Memoria del Proyecto: Spark Streaming

Título del Proyecto: Spark Streaming (Simulador en Tiempo Real)

Asignatura: Arquitecturas Especializadas

Fecha: Diciembre 2025

Integrantes:

- Samuel Corrionero (Arquitecto de Infraestructura)
- Ismael González Loro (Generador de Datos)
- Jairo Pabel Farfán Callau (Ingeniero Spark)
- Yahya El Baroudi El Ouazghari (Documentación)

Índice

Introducción

Este proyecto pone en marcha una arquitectura de streaming en tiempo real: generamos eventos sintéticos parecidos a los de una red social, los inyectamos en Kafka y los procesamos con Spark sobre una red de contenedores.

Decidimos apartarnos de los temas propuestos en clase para darle al proyecto un enfoque más diferente, novedoso y moderno. Quisimos montar una arquitectura realista basada en contenedores y Big Data, y eso es lo que se expone en este documento.

Diseño y Configuración de Infraestructura

Responsable: Samuel Corrionero Fernández

Samuel Corrionero asumió el papel de Arquitecto de Infraestructura. Su misión fue construir los cimientos sobre los que se apoya todo el proyecto. En un entorno de Big Data, el código no flota en el vacío; necesita servidores, redes y sistemas de mensajería robustos.

0. ¿Qué es Docker y por qué lo usamos?

Concepto: Docker como "Máquina Virtual Ligera" Docker es una herramienta que permite **empaquetar aplicaciones con todas sus dependencias** en contenedores aislados. Imagina que Docker es como una caja de transporte hermética. Dentro de esa caja metemos:

- La aplicación (Kafka, Zookeeper, Spark)
- El sistema operativo mínimo que necesita
- Todas las librerías y configuraciones

Ventaja: Esa caja funciona igual en tu PC, en el del compañero, o en un servidor de producción. No importa si tienes Windows, Mac o Linux; Docker se encarga de la compatibilidad.

¿Por qué la usamos en este proyecto?

1. **Reproducibilidad:** Todos los compañeros tienen el mismo entorno exacto. No hay confusiones tipo "a mí me funciona, ¿por qué a ti no?".
2. **Aislamiento:** Kafka y Zookeeper corren en sus propios contenedores sin interferir con tu sistema operativo.
3. **Facilidad:** En lugar de instalar Java, descargar Kafka, configurar todo manualmente (30 minutos de dolor), ejecutamos un comando: `docker compose up -d`. ¡Listo en 10 segundos!
4. **Escalabilidad:** Si necesitáramos 5 brokers de Kafka en lugar de 1, solo cambiaríamos el archivo de configuración. Sin Docker, sería una pesadilla.

Arquitectura de Docker en este Proyecto La arquitectura se organiza de la siguiente manera: En tu ordenador (Host), Docker crea dos contenedores aislados: uno para Zookeeper y otro para Kafka. Estos dos contenedores están conectados entre sí mediante una red interna privada de Docker, lo que permite que se comuniquen directamente entre ellos sin interferencias.

Sin embargo, para que tus aplicaciones Python (que corren en tu máquina local, fuera de Docker) puedan hablar con Kafka y Zookeeper, estos contenedores **exponen puertos hacia el exterior**. Zookeeper escucha en el puerto **2181** (desde tu perspectiva, es `localhost:2181`) y Kafka en el puerto **9092** (`localhost:9092`).

De esta forma, tu código Python no necesita saber que estas aplicaciones están dentro de contenedores; simplemente se conecta a `localhost:9092` como si estuvieran instaladas directamente en tu máquina. Docker maneja toda la magia de enrutamiento de red por debajo.

1. Tecnologías Implementadas (¿Qué son y por qué las usamos?)

Para entender la infraestructura, primero debemos definir las piezas clave que hemos desplegado utilizando contenedores Docker.

A. Apache Zookeeper (El Coordinador) Imaginemos que Kafka es una gran oficina de correos. **Zookeeper** sería el gerente de esa oficina. No reparte cartas (mensajes), pero sabe quién está trabajando, qué ventanillas están abiertas y mantiene el orden.

