Universidad de Las Palmas de Gran Canaria

Escuela Ingeniería Informática

Grado de Ingeniería Informática - Curso 25/26

Práctica 2

Gorka Eymard Santana Cabrera

16-01-2026

Introducción

Esta práctica tiene como objetivo demostrar la construcción de un pipeline de datos en ***Amazon Web Service***, aplicando conceptos de ingeniería de datos en la nube: ingestión en streaming, transformación ETL, catalogación automática de metadatos y análisis interactivo mediante SQL. La motivación principal es recrear un escenario realista de procesamiento de datos a gran escala.

El proyecto implementa varios servicios AWS que trabajan de forma integrada para crear un Data Lake funcional. ***Amazon S3*** actúa como el repositorio centra de almacenamiento. Amazon Kinesis Data Streams funciona como el bus de eventos que captura datos en tiempo real con latencias de milisegundos. ***Amazon Kinesis Data Firehose*** simplifica la entrega de datos de Kinesis a S3, manejando el buffering automáticamente el buffering, la compresión y la posibilidad de invocar transformaciones mediante ***AWS Lambda***, funciones serverless que ejecutan código sin necesidad servidores. Para el procesamiento y realización de transformaciones complejas, utilizamos ***AWS Glue*** un servicio ETL completamente administrado que ejecuta trabajos Apache Spark distribuidos y mantiene un catálogo de metadatos que describe la estructura de nuestros datos. Finalmente usamos ***Amazon Athena*** para realizar consultas SQL directamente sobre los archivos en S3, sin necesidad de cargar datos en una base de datos tradicional, pagando únicamente por los datos escaneados en cada query.

Desarrollo

Configuración del bucket S3

El primer paso de la práctica consiste en configurar el Bucket S3 de forma que actúa como el Data Lake central del proyecto en el que se almacenan tanto datos crudos como procesados.

El bucket se crea asignándole un nombre único basado en el AccountID de AWS evitando colisiones globales.

```
aws s3api put-object --bucket $BUCKET_NAME --key raw/
aws s3api put-object --bucket $BUCKET_NAME --key raw/f1_driver_standings/
aws s3api put-object --bucket $BUCKET_NAME --key processed/
aws s3api put-object --bucket $BUCKET_NAME --key
processed/driver_standings_by_race/
aws s3api put-object --bucket $BUCKET_NAME --key
processed/driver_standings_by_driver/
aws s3api put-object --bucket $BUCKET_NAME --key config/
aws s3api put-object --bucket $BUCKET_NAME --key scripts/
aws s3api put-object --bucket $BUCKET_NAME --key queries/
aws s3api put-object --bucket $BUCKET_NAME --key errors/
aws s3api put-object --bucket $BUCKET_NAME --key logs/
```

Fragmento Código 1: referencia código script

Esto asigna el nombre '`datalake-f1-driver-standings-${Account_ID}`' de forma que para cada cuenta de AWS tiene un nombre único, dado que los nombres de buckets en S3 son globalmente únicos.

La estructura del bucket establece una jerarquía de carpetas que sigue las buenas prácticas de Data Lake:

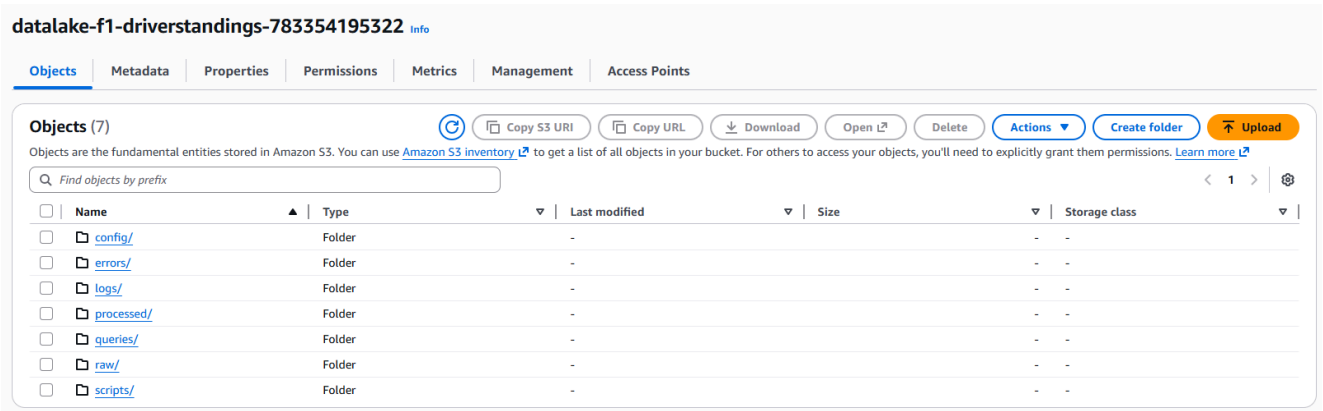

Figura 1: captura estructura S3

Cada directorio tiene un propósito distinto:

