Documentación Técnica: Pipeline de Datos de Películas TMDB

Índice

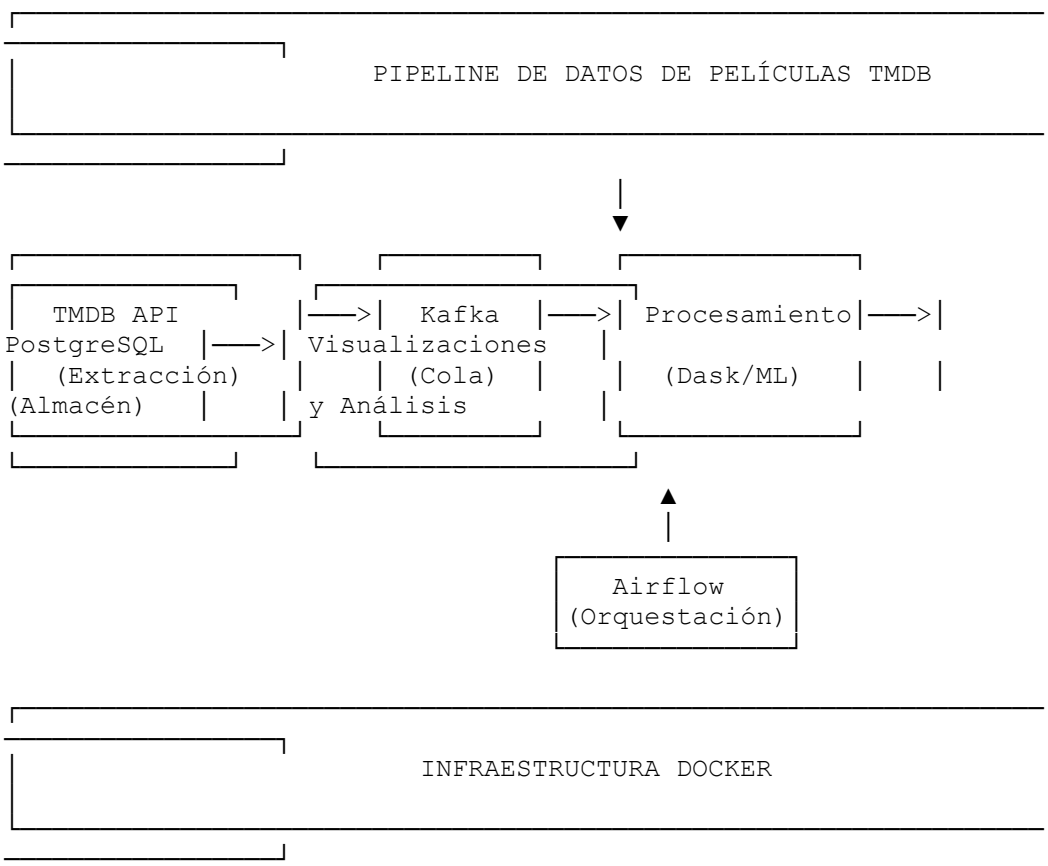
Introducción y Visión General

Este documento detalla un pipeline completo de datos para películas usando la API de TMDB (The Movie Database). El sistema está diseñado para extraer, procesar, almacenar y analizar datos de películas, implementando una arquitectura basada en eventos que aprovecha tecnologías modernas como Kafka, Docker, PostgreSQL, Dask y Airflow.

El objetivo principal es crear un sistema escalable y robusto que proporcione insights valiosos sobre tendencias cinematográficas, popularidad de géneros, y factores de éxito comercial. El pipeline abarca desde la extracción de datos crudos hasta el entrenamiento de modelos de Machine Learning para predecir el éxito de películas.

Arquitectura del Sistema

La arquitectura implementada sigue un modelo de procesamiento basado en eventos con múltiples capas interconectadas:



Infraestructura Docker

Todo el sistema está containerizado utilizando Docker y Docker Compose, con los siguientes servicios:

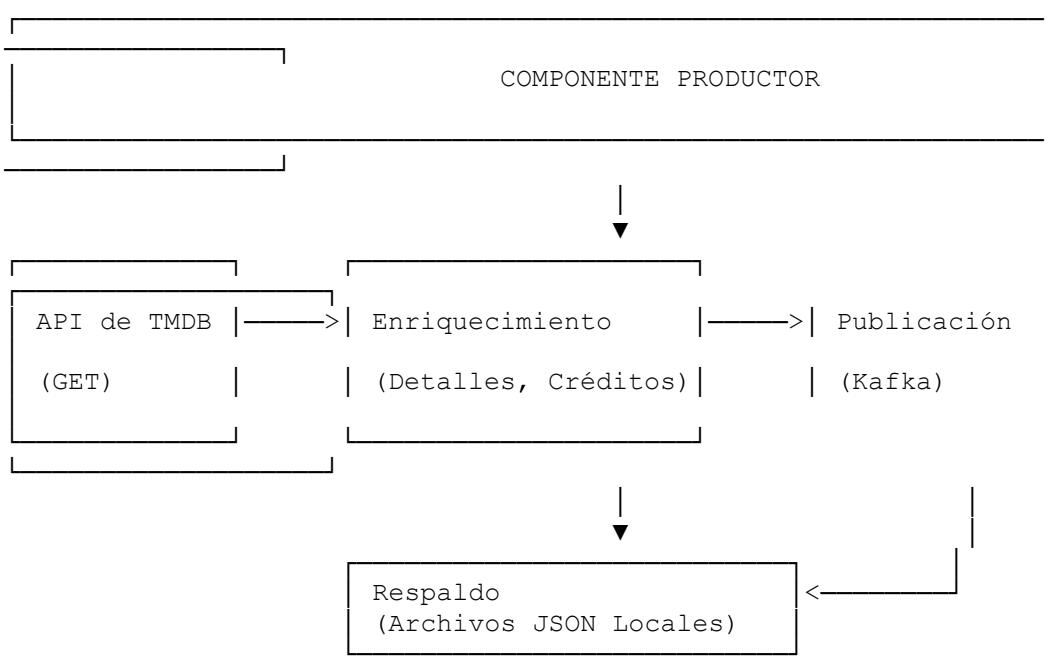
- **Zookeeper**: Coordinación y gestión de Kafka
- **Kafka**: Sistema de mensajería distribuido
- **PostgreSQL (2 instancias)**: Base de datos principal para películas y metadatos de Airflow
- **Redis**: Backend para el ejecutor Celery de Airflow
- **Airflow (Webserver, Scheduler, Worker)**: Orquestación del pipeline
- **Red Docker**: Red dedicada "movie_pipeline" para facilitar la comunicación entre servicios