- **Función:** Gestiona el clúster, elige al nodo líder y guarda la configuración.
- **Necesidad:** Kafka no puede funcionar sin él (en versiones clásicas). Si Zookeeper cae, Kafka pierde el control.

B. Apache Kafka (El Broker / Buzón) Es el corazón del sistema de streaming. Funciona como una tubería de datos de altísima velocidad.

- **Concepto clave:** Desacoplamiento. El productor (Python) deja el mensaje en Kafka y se olvida. El consumidor (Spark) lo recoge cuando puede. No necesitan estar conectados directamente.
- **Topic:** Es como una "carpeta" o "canal" dentro de Kafka. Nosotros creamos uno llamado `tweets_topic` donde se vuelcan todos los mensajes simulados.

2. Despliegue con Docker Compose

En lugar de instalar Java, Kafka y Zookeeper manualmente en el ordenador de cada compañero (lo cual suele dar errores de versiones), creamos un archivo `docker-compose.yml`. Este archivo es una "receta" que le dice a Docker cómo levantar todo el entorno con un solo comando.

Fragmento del código de infraestructura (`docker-compose.yml`):

```yaml
services:
  # SERVICIO 1: ZOOKEEPER
  zookeeper:
    image: confluentinc/cp-zookeeper:7.4.4
    container_name: zookeeper
    ports:
      - "2181:2181"
    environment:
      ZOOKEEPER_CLIENT_PORT: 2181

  # SERVICIO 2: KAFKA
  kafka:
    image: confluentinc/cp-kafka:7.4.4
    container_name: kafka
    depends_on:
      - zookeeper  # Espera a que el jefe (Zookeeper) esté listo
    ports:
      - "9092:9092"
    environment:
      # Configuración crítica para que Spark (fuera de Docker) pueda hablar con Kafka (dent
      KAFKA_ADVERTISED_LISTENERS: PLAINTEXT://localhost:9092
      KAFKA_ZOOKEEPER_CONNECT: 'zookeeper:2181'
```

Explicación técnica:

- **`depends_on`**: Asegura que Kafka no arranque hasta que Zookeeper esté listo.
- **`KAFKA_ADVERTISED_LISTENERS`**: Este fue el mayor reto. Por defecto, Kafka anuncia su IP interna de Docker. Tuvimos que configurarlo para que anuncie `localhost:9092`, permitiendo así que los scripts de Python (que corren en el host, fuera de Docker)

puedan conectarse.

3. Comandos de Gestión y Operación

Para operar esta infraestructura, se definieron una serie de comandos que el equipo debía ejecutar.

Paso 1: Levantar la infraestructura

```
docker compose -f docker/docker-compose.yml up -d
```

El flag `-d` (detached) permite que los contenedores corran en segundo plano sin bloquear la terminal.

Paso 2: Verificar el estado

```
docker ps
```

Este comando nos confirma que los contenedores ***kafka*** *y* ***zookeeper*** *están en estado* ***Up****.*

Paso 3: Creación del Topic (Canal de datos) Una vez el sistema está arriba, hay que crear el canal donde viajarán los datos. Usamos `docker exec` para ejecutar el comando de creación *dentro* del contenedor de Kafka:

```
docker exec -it kafka kafka-topics --create \
    --bootstrap-server localhost:9092 \
    --replication-factor 1 \
    --partitions 1 \
    --topic tweets_topic
```

- `--topic tweets_topic`: El nombre del canal que usarán tanto el Productor como el Consumidor.

Paso 4: Detener / Apagar la infraestructura Cuando termines el desarrollo o quieras limpiar tu máquina, es importante detener los contenedores. Existen dos formas:

Opción A: Detener sin eliminar (mantiene los datos)

```
docker compose -f docker/docker-compose.yml stop
```

Los contenedores se pausan pero siguen existiendo. Puedes reanudarlos con `docker compose ... start`.

Opción B: Detener y eliminar (limpieza total)

```
docker compose -f docker/docker-compose.yml down
```

Este comando:

- Detiene todos los contenedores.
- Elimina los contenedores (no las imágenes base).
- Libera la red de Docker.