- raw/: Zona en la que se almacenan los datos crudos insertados directamente por parte del Firehose. Estos se encuentran en formato JSON y particionados por fecha.
- processed/: en esta zona se encuentran los datos ya transformados y optimizados por los Glue Jobs. Usando el formato Parquet conn compresión Snappy para consultas eficientes.
- scripts/: almacenamiento de scripts ETL de PySpark que ejecutan los Glue Jobs.
- queries/: resultados de las consultas ejecutadas en Amazon Athena.
- errors/: registros que fallaron durante el procesamiento de Firehose.
- logs/: logs de ejecución de Spark generados por los Glue Jobs.
- config/: en esta zona se guardan archivos de configuración, como por ejemplo YML de lambda en caso de usarse.

Implementación del productor de datos

El productor de datos es el punto de entrada del pipeline, encargándose de la ingestión inicial de información al sistema. Este componente lee los registros históricos de driver standings desde un archivo CSV local y los transmite de forma controlada hacia el Kinesis Data Stream mediante batches de registros. Su función es convertir datos estáticos en un flujo continuo de eventos, simulando la llegada de información en tiempo real. El productor implementa características como batching, control de throughput mediante delays, y manejo de errores para garantizar una ingesta fiable.

Creación del Kinesis Stream

En el script de lanzamiento automático, se crea el stream con 1 shard:

```
aws kinesis create-stream --stream-name f1-driver-standings-stream
--shard-count 1
```

Fragmento Código 2: Comando creación stream A[1]

La decisión de utilizar 1 shard está directamente relacionada con el volumen de datos del proyecto. Con nuestro dataset de aproximadamente 35,000 registros de clasificaciones, cada uno con un tamaño aproximado de 200-300 bytes en formato JSON, estamos hablando de unos 6-9 MB de datos totales. Incluso enviando todos los datos en el lapso de pocos minutos, estamos muy por debajo del límite de 1 MB/segundo por shard. Usar múltiples shards aumentaría innecesariamente los costos (cada shard cuesta aproximadamente $0.015/hora o ~$11/mes en us-east-1) sin aportar beneficio alguno para este volumen de datos.

Implementación del Producer

El producer está implementado en kinesis.py[anexo 2] este representa una aplicación que genera eventos hacia Kinesis. Este tiene varias funciones:

```
def load_csv_data(file_path):
    data = []
    with open(file_path, 'r', encoding='utf-8') as f:
        reader = csv.DictReader(f)
        for row in reader:
            data.append(row)
    return data
```

Fragmento Código 3: función carga de csv A[4]

Esta primera función se encarga de cargar por completo el archivo CSV en memoria, puesto que tiene un tamaño manejable ~2Mb. Esto nos da la posibilidad de simplificar el código entre lectura y envío.

```
BATCH_SIZE = 100
BATCH_DELAY = 2.1

for i in range(0, total_records, BATCH_SIZE):
    batch = data[i:i+BATCH_SIZE]
    records = []

    for registro in batch:
        # ... preparación de payload
```

Fragmento Código 4: estrategia Batching A[4]

La configuración de batching es uno de los aspectos críticos para la eficiencia del producer. Kinesis permite enviar hasta 500 registros en una sola llamada a '`PutRecords`', pero elegimos un batch size de 100 registros por varias razones:

1. Por tener cierto margen de error dado que el límite de 500 incluye tanto registros exitosos como fallidos, por ello 100 nos permite reintentos.
2. Mayor manejo de fallos, es decir, si un batch falla solo perderíamos 100 registros que necesitan reintento.

Por otro lado se elige un delay de 2.1 segundos para simular un streaming de datos más realista y a su vez evitar el burst throttling que AWS aplica en picos instantáneos.

```
payload = {
    'driverStandingsId': to_int(registro.get('driverStandingsId')),
    'raceId': to_int(registro.get('raceId')),
    'driverId': to_int(registro.get('driverId')),
    'points': to_int(registro.get('points')),
    'position': to_int(registro.get('position')),
    'raceId': to_int(registro.get('raceId')),
    'driverId': to_int(registro.get('driverId')),
    'points': to_int(registro.get('points')),
    'position': to_int(registro.get('position')),
    'positionText': registro.get('positionText'),
```

```
    'wins': to_int(registro.get('wins')),
    'forename': registro.get('forename'),
    'surname': registro.get('surname'),
    'dob': registro.get('dob'),
    'nationality': registro.get('nationality')
}
```

Fragmento Código 5: definición de la estructura del payload A[4]

La estructura del payload implementa una normalización de datos que es fundamental para el procesamiento downstream. Los CSVs, almacenan todo los valores como strings, por ello la función auxiliar to_int() convierte de forma segura estos strings numéricos a enteros reales.

```
def to_int(value):
    try:
        return int(value) if value else None
    except (ValueError, TypeError):
        return None
```

Fragmento Código 6: función de conversión A[4]

Esta conversión da lugar a muchas ventajas, entre ellas reducir el tamaño del payload, y la ventaja más importante es que facilita el procesamiento downstream en glue donde las operaciones agregadas como avg() o sum() esperan tipos numéricos.

```
records.append({
    'Data': json.dumps(payload),
    'PartitionKey': str(registro['driverId'])
})
```

Fragmento Código 7: carga de registros a Kinesis A[4]

El formato de cara registro enviado a Kinesis sigue una estructura dada por la API PutRecords. El campo Data es un string que representa el payload completo del registro. Usamos json.dumps(payload) para serializar nuestro diccionario Python a un string JSON compacto.