Componentes Principales y Flujo de Datos

El pipeline está compuesto por varios componentes que trabajan juntos para procesar los datos de películas:

1. Productor (`producer.py`)

El productor es responsable de extraer datos de la API de TMDB y enviarlos a Kafka:

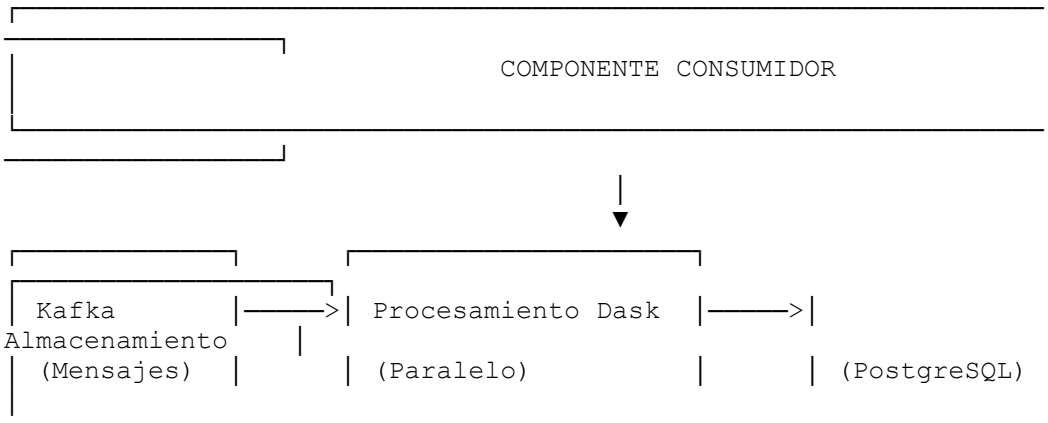


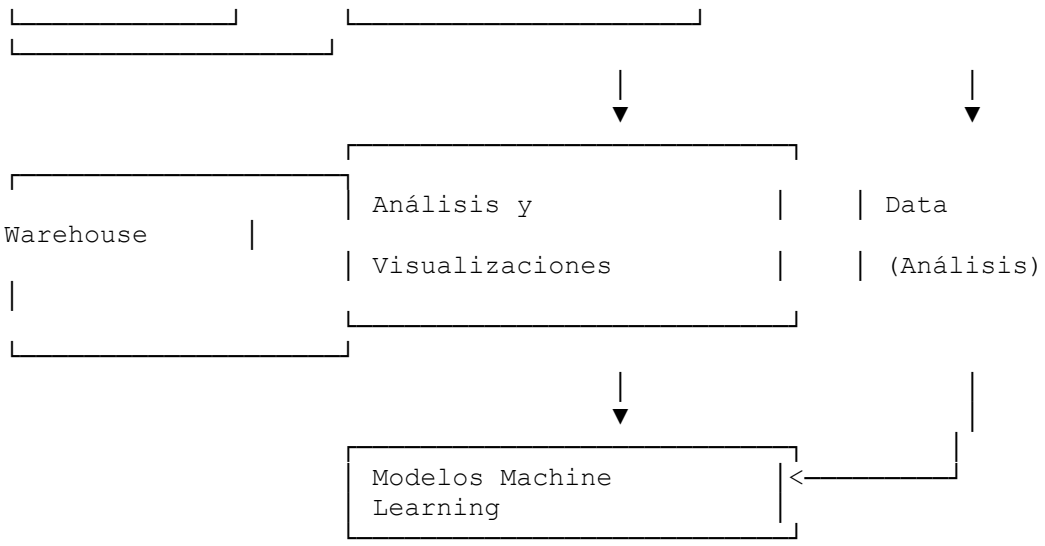
Funcionalidades principales:

- Autenticación con la API de TMDB
- Obtención de películas populares paginadas
- Enriquecimiento con detalles adicionales (presupuesto, ingresos, etc.)
- Publicación en tópico de Kafka "tmdb_data"
- Almacenamiento local de respaldos JSON

2. Consumidor (`consumer.py`)

El consumidor procesa los mensajes de Kafka utilizando Dask para procesamiento paralelo:



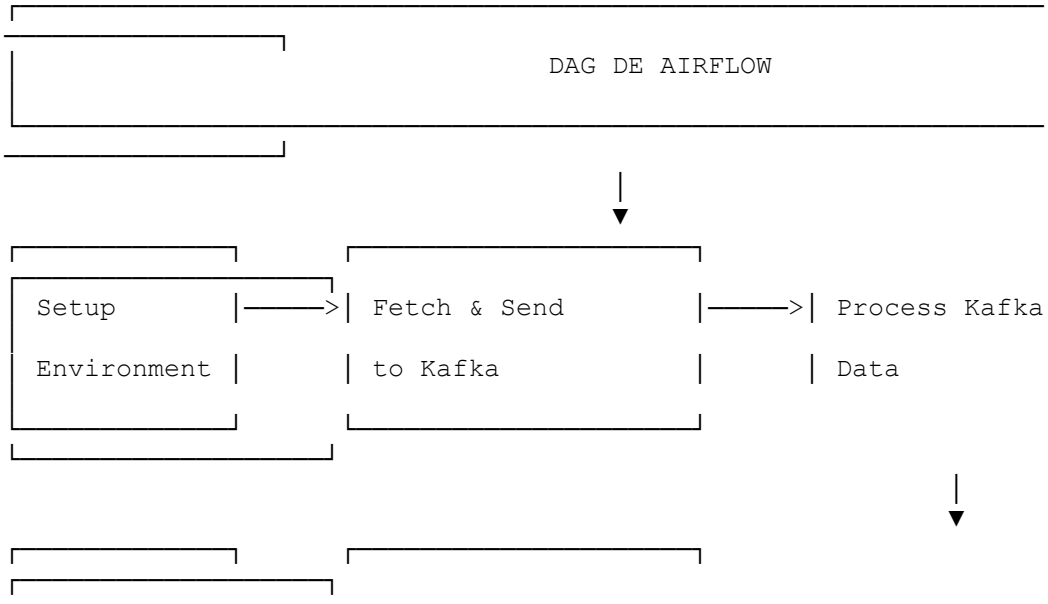


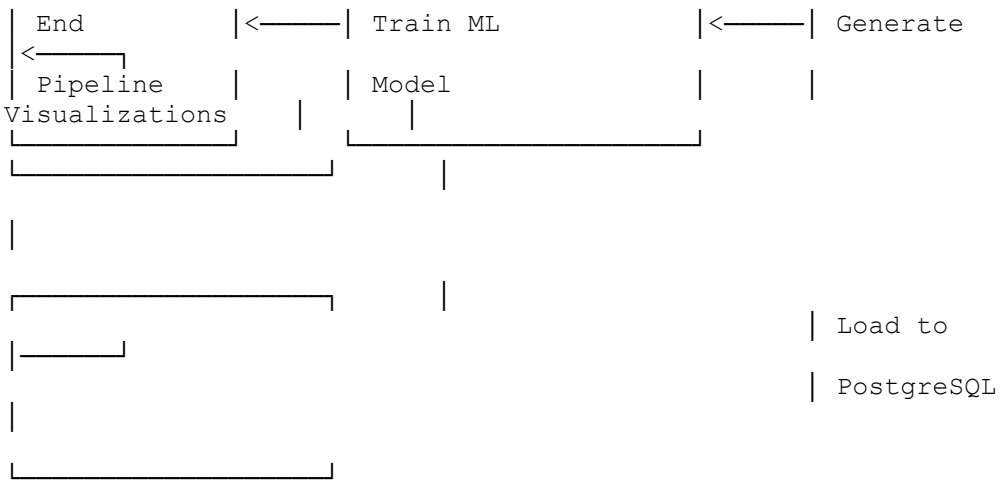
Funcionalidades principales:

- Consumo de mensajes desde Kafka
- Procesamiento paralelo con Dask
- Limpieza y transformación de datos
- Almacenamiento en esquema relacional PostgreSQL
- Creación de data warehouse desnormalizado
- Generación de visualizaciones
- Entrenamiento de modelos predictivos

3. DAG de Airflow (`tmdb_pipeline.py`)

El DAG orquesta el flujo completo del pipeline:





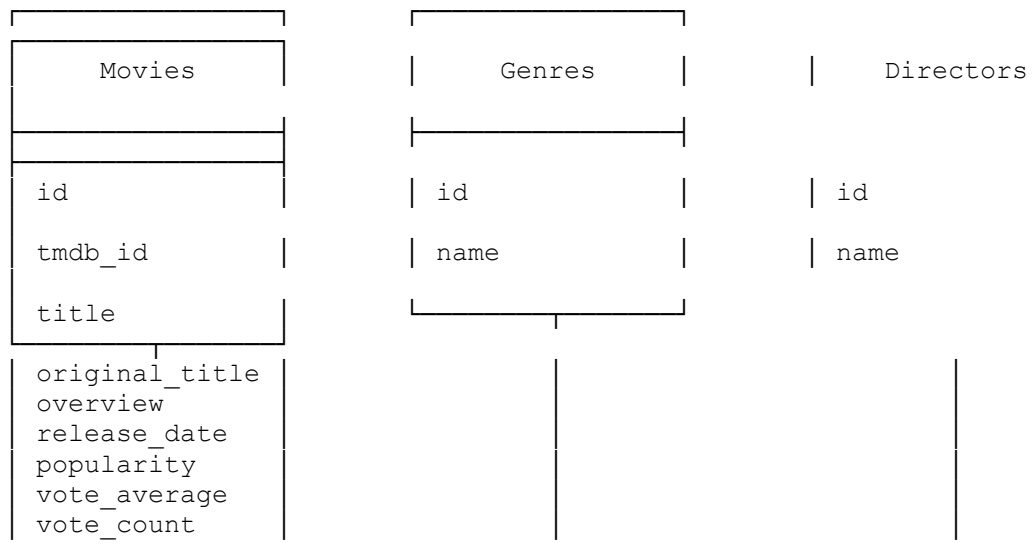
Funcionalidades principales:

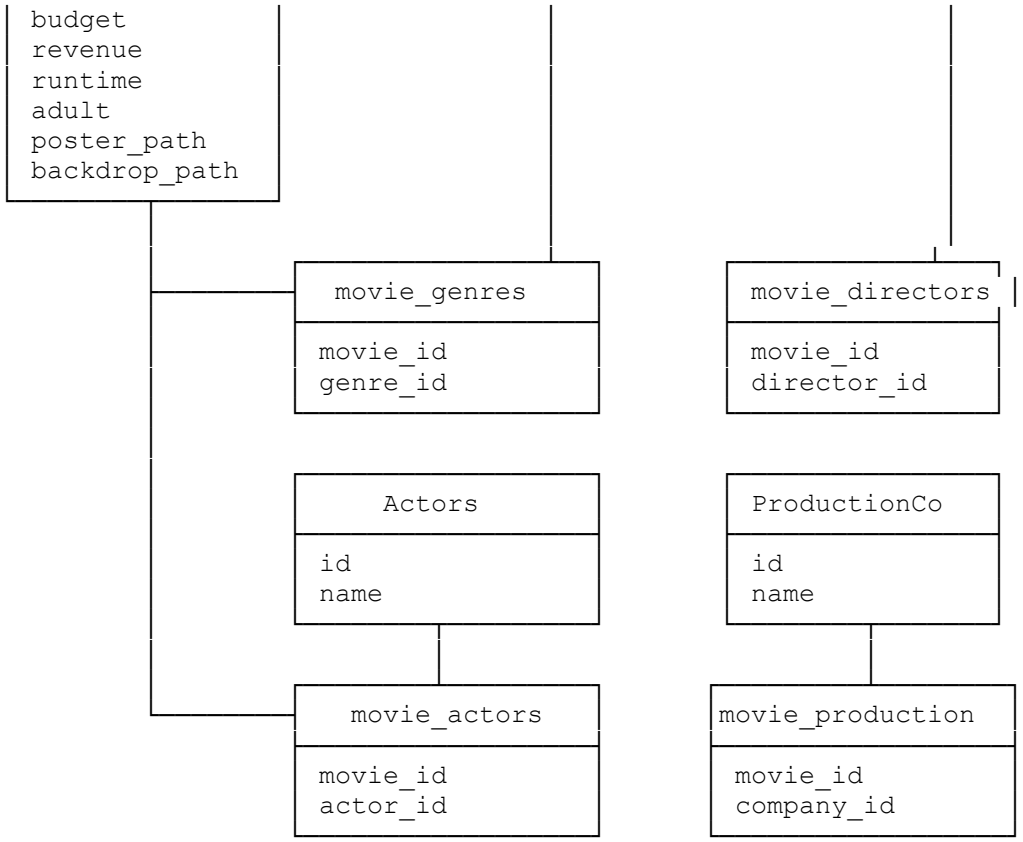
- Orquestación completa del flujo de trabajo
- Manejo de dependencias entre tareas
- Programación de ejecuciones (cada 12 horas)
- Reintentos automáticos en caso de fallos
- Alternativas para componentes que fallen
- Interfaz web para monitoreo y control

Esquema de Base de Datos

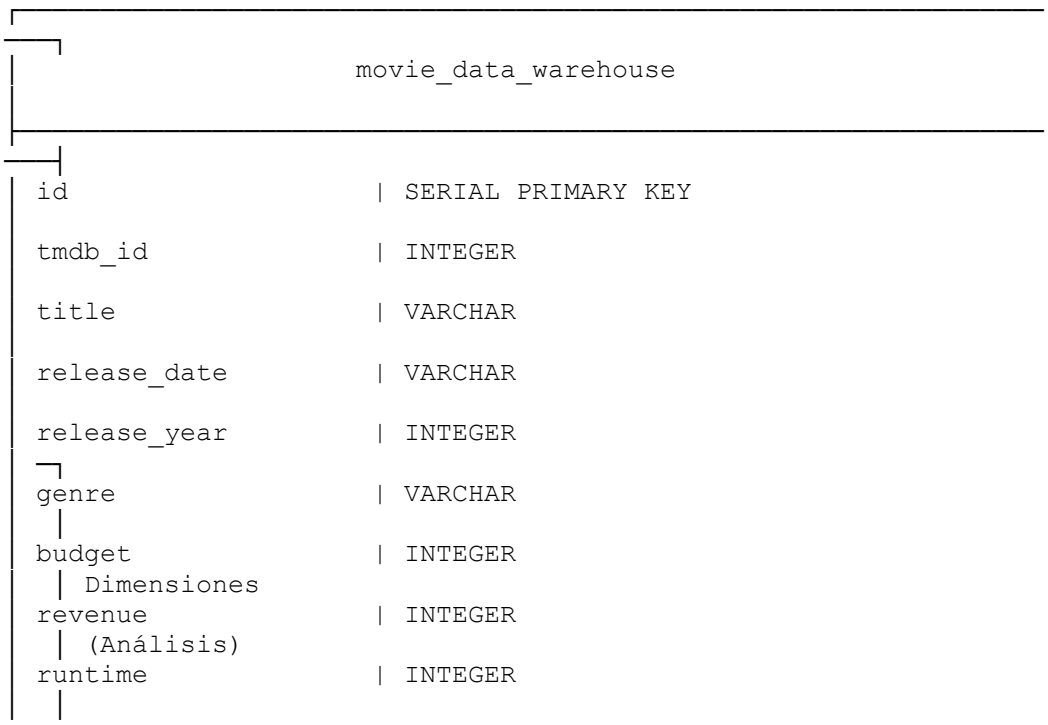
El modelo de datos implementa un diseño relacional normalizado para almacenamiento operativo y un esquema desnormalizado tipo estrella para el data warehouse:

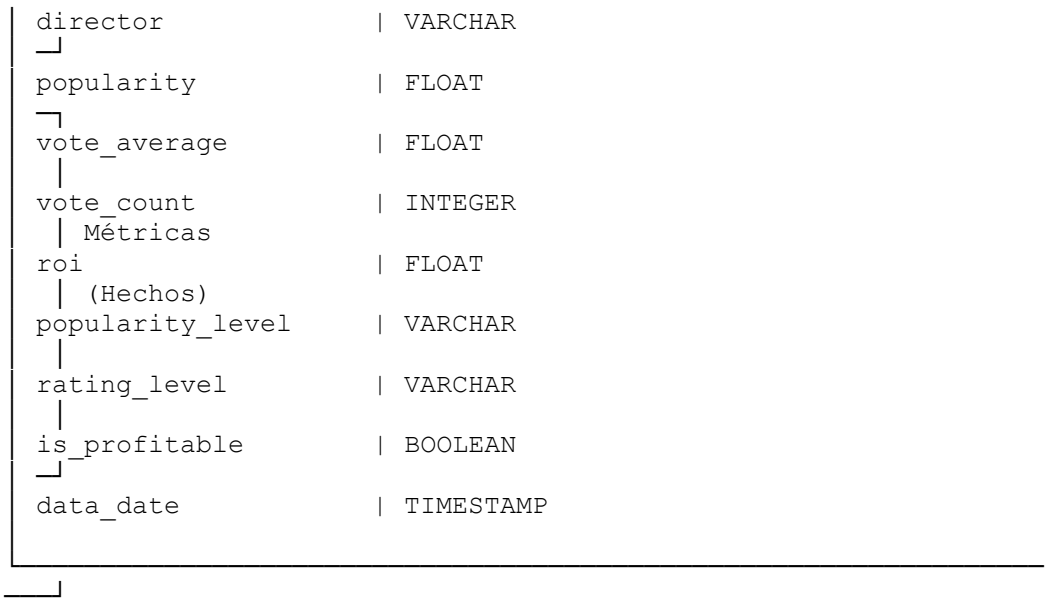
Esquema Relacional (Normalizado)





Data Warehouse (Desnormalizado)





Transformaciones de Datos

El pipeline realiza múltiples transformaciones a medida que los datos fluyen a través del sistema:

1. Transformación de Extracción (Productor)

Transforma la estructura anidada y compleja de la API TMDB en un formato plano más procesable:

Entrada (API TMDB):

```
{
  "id": 550,
  "title": "Fight Club",
  "genres": [{"id": 18, "name": "Drama"}, {"id": 53, "name":
"Thriller"}],
  "production_companies": [{"id": 508, "name": "Regency
Enterprises"}, ...],
  "credits": {
    "crew": [{"id": 7467, "name": "David Fincher", "job":
"Director"}, ...],
    "cast": [{"id": 819, "name": "Edward Norton", "character":
"The Narrator"}, ...]
  }
}
```

Salida (Mensaje Kafka):

```
{
  "id": 550,
```

```
  "title": "Fight Club",
  "original_title": "Fight Club",
  "overview": "A ticking-time-bomb insomniac...",
  "release_date": "1999-10-15",
  "popularity": 68.626,
  "vote_average": 8.4,
  "vote_count": 24408,
  "budget": 63000000,
  "revenue": 100853753,
  "runtime": 139,
  "genres": ["Drama", "Thriller"],
  "production_companies": ["Regency Enterprises", "Fox 2000
Pictures"],
  "directors": ["David Fincher"],
  "cast": ["Edward Norton", "Brad Pitt", "Helena Bonham Carter"]
}
```


2. Transformación de Procesamiento (Consumidor)

Limpia, estandariza y deriva nuevas métricas a partir de los datos crudos:

Transformaciones aplicadas:

- Manejo de valores nulos y faltantes
- Corrección de tipos de datos
- Cálculo de ROI: (revenue - budget) / budget (si budget > 0)
- Categorización de popularidad: Alta (>20), Media (>10), Baja (>5), Muy Baja (<5)
- Clasificación por calificación: Excelente (≥8), Buena (≥6), Regular (≥4), Mala (<4)
- Normalización en entidades separadas (películas, géneros, directores, etc.)

3. Transformación para Data Warehouse

Desnormaliza los datos para optimizar consultas analíticas:

Transformaciones aplicadas:

- Creación de una entrada por cada combinación película-género-director
- Extracción del año de lanzamiento desde la fecha
- Cálculo de indicador de rentabilidad (is_profitable)
- Aplicación de categorías predefinidas de popularidad y calificación

4. Transformación para Machine Learning

Prepara los datos para el entrenamiento de modelos predictivos:

Transformaciones aplicadas:

- Selección de features relevantes (presupuesto, runtime, votos, etc.)
- Creación de target combinado: éxito = 0.7*popularidad* + *0.3*(calificación*10)
- División en conjuntos de entrenamiento (80%) y prueba (20%)
- Normalización de características para el modelo

Decisiones Técnicas y Sus Justificaciones

1. Arquitectura Basada en Eventos con Kafka

Decisión: Adoptar una arquitectura basada en eventos con Kafka como intermediario.

Justificación: Esta decisión proporciona múltiples beneficios fundamentales para el sistema:

El desacoplamiento entre productores y consumidores permite que operen de forma independiente. Si el productor o consumidor falla o necesita reiniciarse, el otro componente puede continuar operando sin interrupciones. Esto mejora significativamente la resiliencia del sistema, ya que el fallo de un componente no causa la caída de todo el pipeline.

Kafka actúa como un buffer de datos efectivo, permitiendo que el productor y el consumidor operen a diferentes velocidades. Cuando el productor genera datos más rápido que la capacidad de procesamiento del consumidor, Kafka almacena temporalmente el exceso, evitando pérdidas de datos o sobrecarga en el consumidor. Esta característica es especialmente valiosa durante picos de actividad o cuando el consumidor está temporalmente inactivo.

La arquitectura basada en Kafka permite escalar horizontalmente. Al aumentar el número de particiones en un tópico y añadir múltiples instancias de consumidores, podemos procesar datos en paralelo, incrementando significativamente el rendimiento sin cambiar el código base. Esta escalabilidad es crucial para manejar volúmenes crecientes de datos a largo plazo.

Los datos permanecen en Kafka hasta que son procesados correctamente, lo que permite una recuperación robusta tras fallos. Si un consumidor falla durante el procesamiento, puede reiniciarse y continuar desde donde se quedó, garantizando que ningún dato se pierda debido a interrupciones temporales. Esta característica es esencial para mantener la integridad y completitud de los datos.