Nota importante: Si utilizas volúmenes persistentes (datos almacenados en disco), agrégale el flag para eliminarlos también:

```
docker compose -f docker/docker-compose.yml down -v
```

Verificar que todo se apagó:

```
docker ps
```

Debería mostrar una lista vacía (sin contenedores corriendo).

4. Validación de Infraestructura (Testing)

Para asegurar que todo funcionaba correctamente antes de que el resto del equipo empezara a desarrollar, Samuel creó un **script de prueba** (`tests/tester.py`). Este script actúa como un simulador de "salud del sistema".

¿Qué hace el Tester?

1. Actúa como **Productor**: Envía 5 mensajes de prueba a `tweets_topic`.
2. Actúa como **Consumidor**: Lee esos mismos mensajes para confirmar que todo fluye correctamente.

El código (versión simplificada):

```
from kafka import KafkaProducer, KafkaConsumer
import json

# Paso 1: PRODUCTOR - Enviar datos
producer = KafkaProducer(
    bootstrap_servers='localhost:9092',
    value_serializer=lambda v: json.dumps(v).encode('utf-8')
)

mensaje = {"usuario": "User_1", "contenido": "Hola Kafka"}
producer.send('tweets_topic', value=mensaje)
print(f"[OK] Enviado: {mensaje}")

# Paso 2: CONSUMIDOR - Leer datos
consumer = KafkaConsumer(
    'tweets_topic',
    bootstrap_servers='localhost:9092',
    auto_offset_reset='earliest',
    value_deserializer=lambda x: json.loads(x.decode('utf-8'))
)

for mensaje in consumer:
    print(f"[OK] Recibido: {mensaje.value}")
```

Cómo ejecutar el test:

```
# 1. Asegúrate de que Docker está corriendo
docker ps

# 2. Ejecuta el tester
python tests/tester.py
```

Salida esperada si TODO funciona:

```
>>> 1. INICIANDO PRODUCTOR (Enviando mensajes a localhost:9092)...
 [OK] Enviado: {'usuario': 'User_1', 'mensaje': 'Hola Kafka', 'contador': 1}
 [OK] Enviado: {'usuario': 'User_2', 'mensaje': 'Hola Kafka', 'contador': 2}
 ...
>>> 2. INICIANDO CONSUMIDOR (Leyendo de tweets_topic)...
 [OK] Recibido: {'usuario': 'User_1', 'mensaje': 'Hola Kafka', 'contador': 1}
 [OK] Recibido: {'usuario': 'User_2', 'mensaje': 'Hola Kafka', 'contador': 2}
```

```
 ...
>>> Consumidor finalizado con éxito. ¡TODO FUNCIONA!
```

¿Qué demuestra este test?

- OK Docker está levantado y Kafka escucha en `localhost:9092`.
- OK El topic `tweets_topic` existe y acepta mensajes.
- OK La infraestructura es capaz de desacoplar productor-consumidor (el patrón fundamental del streaming).
- OK Los datos fluyen correctamente desde Python hacia Kafka.

Este test fue crítico para identificar problemas de configuración antes de empezar el trabajo real de Spark.

Generación de Datos (Producer)

Responsable: Ismael González Loro

Tras establecer la infraestructura base (Docker y Kafka), el siguiente reto fue alimentar el sistema. En una arquitectura Big Data, el "pipeline" o tubería de procesamiento no sirve de nada si no tiene un flujo de entrada constante.

Mi misión como **Responsable de la Ingesta** fue crear la "realidad simulada": un generador de tráfico que emule el comportamiento de usuarios de una red social (tipo Twitter/X) en tiempo real, garantizando que el clúster de Spark siempre tenga datos que procesar.