El partitionKey es el mecanismo de Kinesis para distribuir registros entre shards y garantiza ordenamiento. Cuando usamos str(registro[‘driverId’]) como partition key, estamos garantizando que todos los registros del mismo piloto irán al mismo shard, y por tanto se procesarán en ese orden.

```
response = kinesis.put_records(
    StreamName=STREAM_NAME,
    Records=records
)
```

```
failed_count = response['FailedRecordCount']
if failed_count > 0:
    logger.warning(f"Batch {i//BATCH_SIZE + 1}: {failed_count} registros
fallaron")
    for idx, record in enumerate(response['Records']):
        if 'ErrorCode' in record:
            logger.error(f"  Registro {idx}: {record['ErrorCode']} -
{record['ErrorMessage']}")
```

Fragmento Código 8: Manejo de errores de Kinesis A[4]

Dado que la API PutRecords de Kinesis retorna éxito pese a haber ocurrido algún fallo con algún registro del batch, nuestro código loguea errores para que sean tratados con posterioridad.

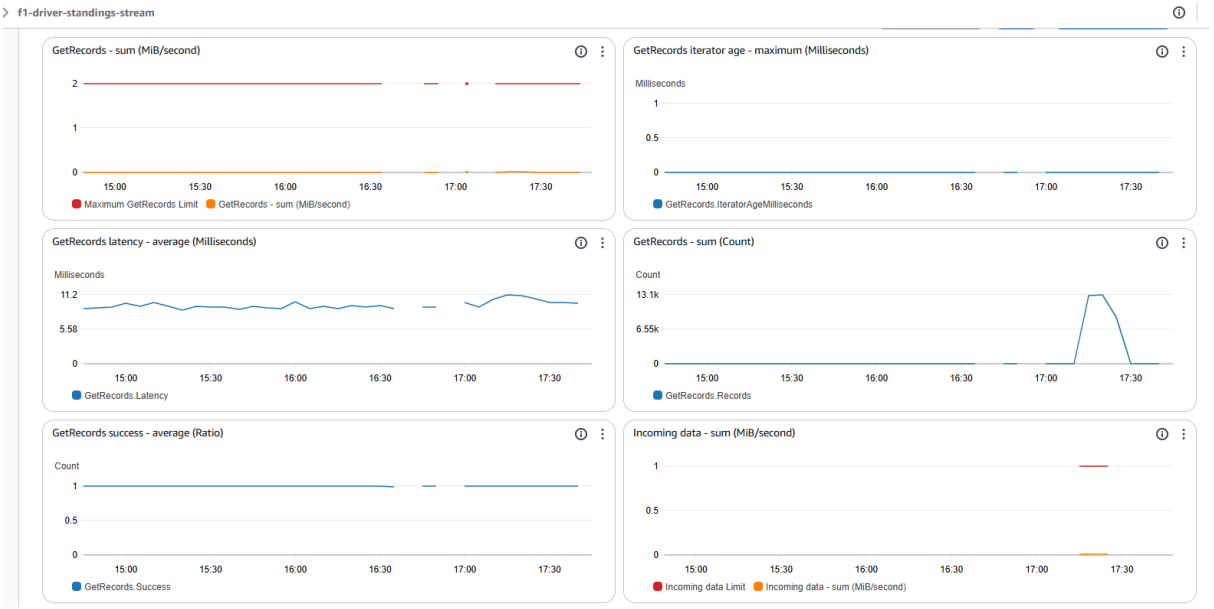

Figura 2: captura Monitorización Kinesis Data Streams

Como se puede apreciar en la imagen los registros se están subiendo correctamente al Kinesis

Configuración del consumidor (Kinesis Firehose)

Amazon Kinesis Data Firehose es un servicio de delivery completamente administrado que simplifica la entrega de datos streaming a S3 y otros destinos. A diferencia de Kinesis Data Streams donde el consumidor debe gestionar checkpoints, resharding y lógica de entrega, Firehose abstrae toda esta complejidad ofreciendo una experiencia "configurar y olvidar". El servicio maneja automáticamente el buffering, compresión, transformación opcional mediante Lambda, y reintentos para registros problemáticos.

