo y volumen de datos.

Cada capa tiene un propósito

Saltarse una no es un atajo; es una decisión con consecuencias. Bronce preserva, Plata transforma, Oro modela. Cada una existe por una razón.

De "en verde" a "bien diseñado"

La diferencia no está en que el pipeline corra, sino en cómo está diseñado. Un diseño sólido te da robustez, consistencia y capacidad de reejecución. Eso es confiabilidad real.

Qué aprendiste en este recorrido:

- **Extracción:** Volumen y frecuencia determinan tu estrategia. Ignorarlo es un riesgo.
- **Bronze:** Reprocesabilidad es tu seguro de vida. Sin inmutabilidad, no hay vuelta atrás.
- **Silver:** Consistencia antes de modelar. La limpieza es la base de todo análisis.
- **Gold:** Reglas de negocio centralizadas. Sin esto, cada reporte es una isla.
- **Consumo:** Desacoplar experiencia de modelo. No mezcles peras con manzanas.
- **Orquestación:** Idempotencia es la diferencia entre frágil y confiable. ¿Puedes reejecutar?

Ahora entendés el por qué, no solo el cómo. Eso cambia cómo diseñás pipelines.

CIERRE

De entender a implementar

La diferencia entre implementar y copiar código está en entender estas decisiones.

Profundizar en cada capa

Cada decisión arquitectónica tiene trade-offs. Entender el por qué es más importante que memorizar el cómo. Lo que no entendés hoy es deuda técnica mañana.

Aplicar estos conceptos

El mejor aprendizaje es implementar. No esperes el proyecto perfecto. Empezá con un caso simple y construí capa por capa, con criterio.

Seguir aprendiendo

La arquitectura de datos evoluciona, pero los principios se mantienen. Mantenete actualizado con casos reales y decisiones fundamentadas.

CHECKLIST

¿Tu pipeline está bien diseñado? Checklist de auto-evaluación

Respondé estas preguntas con honestidad. No se trata de tener todo perfecto, sino de saber dónde estás parado.

- [ ] ¿Podés reprocesar sin tocar el origen? (Bronze)
- [ ] ¿Todos los equipos reportan las mismas métricas? (Gold)
- [ ] ¿Los dashboards se rompen cuando cambia el origen? (Silver)
- [ ] ¿Cuánto tardás en debuggear un número incorrecto? (Separación de capas)
- [ ] ¿Podés reejecutar una fecha sin duplicados? (Orquestación)
- [ ] ¿Las validaciones están centralizadas? (Silver)
- [ ] ¿El negocio consulta tablas directamente? (Consumo)

Si respondiste NO a 3 o más: tu pipeline funciona, pero no está bien diseñado.

Esta checklist es tu punto de partida. Cada NO es una oportunidad de mejora.