2.1. Estrategia: ¿Por qué un simulador y no la API real?

Inicialmente se valoró conectar con la API oficial de Twitter (X). Sin embargo, se optó por desarrollar un **Generador Sintético (Synthetic Data Generator)** por tres razones técnicas y operativas:

1. **Independencia y Resiliencia:** Las APIs públicas tienen límites de tasa (*rate-limits*) y costes asociados. Un simulador nos permite generar tráfico infinito sin bloqueos, asegurando que la demostración en vivo no falle por causas externas.
2. **Control del Dato:** Para probar las capacidades de Spark Streaming (agrupaciones y ventanas de tiempo), necesitábamos asegurar que ciertos hashtags (ej: `#Spark`) aparecieran con más frecuencia que otros. Un generador sintético nos permite "trucar" las probabilidades para hacer la demo más visual.
3. **Velocidad Ajustable:** Podemos acelerar o frenar el flujo de datos modificando una simple variable de tiempo (`sleep`), algo imposible con datos reales.

2.2. Diseño Técnico del Productor

El componente desarrollado es un script en Python (`producer.py`) que implementa el patrón de diseño **Producer** de Kafka.

A. Librerías y Conexión Se utilizó la librería `kafka-python` por su robustez y simplicidad. El script se conecta al "Broker" expuesto por Docker en `localhost:9092`.

Una decisión clave fue la **Serialización**. Kafka transmite bytes puros, por lo que el productor se encarga de transformar nuestros objetos Python (diccionarios) a formato JSON y codificarlos en UTF-8 antes del envío.

```
# Configuración del Serializador en el Producer
producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda x: json.dumps(x).encode('utf-8') # Diccionario -> JSON Bytes
)
```

B. Estructura del Dato (Schema) Para facilitar el trabajo del **Consumidor (Persona C)**, diseñé un esquema de datos JSON limpio pero enriquecido. En lugar de enviar solo texto plano, enviamos una estructura estructurada:

- `usuario`: Simula quién escribe.
- `texto`: La frase completa (ej: "estoy aprendiendo mucho con #Spark").
- `hashtag_principal`: Aquí apliqué una lógica de pre-procesamiento. Aunque el hashtag está en el texto, lo envío también en un campo separado. Esto permite a Spark hacer el `groupBy` directamente sin necesidad de expresiones regulares complejas, optimizando el rendimiento.
- `timestamp`: Marca de tiempo para posibles análisis de latencia.

2.3. Implementación del Algoritmo

El núcleo del generador reside en un bucle infinito que construye mensajes aleatorios mediante listas predefinidas.

Fragmento destacado del código (`producer.py`):

```
def generador_tweets():
    # Listas de componentes aleatorios
    usuarios = ["@DataFan", "@SparkGuru", "@PythonDev"]
    hashtags = ["#Spark", "#BigData", "#RealTime", "#IA"]
    frases = ["increíble la velocidad de", "mañana examen de", "proyecto sobre"]

    # Selección aleatoria (Randomness)
    usuario = random.choice(usuarios)
    hashtag = random.choice(hashtags)
    frase = random.choice(frases)

    # Construcción del objeto JSON
    return {
        "usuario": usuario,
        "texto": f"{frase} {hashtag}",
        "hashtag_principal": hashtag, # Campo optimizado para Spark
        "timestamp": time.time()
    }

# Bucle de emisión
while True:
    tweet = generador_tweets()
    producer.send('tweets_topic', value=tweet)
    time.sleep(1) # Simulación de ritmo humano (1 tweet/seg)
```

2.4. Validación y Pruebas de Integración

Antes de entregar el testigo a la fase de procesamiento, verifiqué la correcta inyección de datos en el bus de mensajería.

Al no tener aún el consumidor de Spark listo, utilicé las herramientas nativas de Kafka dentro del contenedor Docker para "espiar" el topic `tweets_topic` y confirmar la llegada de los JSONs.

Comando de validación utilizado:

```
docker exec -it kafka kafka-console-consumer \
    --bootstrap-server localhost:9092 \
    --topic tweets_topic \
    --from-beginning
```

Conclusión de la Fase B: El sistema de ingesta es operativo y autónomo. Genera un flujo continuo de datos estructurados, desacoplando la generación del procesamiento y cumpliendo con los requisitos de la arquitectura Lambda propuesta.