La arquitectura de Firehose se basa en micro-batching: acumula registros del stream durante un período o hasta alcanzar cierto tamaño, escribiéndolos a S3 como archivo

único. Este enfoque optimiza costos al reducir operaciones PUT en S3 y mejora la eficiencia de queries posteriores, ya que leer pocos archivos grandes es significativamente más rápido que procesar miles de archivos pequeños debido al overhead de metadatos y red.

Configuración del Firehose

```
$firehoseConfig = @{
    BucketARN = "arn:aws:s3:::$BUCKET_NAME"
    RoleARN = $ROLE_ARN
    Prefix =
"raw/f1_driver_standings/partition_date=!{partitionKeyFromLambda:partitio
n_date}/"
    ErrorOutputPrefix = "errors/!{firehose:error-output-type}/"
    CompressionFormat = "UNCOMPRESSED"
    DynamicPartitioningConfiguration = @{
        Enabled = $true
        RetryOptions = @{ DurationInSeconds = 300 }
    }
}
```

Fragmento Código 9: Definición del Firehose A[1]

Como se puede apreciar en el código adjunto en el código se tiene habilitado el particionado dinámico permitiendo crear prefijos con metadata añadido por una lambda. En este caso concreto se indica que se extraiga el valor de partition_date del objeto metada.partitionKeys.

Por ejemplo, si la Lambda devuelve metadata.partitionKeys.partition_date = "2026-01-16", Firehose escribirá el archivo en: s3://bucket/raw/f1_driver_standings/partition_date=2026-01-16/delivery-stream-name-yyyy-mm-dd-hh-random.json

```
BufferingHints = @{ SizeInMBs = 64; IntervalInSeconds = 60 }
```

Fragmento Código 10: configuración de Buffering A[1]

La configuración de buffering define cuándo Firehose agrupará registros acumulados y los escribirá como un archivo en S3. Opera con lógica de "OR": el delivery ocurre cuando se cumple cualquiera de las dos condiciones:

1. SizeInMBs = 64: El buffer acumulado alcanza 64 MB de datos
2. IntervalInSeconds = 60: Han pasado 60 segundos desde el último delivery

Esta configuración representa un trade-off cuidadosamente considerado entre latencia, eficiencia de costo, y tamaño de archivo óptimo. Con nuestro throughput de ~100 registros cada 2.1 segundos (desde el producer), cada registro de ~300 bytes, estamos generando aproximadamente 14 KB/segundo. Esto significa que tardaríamos más de 1 hora en llenar el buffer de 64 MB. Por tanto, en nuestro caso, el criterio de tiempo (60 segundos) siempre se cumple primero, resultando en archivos pequeños de aproximadamente 840 KB cada uno (14 KB/s × 60s).

```
ProcessingConfiguration = @{
    Enabled = $true
```

```
    Processors = @(
        @{
            Type = "Lambda"
            Parameters = @(
                @{
                    ParameterName = "LambdaArn"
                    ParameterValue = $LAMBDA_ARN
                }
            )
        }
    )
}
```

Fragmento Código 11: configuración de procesamiento de datos A[1]

Integrar la lambda nos permite transformaciones en tiempo real antes de persistir en S3, centralizando la lógica de limpieza en lugar de duplicarla en múltiples consumidores. Firehose invoca Lambda asíncronamente con batches de hasta 3 MB o 500 registros. Lambda debe responder con registros transformados y metada de estado. Tras 3 intentos fallidos, registros problemáticos van al bucket de errores.

Configuración de AWS Lambda

```
def lambda_handler(event, context):
    output = []
    partition_date = datetime.utcnow().strftime('%Y-%m-%d')
```

Fragmento código 12: función lambda A[5]

Usamos datetime.utcnow() en lugar de hora local garantizando consistencia global. UTC elimina ambigüedades cuando datos se generan desde múltiples regiones o zonas horarias. El formato garantiza ordenamiento lexicográfico correcto: las particioens se ordenan cronológicamente automáticamente.

```
for record in event['records']:
    payload = base64.b64decode(record['data']).decode('utf-8')
    data_dict = json.loads(payload)
    data_without_redundant = {k: v for k, v in data_dict.items() if k not
in ['positionText']}
```

Fragmento código 13: procesamiento registros A[5]

El proces implementa 3 operaciones:

1. Decodificación: Firehose codifica los datos transmitidos.
2. Parsin JSON: convierte el string a diccionario Python
3. Limpieza: eliminamos positionText porque es redundante.

```
output_record = {
    'recordId': record['recordId'],
    'result': 'Ok',
    'data': base64.b64encode((data_json +
'\n').encode('utf-8')).decode('utf-8'),
    'metadata': {
```

```
        'partitionKeys': {
            'partition_date': partition_date
        }
    }
}
```

Fragmento Código 14: definición de registro de salida de la lambda A[5]

El formato sigue el protocolo de Firehose: recordId debe ser idéntico al input para correlación. El campo data requiere añadir salto de línea para formato JSON Lines cada línea es un objeto independiente, y decodificar la string ASCII para serializarla.