2. Procesamiento Distribuido con Dask

Decisión: Utilizar Dask para el procesamiento paralelo en lugar de Pandas estándar.

Justificación: Dask ofrece ventajas considerables para el procesamiento de datos:

El rendimiento se mejora significativamente al distribuir el procesamiento en múltiples núcleos o incluso en un clúster. Las operaciones que serían secuenciales en Pandas se ejecutan en paralelo con Dask, reduciendo dramáticamente los tiempos de procesamiento para grandes volúmenes de datos. Esto es crucial para mantener la eficiencia del pipeline a medida que escala.

Dask puede procesar conjuntos de datos que exceden la memoria RAM disponible. A diferencia de Pandas, que requiere que todo el dataset quepa en memoria, Dask divide los datos en fragmentos manejables (particiones) y los procesa de forma incremental. Esta capacidad es esencial para analizar grandes colecciones de películas sin requerir hardware costoso.

La API de Dask es intencionalmente similar a Pandas, lo que facilita enormemente la transición y reduce la curva de aprendizaje. Los desarrolladores familiarizados con Pandas pueden comenzar a utilizar Dask con cambios mínimos en su código existente, manteniendo la productividad durante la implementación.

Dask permite escalar desde un portátil hasta un clúster sin cambiar significativamente el código. El mismo pipeline puede ejecutarse localmente durante el desarrollo y desplegarse en un entorno distribuido en producción, proporcionando una ruta de escalabilidad gradual conforme crecen las necesidades de procesamiento.

3. Esquema Normalizado vs. Data Warehouse

Decisión: Implementar un esquema normalizado para almacenamiento operativo y un data warehouse desnormalizado para análisis.

Justificación: Esta dualidad en el modelado de datos ofrece lo mejor de ambos mundos:

El esquema normalizado evita redundancias y mantiene la integridad referencial. Al separar datos en tablas como películas, géneros, directores y actores, eliminamos la duplicación y garantizamos que cada entidad se almacene una sola vez. Esto reduce el espacio de almacenamiento necesario y previene inconsistencias cuando los datos se actualizan.

El data warehouse desnormalizado está optimizado para consultas analíticas complejas. Al incluir dimensiones como género, director y tiempo junto con las métricas relevantes en una sola tabla, eliminamos la necesidad de costosas operaciones JOIN. Esto acelera significativamente las consultas analíticas, mejorando la experiencia de usuario y permitiendo análisis más complejos y frecuentes.

La separación de responsabilidades permite que cada esquema se optimice para su propósito específico. El esquema normalizado se enfoca en la correcta inserción y actualización de datos, mientras que el data warehouse prioriza la velocidad de consulta y la facilidad de análisis. Esta especialización mejora el rendimiento global del sistema.

Este enfoque dual permite que ambos esquemas evolucionen según sus propias necesidades. Podemos modificar el esquema normalizado para acomodar nuevos tipos de relaciones entre entidades sin afectar las consultas analíticas existentes. Igualmente, podemos añadir nuevas métricas o dimensiones al data warehouse sin alterar la estructura operacional.

4. Containerización con Docker

Decisión: Containerizar toda la infraestructura con Docker y Docker Compose.

Justificación: La containerización proporciona numerosos beneficios para el desarrollo y despliegue:

Docker garantiza que el entorno sea idéntico en cualquier máquina, eliminando el clásico problema "en mi máquina funciona". Todos los servicios (Kafka, PostgreSQL, Airflow) se ejecutan con las mismas versiones y configuraciones independientemente del sistema operativo subyacente, reduciendo significativamente los problemas de compatibilidad.

Cada servicio opera en su propio contenedor sin interferencias de otros componentes. Este aislamiento evita conflictos de dependencias y facilita la depuración al delimitar claramente los límites de cada servicio. Los problemas en un componente no afectan a otros, mejorando la estabilidad general del sistema.

Docker Compose simplifica enormemente la gestión de dependencias y versiones. Todas las configuraciones de servicios se definen en un único archivo declarativo, facilitando el versionado y documentando implícitamente las relaciones entre componentes. Esto reduce la sobrecarga administrativa y los errores de configuración.

La transición entre entornos de desarrollo y producción se vuelve más suave. Los mismos contenedores utilizados durante el desarrollo pueden desplegarse en producción, minimizando las sorpresas y reduciendo el tiempo de implementación. Esto es particularmente valioso para mantener la consistencia en el ciclo de vida del software.

La containerización facilita notablemente el despliegue en diferentes sistemas operativos. El proyecto puede ejecutarse igualmente bien en Linux, macOS (como en este caso con un MacBook Pro) o Windows, ampliando significativamente la base de desarrolladores y entornos de despliegue potenciales.

5. Orquestación con Airflow

Decisión: Utilizar Airflow para orquestar el pipeline completo.

Justificación: Airflow proporciona capacidades avanzadas de orquestación que benefician al proyecto:

Airflow permite definir programaciones complejas basadas en tiempo o eventos, automatizando completamente la ejecución del pipeline. Podemos configurar ejecuciones periódicas (cada 12 horas en este caso) o en respuesta a eventos específicos, eliminando la necesidad de intervención manual y asegurando la frescura de los datos.

Las dependencias entre tareas se gestionan eficientemente a través de un grafo acíclico dirigido (DAG). Airflow garantiza que cada tarea se ejecute solo cuando todas sus dependencias previas se han completado con éxito, manteniendo la integridad del flujo de datos y evitando inconsistencias por ejecuciones parciales.

Airflow implementa políticas sofisticadas de reintento para tareas fallidas. Si una tarea falla, Airflow puede reintentar automáticamente su ejecución según parámetros configurables, mejorando la resiliencia del pipeline frente a fallos transitorios como problemas de red o indisponibilidad temporal de servicios externos.