Procesamiento con Spark (Consumer)

Responsable: Jairo Pabel Farfán Callau

El rol del consumidor es transformar el flujo bruto de mensajes de Kafka en información agregada sobre *trending topics* en tiempo (casi) real. Mientras que el productor simula una "red social" generando tweets con hashtags, el consumidor con Spark Structured Streaming se encarga de:

- Leer continuamente esos mensajes desde Kafka.
- Parsear el JSON para extraer el hashtag_principal.
- Contar apariciones por hashtag en ventanas de tiempo.
- Mostrar cada pocos segundos un ranking actualizado de los hashtags más usados.

Todo esto se implementa en el script src/spark_consumer.py, utilizando **PySpark** y el conector oficial **spark-sql-kafka-0-10**.

1. Objetivo del consumidor

El objetivo principal del consumidor es **calcular y mostrar los hashtags más utilizados en la última ventana de tiempo**, imitando el funcionamiento de un sistema de "trending topics":

- Se define una **ventana fija de 60 segundos** (un minuto "simulado" de red social).
- Cada **10 segundos** se recalculan los conteos de hashtags de ese minuto.
- La salida es una tabla ordenada de mayor a menor número de apariciones.

De esta forma se consigue un compromiso razonable entre:

- *Estabilidad* del ranking (se mantiene el histórico de un minuto completo).
- *Reactividad* (la tabla se actualiza cada 10 segundos).

2. Lectura del flujo desde Kafka

Para consumir los mensajes, se inicializa una sesión de Spark y se configura una **fuente de streaming Kafka** apuntando al mismo topic que usa el productor (tweets_topic):

```
raw_df = (
    spark.readStream
        .format("kafka")
        .option("kafka.bootstrap.servers", "localhost:9092")
        .option("subscribe", "tweets_topic")
        .option("startingOffsets", "latest")
        .load()
)
```

Puntos clave:

- startingOffsets = "latest" indica que el consumidor solo procesa **mensajes nuevos** a partir del momento en que se arranca, sin re-leer históricos anteriores.
- La columna value llega en binario, por lo que se castea a STRING para poder parsear el JSON enviado por el productor.

3. Modelo de datos y parseo del JSON

El productor envía mensajes JSON con esta estructura simplificada:

```
{
  "usuario": "User_X",
  "texto": "mensaje de ejemplo",
  "hashtag_principal": "#BigData",
  "timestamp": 1733940000.0
}
```

En el consumidor se define un **schema explícito** en PySpark y se realiza el parseo:

```
schema = StructType([
    StructField("usuario", StringType(), True),
    StructField("texto", StringType(), True),
    StructField("hashtag_principal", StringType(), True),
    StructField("timestamp", DoubleType(), True),
])


value_df = raw_df.selectExpr("CAST(value AS STRING) as json_str")


parsed_df = (
    value_df
    .select(from_json(col("json_str"), schema).alias("data"))
    .select("data.*")
)
```

Después se limpian posibles valores nulos y se añade una marca de tiempo de procesamiento que servirá para las ventanas temporales:

```
df_with_ts = (
    parsed_df
    .where(col("hashtag_principal").isNotNull())
    .withColumn("ts", current_timestamp())
)
```

4. Ventanas temporales y lógica de agregación

El núcleo del procesamiento consiste en agrupar los mensajes por hashtag dentro de ventanas temporales:

- **Ventana fija de 60 segundos**, alineada a minutos naturales (HH:MM:00,HH:MM:59HH:MM:00, HH:MM:59HH:MM:00,HH:MM:59).
- **Watermark de 60 segundos** para descartar datos demasiado atrasados y controlar el estado interno de Spark.
- Conteo de mensajes por cada hashtag_principal.