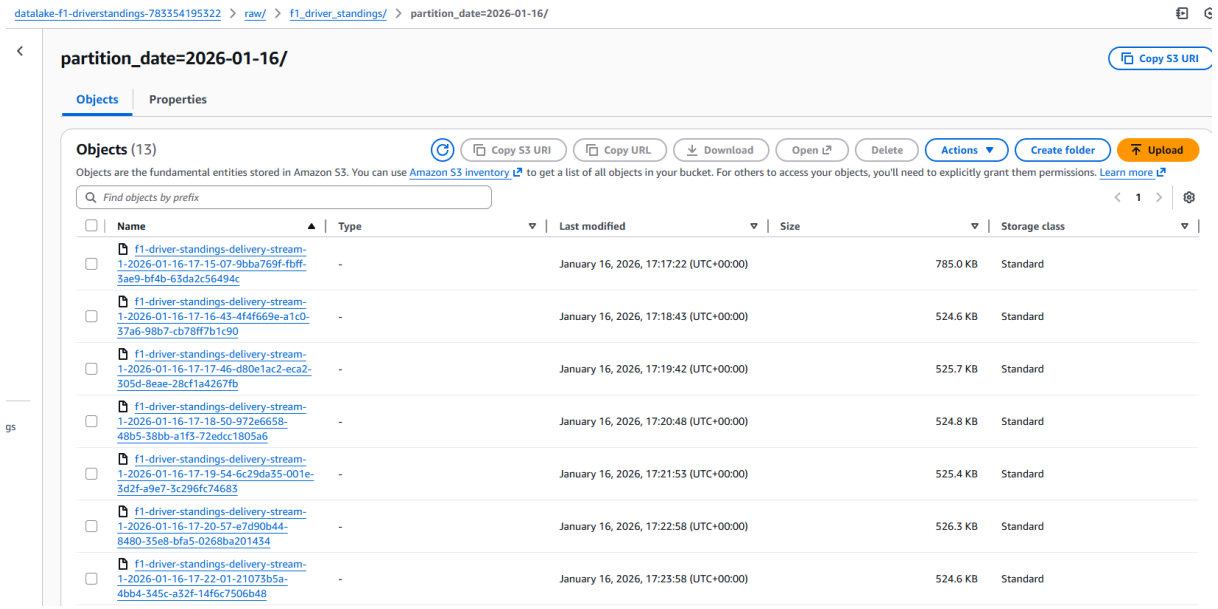

Figura 3: registros almacenados en la zona raw/

En la figura se pued ver como los registros tratados por la lambda han sido almacenados la partición 2026-01-16.

Configuración de AWS Glue

AWS Glue es un servicio ETL serverless que simplifica la preparación de datos para analytics. Comprende dos componentes: el **Glue Data Catalog** (repositorio de metadatos compatible con Hive que almacena esquemas, particiones y estadísticas) y **Glue Jobs** (trabajos PySpark ejecutados en clústeres efímeros administrados por AWS). Esto permite escribir código ETL que referencia tablas abstractas del catálogo en lugar de paths S3 hardcodeados.

Database y Crawlers

```
$dbInput = @{ Name = "f1_db" } | ConvertTo-Json
aws glue create-database --database-input file://glue_db_input.json
```

Fragmento Código 15: definición del nombre del Catalog A[1]

El Glue Database es un namespace lógico que agrupa tablas relacionadas. Su propósito es organizacional.

```
$crawlerTargets = @{ S3Targets = @( @{ Path =
"s3://$BUCKET_NAME/raw/f1_driver_standings" } ) }
aws glue create-crawler --name f1-driver-standings-raw-crawler --role
$ROLE_ARN --database-name f1_db --targets file://crawler_targets.json
```

Fragmento Código 16: crawler raw A[1]

Los Glue Crawlers escanean data stores, infieren esquemas mediante muestreo detectando particiones, y crean/actualizan tablas en Data Catalog. Para archivos JSON, parsean campos únicos, infieren tipos y detectan particiones analizando prefijos S3.

```
$crawlerTargets = @{
    S3Targets = @(
        @{ Path = "s3://$BUCKET_NAME/processed/driver_standings_by_race"
}
        @{ Path =
"s3://$BUCKET_NAME/processed/driver_standings_by_driver" }
    )
}
```

Fragmento Código 17: crawler processed A[1]

Cataloga las tablas Parquet procesadas, detectando particiones automáticamente

ETL Jobs

```
dynamic_frame =
glueContext.create_dynamic_frame.from_catalog(database=database,
table_name=table)
df = dynamic_frame.toDF()
```

Fragmento Código 18: lectura de tabla catalogada A[6] A[7]