La interfaz web de Airflow proporciona un panel completo para monitorear el progreso y revisar logs. Podemos visualizar la ejecución de tareas en tiempo real, identificar cuellos de botella, analizar tiempos de ejecución históricos y acceder rápidamente a logs detallados para diagnóstico. Esta visibilidad mejora significativamente la operabilidad del sistema.

Airflow es altamente extensible, permitiendo integrar herramientas externas y APIs a través de operadores personalizados. Esto facilita la incorporación de nuevos componentes o servicios en el futuro sin modificar la estructura fundamental del pipeline, proporcionando un marco flexible que puede evolucionar con las necesidades del proyecto.

Archivos Generados y su Propósito

El pipeline genera varios tipos de archivos en diferentes etapas del procesamiento:

Productor (`producer.py`)

1. **Archivos JSON de Películas Individuales**:
2. `data/movie_analytics/movie_{id}.json`

Estos archivos almacenan los datos completos de cada película extraída de la API. Sirven como respaldo persistente y pueden utilizarse para recuperación en caso de fallos en el pipeline. Contienen toda la información enriquecida, incluyendo detalles, géneros, directores y elenco.

Consumidor (`consumer.py`)

1. **Visualizaciones por Lote**:
2. `data/movie_analytics/batch_{número}/top_peliculas_popularidad.png`
3. `data/movie_analytics/batch_{número}/generos_populares.png`
4. `data/movie_analytics/batch_{número}/generos_valoracion.png`
5. `data/movie_analytics/batch_{número}/analisis_warehouse.png`

   Estos archivos PNG contienen gráficos que visualizan diferentes aspectos de los datos (películas más populares, distribución de géneros, etc.). Se organizan por lotes de procesamiento, permitiendo un seguimiento histórico de tendencias.

6. **Modelos de Machine Learning**:
7. `data/movie_analytics/batch_{número}/movie_success_model.pkl`
8. `data/movie_analytics/batch_{número}/feature_importance.csv`
9. `data/movie_analytics/batch_{número}/feature_importance.png`

   El archivo PKL contiene el modelo serializado para predicciones. El CSV documenta la importancia de cada feature, y el PNG la visualiza gráficamente. Estos archivos permiten utilizar el modelo posteriormente sin necesidad de reentrenarlo.

DAG de Airflow (`tmdb_pipeline.py`)

1. **Archivos Temporales**:
2. `data/movie_analytics/movies_data.json`
3. `data/movie_analytics/processed_movies_v2.csv`

   Estos archivos intermedios contienen datos extraídos y procesados durante la ejecución del DAG. Sirven como puente entre tareas y facilitan la depuración del proceso.

4. **Visualizaciones con Marcas de Tiempo**:
5. `data/movie_analytics/visualizations_v2_{fecha}_{hora}/{nombre_visualización}.png`
6. `data/movie_analytics/visualizations_v2_{fecha}_{hora}/report.txt`

   Visualizaciones generadas por el DAG con marcas de tiempo para identificar cada ejecución. El archivo report.txt contiene un resumen textual de los hallazgos del análisis.

7. **Modelos ML con Marcas de Tiempo**:
8. `data/movie_analytics/ml_model_v2_{fecha}_{hora}/model.pkl`
9. `data/movie_analytics/ml_model_v2_{fecha}_{hora}/metrics.json`
10. `data/movie_analytics/ml_model_v2_{fecha}_{hora}/feature_importance.png`

    Archivos relacionados con el modelo de Machine Learning generado por el DAG, identificados con marca de tiempo para facilitar el versionado y la comparación entre ejecuciones.

Desafíos Técnicos y Soluciones

Desafío 1: Gestión de Dependencias en Docker

Problema: Los contenedores de Airflow predefinidos no incluían todas las bibliotecas Python necesarias para el pipeline.

Solución implementada: Se desarrolló el script `install_deps.sh` para instalar dependencias adicionales en los contenedores ya creados, evitando la necesidad de construir imágenes personalizadas. Adicionalmente, se configuraron variables `_PIP_ADDITIONAL_REQUIREMENTS` en docker-compose para instalar dependencias clave durante la creación inicial del contenedor. Esta aproximación mantuvo la simplicidad del despliegue mientras aseguraba que todas las dependencias estuvieran disponibles.

Desafío 2: Conectividad entre Servicios en Docker

Problema: Los servicios en Docker tenían dificultades para comunicarse entre sí, especialmente Kafka y los consumidores.

Solución implementada: Se configuró una red Docker dedicada (`movie_pipeline`) para todos los servicios. Para Kafka, se implementaron diferentes configuraciones de listeners para conexiones internas (INSIDE) y externas (OUTSIDE), permitiendo que tanto los servicios dentro de Docker como las aplicaciones en la máquina host pudieran conectarse. Además, se añadieron verificaciones de salud (healthchecks) para garantizar que los servicios dependientes solo intentaran conectarse cuando el servicio requerido estuviera completamente operativo.

Desafío 3: Manejo de Fallos en Componentes Distribuidos

Problema: En un sistema distribuido, cualquier componente puede fallar, lo que podría interrumpir todo el pipeline.

Solución implementada: Se implementó una estrategia de "defensa en profundidad" con múltiples mecanismos de resiliencia:

1. Sistema de reintentos en el productor para manejar fallos temporales de la API
2. Reintentos automáticos configurados en las tareas de Airflow
3. Caminos alternativos para cada componente (usando datos de ejemplo si Kafka falla)
4. Copias locales de datos en varios puntos del pipeline para permitir recuperación
5. Gestión robusta de excepciones con logging detallado

Esta estrategia garantiza que el pipeline pueda continuar funcionando incluso si algunos componentes fallan temporalmente.