La definición de la ventana es:

```
windowed_counts = (
    df_with_ts
    .withWatermark("ts", "60 seconds")
    .groupBy(
        window(col("ts"), "60 seconds"),
        col("hashtag_principal")
    )
    .agg(count("*").alias("num_ocurrencias"))
)
```

A partir de ahí, se utiliza foreachBatch para procesar cada micro-batch de forma "estática", ordenar y limitar los resultados:

```
def process_batch(batch_df, batch_id):
    if batch_df.isEmpty():
        return


    # Ventana más reciente disponible en este batch
    max_end_row = batch_df.agg(max_("window.end").alias("max_end")).collect()[0]
    max_end = max_end_row["max_end"]


    if max_end is None:
        return


    latest_window_df = batch_df.filter(col("window.end") == max_end)


    top_hashtags = (
        latest_window_df
        .orderBy(col("num_ocurrencias").desc())
    )


    print("\n========================================")
    print(f"Batch {batch_id} - Snapshot en {datetime.now().strftime('%Y-%m-%d %H:%M:%S')}")
    print(f"Ventana del minuto [{max_end - timedelta(seconds=60)} , {max_end}]")
    print("Top hashtags en este minuto:")
    top_hashtags.show(truncate=False)
    print("========================================\n")
```

Por último, se arranca el streaming con un **trigger de 10 segundos**:

```
query = (
    windowed_counts
    .writeStream
    .outputMode("update")
    .foreachBatch(process_batch)
    .trigger(processingTime="10 seconds")
    .start()
)


query.awaitTermination()
```

Esto implica que cada ~10 segundos se recibe un nuevo lote (*batch*) con las actualizaciones y se imprime un nuevo snapshot del ranking.

5. Formato de salida e interpretación

En ejecución, el consumidor genera bloques de salida de este estilo:

```
========================================
Batch 12 - Snapshot en 2025-12-16 00:31:20
Ventana del minuto [2025-12-16 00:30:20 , 2025-12-16 00:31:20]
Top hashtags en este minuto:
+-------------------+-----------------+---------------+
|window             |hashtag_principal|num_ocurrencias|
+-------------------+-----------------+---------------+
|{...}              |#BigData         |42             |
|{...}              |#Spark           |35             |
|{...}              |#Kafka           |18             |
+-------------------+-----------------+---------------+
========================================
```

Interpretación:

- **Batch N**: número de actualización desde que se arrancó el consumidor.
- **Snapshot en ...**: momento en el que se ha generado ese resumen.
- **Ventana del minuto t0,t1t0, t1t0,t1**: intervalo de 60 segundos que se está analizando.
- La tabla muestra, para esa ventana de un minuto, cuántas veces ha aparecido cada hashtag (num_ocurrencias), ordenados de mayor a menor.

En términos conceptuales, esto equivale a un "trending topics del último minuto", recalculado y reimpreso de forma periódicacada 10 segundos.

Documentación

Responsable: Yahya El Baroudi El Ouazghari

Yahya El Baroudi El Ouazghari ha asumido el rol de Responsable de Documentación e Integración. Como responsable de la documentación, el objetivo principal ha sido traducir la complejidad técnica de la infraestructura (Docker) y el código (Python/Spark) en un entregable claro, reproducible y educativo. En un proyecto de Big Data, la documentación actúa como el puente que permite que el trabajo técnico sea comprendido y evaluado correctamente, cumpliendo con los criterios de evaluación que otorgan un gran peso a este apartado.

Para cumplir con los requisitos de la asignatura, se ha diseñado una estrategia de presentación híbrida y se ha elaborado tanto la guía de ejecución como el material de defensa.

1. **Estrategia Híbrida: Scripts vs. Notebooks** Uno de los desafíos principales ha consistido en adaptar una arquitectura de streaming real (que habitualmente funciona con scripts .py en servidores) al formato académico solicitado.

Según los requerimientos de la asignatura, se exigía un "Caso práctico comentado en detalle utilizando Google Colab o Jupyter". Sin embargo, ejecutar servicios de infraestructura como Kafka dentro de un entorno de celdas (Notebook) presenta problemas de bloqueo en los bucles de ejecución.

Para solucionar esta problemática, se ha estructurado el proyecto en dos niveles:

- **Nivel de Producción** (src/): Se mantiene el código limpio en archivos .py (producer.py y consumer.py), demostrando capacidad de desarrollo de software profesional.
- **Nivel de Presentación** (Notebook.ipynb): Se ha creado un "Notebook Maestro" que orquesta todo el proyecto. Este cuaderno no solo contiene código, sino que utiliza celdas Markdown para explicar paso a paso la teoría detrás de cada bloque, actuando como una memoria interactiva ejecutable.