Ambos jobs leen la tabla catalogada sin tener que especificar paths S3.

```
$commandConfig = @{
    Name = "glueetl"
    ScriptLocation =
"s3://$BUCKET_NAME/scripts/driver_standing_aggregation_by_driver.py"
    PythonVersion = "3"
}
$defaultArgs = @{
    "--database" = $DATABASE
    "--table" = $TABLE
    "--output_path" =
"s3://$BUCKET_NAME/processed/driver_standings_by_driver/"
    "--enable-continuous-cloudwatch-log" = "true"
}
```

Fragmento Código 19: configuración de los Jobs A[6] A[7]

Como se puede observar en la definición de los jobs se puede ver que apunta al catálogo Glue, define output path, habilita logging CloudWatch. Workers: 2 nodos G.1X (0.5 DPU c/u).(Para el caso del Job de ETL que agrupa por carrera tiene la misma configuración pero distinto script y distinto target)

1. **Agregación por Piloto**

```
driver_df = df.groupBy("driverId") \
    .agg(
        spark_min("position").alias("mejor_posicion_historica"),
        avg("position").alias("posicion_promedio_historica"),
        stddev("position").alias("desviacion_estandar_posicion"),
        count("raceId").alias("total_carreras_participadas")
    ) \
    .orderBy("driverId")
```

Fragmento Código 20: agregación por pilotos A[6]

Calcula métricas: mejor posición histórica, posición promedio, desviación estándar y total de carreras participadas.

2. **Agregación por Race**

```
# Agregación por Carrera
race_df = df.groupBy("raceId") \
    .agg(
        spark_min("position").alias("mejor_posicion"),
        avg("position").alias("posicion_promedio"),
        stddev("position").alias("desviacion_estandar_posicion"),
        count("driverId").alias("total_pilotos_participantes")
    ) \
    .orderBy("raceId")
```

Fragmento Código 21: agregación por carrera A[7]

Calcula métricas por carrera: mejor posición (generalmente 1), posición promedio de todos los pilotos, desviación estándar de las posiciones y total de pilotos que participaron en esa carrera.

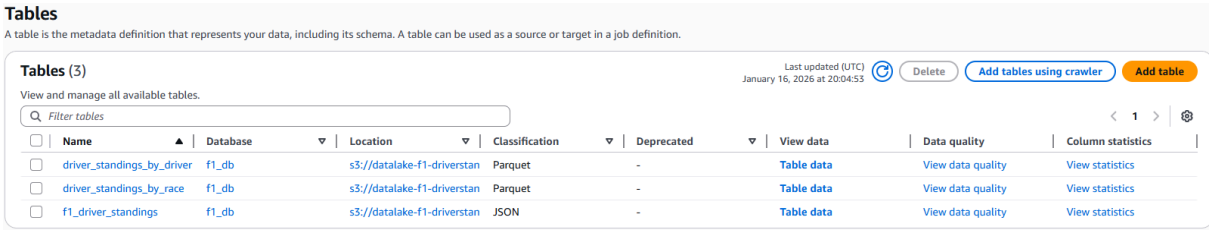

Figura 5: tablas generadas tras la ejecución de los Jobs y los crawlers

Amazon Athena

Amazon Athena se configura automáticamente al crear las tablas mediante los Glue Crawlers, que escanean los datos en S3 y generan el esquema en el Data Catalog. Las consultas SQL se ejecutan usando el AWS CLI mediante el comando aws athena start-query-execution, especificando la base de datos (f1_db), la query SQL y la ubicación de resultados en S3.

```
$query4 = "SELECT driverId, forename, surname, total_points,
posicion_promedio_historica, num_races FROM
f1_db.driver_standings_by_driver ORDER BY total_points DESC LIMIT 10"
```

Fragmento código 22: definición de una queryA[3]

Define una consulta SQL que selecciona los 10 pilotos con más puntos totales de la tabla agregada driver_standings_by_driver. La query se almacena en la variable `$query4` y posteriormente se ejecuta con `Execute-AthenaQuery`, especificando la base de datos f1_db y ordenando por puntos en orden descendente.