Desafío 4: Procesamiento Eficiente de Grandes Volúmenes de Datos

Problema: El procesamiento secuencial de grandes volúmenes de datos podría ser ineficiente y causar cuellos de botella significativos.

Solución implementada: Se implementó un enfoque de procesamiento paralelo y por lotes utilizando Dask:

1. **Procesamiento distribuido con Dask**: Se configuró un cliente Dask para distribuir la carga de trabajo entre múltiples núcleos, aprovechando toda la capacidad de procesamiento disponible. El código incluye:

```
self.client = Client(n_workers=4, threads_per_worker=2, memory_limit='2GB')
```

3. **Procesamiento por lotes**: Los datos se agrupan en lotes (batches) antes de procesarlos, minimizando la sobrecarga asociada con operaciones frecuentes de E/S y comunicación con la base de datos.

```
if len(buffer) >= 10:  # Procesar cada 10 películas
    processed_df = self.process_batch(buffer)
```

6. **Particionamiento de datos**: Los dataframes de Dask se dividen en particiones que se procesan en paralelo:

```
ddf = dd.from_pandas(df, npartitions=4)
ddf = ddf.map_partitions(self.clean_dataframe)
```

9. **Optimización de operaciones de base de datos**: Se implementaron transacciones por lotes y verificaciones para evitar duplicados, reduciendo significativamente el número de operaciones de base de datos.

Esta estrategia mejoró considerablemente el rendimiento, reduciendo los tiempos de procesamiento y permitiendo manejar volúmenes de datos mucho mayores con los mismos recursos.

Desafío 5: Manejo de Relaciones Complejas en PostgreSQL

Problema: Las relaciones muchos a muchos entre entidades (películas, géneros, directores, etc.) son complejas de gestionar eficientemente.

Solución implementada: Se adoptó un enfoque híbrido que combina SQLAlchemy ORM para la mayoría de las operaciones con acceso directo a través de psycopg2 cuando se necesita mayor control:

1. **Esquema relacional con SQLAlchemy ORM**: Se definieron clases de modelo con relaciones explícitas:

```
class Movie(Base):
    __tablename__ = 'movies'
    # ...
    genres = relationship("Genre", secondary="movie_genres", back_populates="movies")
```

6. **Tablas de unión para relaciones muchos a muchos**:

```
movie_genres = Table('movie_genres', Base.metadata,
    Column('movie_id', Integer, ForeignKey('movies.id'), primary_key=True),
    Column('genre_id', Integer, ForeignKey('genres.id'), primary_key=True)
)
```

11. **Verificaciones de existencia antes de insertar**:

```
genre = self.session.query(Genre).filter_by(name=genre_name).first()
if not genre:
    genre = Genre(name=genre_name)
    self.session.add(genre)
```

16. **Método alternativo con psycopg2**: Para casos donde SQLAlchemy no es óptimo:

```
def create_schema_and_tables_raw(self):
    conn = psycopg2.connect(...)
    cur = conn.cursor()
    cur.execute("CREATE TABLE IF NOT EXISTS...")
```

Este enfoque híbrido proporciona la flexibilidad del ORM para operaciones habituales, mientras mantiene la posibilidad de acceso directo SQL para operaciones más complejas o críticas en rendimiento.

Desafío 6: Compatibilidad entre Airflow y Componentes Externos

Problema: Integrar diferentes componentes (Kafka, Dask, PostgreSQL) en Airflow puede ser desafiante debido a diferencias en entornos y dependencias.

Solución implementada: Se adoptó un enfoque pragmático y orientado a la fiabilidad:

1. **Versión simplificada del procesamiento en el DAG**: Se implementó una versión alternativa utilizando Pandas en lugar de Dask para el DAG de Airflow, reduciendo la complejidad y dependencias:

```
# Procesar con Pandas (en vez de Dask)
logger.info(f"Procesando {len(messages)} mensajes con Pandas")
df = pd.DataFrame(messages)
```

5. **Mecanismos robustos de manejo de errores**:

```
try:
    # Intentar obtener datos de PostgreSQL
except Exception as e:
    logger.error(f"Error al leer datos de PostgreSQL: {e}")
     # Usar datos alternativos si falla
```

11. **Paso de información entre tareas mediante XComs**:

```
 kwargs['ti'].xcom_push(key='processed_data_path', value=output_file)
 processed_data_path = ti.xcom_pull(key='processed_data_path', task_ids='process_kafka_data')
```

14. **Almacenamiento de archivos intermedios** para facilitar la comunicación entre tareas:

```
 output_file = f"{OUTPUT_DIR}/processed_movies_v2.csv"
 df.to_csv(output_file, index=False)
```

Este enfoque asegura que el DAG pueda ejecutarse de manera confiable en el entorno de Airflow, manteniendo la compatibilidad con los componentes externos y proporcionando alternativas en caso de fallos.

Conclusión

El pipeline de datos de películas TMDB presenta una arquitectura moderna y robusta para la extracción, procesamiento, almacenamiento y análisis de datos cinematográficos. Las decisiones técnicas tomadas durante su diseño e implementación priorizan la escalabilidad, resiliencia y mantenibilidad del sistema.

La combinación de Kafka para mensajería, Dask para procesamiento paralelo, PostgreSQL para almacenamiento y Airflow para orquestación crea un sistema que puede crecer con las necesidades del proyecto, desde un entorno de desarrollo local hasta una implementación a gran escala.

Los desafíos técnicos encontrados durante el desarrollo se abordaron con soluciones pragmáticas que equilibran la complejidad con la fiabilidad, resultando en un pipeline que no solo es técnicamente sólido sino también práctico para el uso diario.

El esquema de base de datos híbrido, que combina un modelo normalizado para operaciones transaccionales con un data warehouse optimizado para análisis, proporciona la base para obtener insights valiosos sobre tendencias cinematográficas, preferencias de género y factores de éxito comercial.