2. **Implementación del Notebook Maestro** El Notebook entregable integra las tres partes del proyecto en un solo flujo visual. Se han implementado soluciones técnicas específicas para hacer esto posible:

- **Orquestación desde Jupyter**: Se utilizan comandos mágicos (!docker compose) para controlar la infraestructura desarrollada por Samuel directamente desde el navegador.
- **Ejecución en Segundo Plano (Background)**: Para el Generador de Datos de Ismael, se implementó el uso de la librería subprocess. Esto permite lanzar el productor en un hilo paralelo sin bloquear la celda de ejecución, posibilitando que el notebook continúe hacia la sección de Spark.
- **Visualización en Vivo**: Para el consumidor de Jairo, se configuró una

salida visual utilizando IPython.display y Pandas. En lugar de imprimir texto plano infinito en la consola, el notebook muestra una tabla HTML dinámica que se actualiza y refresca cada 5 segundos con los nuevos Trending Topics, ofreciendo una experiencia de usuario superior.

3. **Guía de Despliegue (Manual de Usuario)** Se ha elaborado el archivo README.md del repositorio, el cual sirve como manual de instrucciones para cualquier persona que desee replicar el proyecto. La guía se resume en tres pasos críticos para garantizar la reproducibilidad:

- **Prerrequisitos**: Instalación de Docker Desktop y Anaconda, así como la creación del entorno virtual arqesp para aislar las librerías pyspark y kafka-python.
- **Orden de Ejecución** (Start-up Sequence): Se documentó la importancia estricta del orden de encendido:
  - **1º Infraestructura** (Docker) -> Espera de 30s de "calentamiento".
  - **2º Productor** (Generar datos).
  - **3º Consumidor** (Procesar datos).
- **Gestión de Errores**: Se incluyó una sección de "Troubleshooting" en la documentación para resolver problemas comunes, tales como la falta de memoria en Docker o conflictos de puertos con servicios de Windows.

Gracias a esta documentación, el proyecto no es solo un código funcional en un entorno local específico, sino un sistema robusto capaz de ser desplegado y evaluado en cualquier máquina con Docker instalado.

4. **Elaboración del Material de Defensa (Presentación)** Para cumplir con el requerimiento de "Defensa pública mediante presentación" especificado en la guía docente, se ha diseñado y estructurado el material visual de apoyo.

Esta presentación ha sido elaborada con un enfoque ejecutivo, sintetizando las horas de trabajo técnico en una exposición clara. El documento de diapositivas incluye:

- **Diagramas de Arquitectura**: Simplificación visual de la comunicación entre Docker, Kafka y Spark.
- **Justificación Tecnológica**: Explicación comparativa de por qué se eligió Streaming frente a procesamiento Batch tradicional.
- **Evidencias de Funcionamiento**: Capturas de pantalla del sistema en operación para respaldar la demostración en vivo ante posibles fallos del directo ("Efecto Demo").

De esta forma, se entrega no solo la documentación técnica exhaustiva (Notebook y código fuente), sino también un soporte visual adecuado para la comunicación efectiva de los resultados.

Conclusiones

Este proyecto ha permitido simular con éxito un entorno de Big Data en tiempo real, integrando tecnologías punteras como **Docker**, **Kafka** y **Apache Spark**.

A diferencia de los enfoques tradicionales de procesamiento por lotes (Batch) vistos en clase (como Hive o Pig), esta arquitectura **Streaming** permite:

1. **Inmediatez:** Los datos se procesan conforme llegan, permitiendo reacciones al instante.
2. **Desacoplamiento:** Kafka actúa como un buffer robusto que separa la generación de datos de su consumo, evitando cuellos de botella.
3. **Escalabilidad:** El uso de contenedores Docker facilita el despliegue y la escalabilidad horizontal de los servicios.

En resumen, hemos logrado construir un pipeline completo de datos ("End-to-End") que no solo cumple con los requisitos académicos, sino que se acerca a las arquitecturas reales utilizadas en la industria para el análisis de redes sociales y eventos en vivo.