Diagrama del flujo de datos

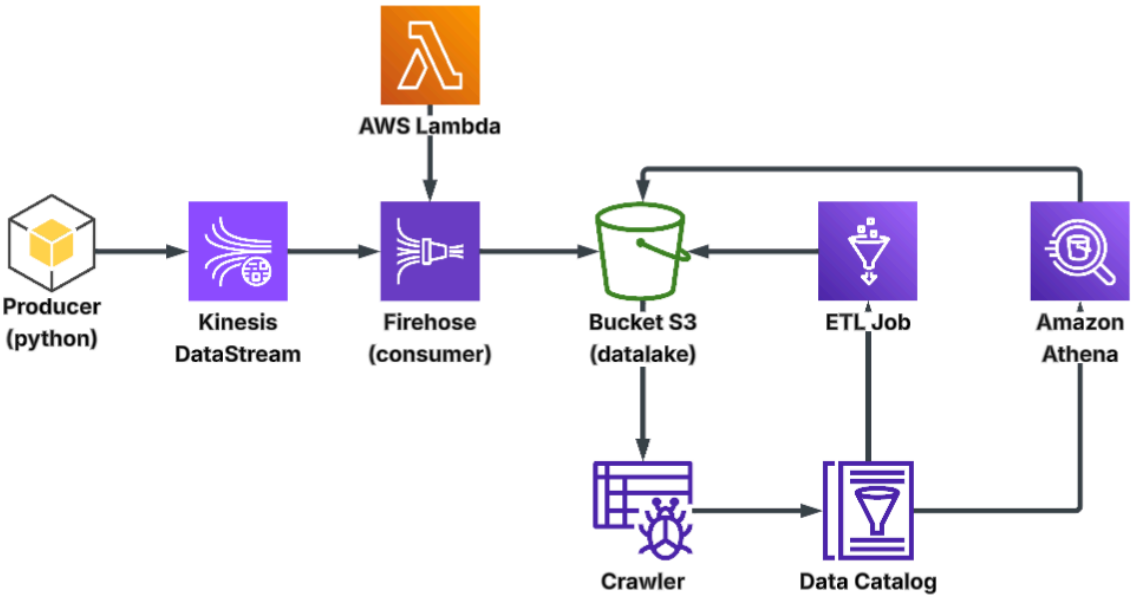

Figura 5: diagrama de Flujo de datos

El diagrama sigue el flujo de datos que sigue la implementación realizada, donde:

1. **Producer**(python): lee y envía datos a **Kinesis DataStream**, donde son almacenados durante 24 horas en 1 shard usando de clave de partición el driverId
2. **Firehose:** este es el encargado de consumir los datos del Kinesis y transformarlos usando una **Lambda** para almacenarlos posteriormente en el **Bucket S3**, que actúa como data lake. Los datos procesados por la lambda van al directorio /raw dentro del Bucket.
3. **Crawler:** este elemento se encarga de crear tablas en el Data Catalog de glue a partir de los datos almacenados en el Bucket S3. En el caso de la implementación realizada se definen y ejecutan 2 crawlers:
    a. **(raw):** escanea la zona raw/ del S3 y registra la tabla f1_driver_standings en el **Data Catalog** de Glue.

   b. **(processed)**: este crawler escanea la zona processed/ del S3 y registra las tablas **driver_standings_by_race** y **driver_standings_by_driver** en el **Data Catalog**.

4. **ETL Jobs:** se encargan de procesar los datos raw usando Spark con 2 workers G.1X:
   a. **driver-standings-by-race**: agrupa por raceId calculado estadísticas (avg_postion, total_drivers, top_driver). Generando particiones Parquet en la zona processed/by_race/ del S3.
   b. **driver-standings-by-driver**: agrupa por driverId calculando estadísticas. Generando particiones Parquet en la zona processed/by_driver del S3.

Presupuesto y estimación de costes

Descripción del escenario

El escenario propuesto para este presupuesto sería el estudio de análisis de datos de F1. Vamos a suponer que vamos a estudiar los 34863 datos históricos. Y vamos a suponer que se ejecuta el análisis 2 veces mensual

Servicio	Uso	Coste Unitario	Total Mensual
Amazon Kinesis Data Streams	58-70 records/sec, 1 día retención	0.086$/registro 0.13 $/GB	13.08
Amazon Data firehose	1.59 records/min, 5 KB por record	0.029/GB	0.02
AWS Lambda	50 invocaciones/mes, 128 MB, 50 ms	0.20/1M req	free
AWS Glue Data Catalog	1,131 objetos, ~70 requests	1/100K objetos	0.01
AWS ETL Jobs	2 DPU, 7.14 min/mes	0.44/DPU-hour	0.12
AWS Glue Crawler	2 crawlers, 10 min mínimo	0.44/DPU-hour	0.15
AWS S3	0.0184 GB, 3K PUT, 4K GET	0.023/GB + requests	0.02
Amazon Athena	10 queries/mes, 0.02 GB escaneados	5/TB	free
Total			13.4 $

Tabla 1: Costos

A este presupuesto se le suma un 15% para posibles costos extras por fallas imprevistas.

Conclusiones

Este proyecto implementa un Data Lake completo en AWS para el análisis de datos históricos de Fórmula 1, procesando 34,863 registros mediante una arquitectura serverless que combina ingesta en tiempo real (Kinesis), transformación con Lambda, almacenamiento optimizado en S3, procesamiento ETL con Glue y análisis SQL con Athena. La solución permite obtener estadísticas agregadas por piloto y por carrera almacenadas en formato Parquet particionado, lo que optimiza las consultas analíticas. El script automatizado completo ejecuta en aproximadamente 17-20 minutos, desde la ingesta inicial hasta la disponibilidad de las tablas finales para consulta.

Desde el punto de vista económico, el análisis de costos revela que Kinesis Data Streams es el elemento más caro de la arquitectura.

Posibles Avances Futuros

Se podría añadir el registro de tiempos de vuelta para poder realizar análisis e ir llevando un histórico de todos los datos obtenidos en todas las temporadas de Fórmula 1 para estudiar el avance de los pilotos y los circuitos.

Referencias y bibliografía

Rao, R. (2023). *Formula 1 World Championship (1950-2023). Kaggle Datasets*. https://www.kaggle.com/datasets/rohanrao/formula-1

Amazon Web Services. (2026). *AWS Glue Developer Guide. Amazon Web Services* https://docs.aws.amazon.com/glue/

Amazon Web Services. (2026). *Amazon Kinesis Data Streams Developer Guide. Amazon* https://docs.aws.amazon.com/kinesis/

Amazon Web Services. (2026). *Amazon Athena User Guide. Amazon Web Services* https://docs.aws.amazon.com/athena/

Amazon Web Services. (2026). *AWS Lake Formation - Best Practices for Data Lakes*. https://aws.amazon.com/lake-formation/

Anexos

El repositorio: https://github.com/gorkaftv1/CN_P2_F1 contiene los ficheros adjuntos a continuación.

A[1]

```
# Parametros del script
param(
    [Parameter(Mandatory=$false)]
    [ValidateSet("race", "driver", "both")]
    [string]$JobToRun = "both"
)

$WAIT_AFTER_COMPLETION = 150
$MAX_WAIT_TIMEOUT = 600
$ScriptDir = Split-Path -Parent $MyInvocation.MyCommand.Path
$ProjectRoot = Split-Path -Parent $ScriptDir
$SrcDir = Join-Path $ProjectRoot "src"
$ScriptsDir = Join-Path $ProjectRoot "scripts"

$AWS_REGION = "us-east-1"
$ACCOUNT_ID = aws sts get-caller-identity --query Account
--output text
$BUCKET_NAME = "datalake-f1-driverstandings-$ACCOUNT_ID"
$ROLE_ARN = aws iam get-role --role-name LabRole --query
'Role.Arn' --output text

Write-Host "`n========================================="
-ForegroundColor Cyan
Write-Host "F1 Driver Standings - Timed Setup Script"
-ForegroundColor Cyan
Write-Host "========================================="
-ForegroundColor Cyan
Write-Host "Bucket: $BUCKET_NAME" -ForegroundColor Yellow
Write-Host "Role: $ROLE_ARN" -ForegroundColor Yellow
Write-Host "Project Root: $ProjectRoot" -ForegroundColor Yellow
Write-Host "Job a ejecutar: $JobToRun" -ForegroundColor Yellow
Write-Host "=========================================`n"
-ForegroundColor Cyan

# S3
Write-Host "[1/11] Creando S3 Bucket y carpetas..."
-ForegroundColor Green
aws s3 mb s3://$BUCKET_NAME

aws s3api put-object --bucket $BUCKET_NAME --key raw/
aws s3api put-object --bucket $BUCKET_NAME --key
raw/f1_driver_standings/
aws s3api put-object --bucket $BUCKET_NAME --key processed/
aws s3api put-object --bucket $BUCKET_NAME --key
processed/driver_standings_by_race/
aws s3api put-object --bucket $BUCKET_NAME --key
processed/driver_standings_by_driver/
aws s3api put-object --bucket $BUCKET_NAME --key config/
```

```
aws s3api put-object --bucket $BUCKET_NAME --key scripts/
aws s3api put-object --bucket $BUCKET_NAME --key queries/
aws s3api put-object --bucket $BUCKET_NAME --key errors/
aws s3api put-object --bucket $BUCKET_NAME --key logs/

Write-Host "[2/11] Creando Kinesis Stream..." -ForegroundColor
Green
aws kinesis create-stream --stream-name
f1-driver-standings-stream --shard-count 1

Write-Host "Esperando a que Kinesis Stream este ACTIVE..."
-ForegroundColor Yellow
$waited = 0
while ($waited -lt $MAX_WAIT_TIMEOUT) {
    $streamStatus = aws kinesis describe-stream --stream-name
f1-driver-standings-stream --query
'StreamDescription.StreamStatus' --output text
    if ($streamStatus -eq "ACTIVE") {
        Write-Host "Kinesis Stream ACTIVE" -ForegroundColor
Green
        break
    }
    Start-Sleep -Seconds 5
    $waited += 5
}
Start-Sleep -Seconds $WAIT_AFTER_COMPLETION

Write-Host "[3/11] Creando Lambda Function..." -ForegroundColor
Green
Set-Location (Join-Path $SrcDir "lambda")
if (Test-Path firehose_driver_standings.zip) { Remove-Item
firehose_driver_standings.zip }
Compress-Archive -Path firehose_driver_standings.py
-DestinationPath firehose_driver_standings.zip -Force

aws lambda create-function `
    --function-name f1-firehose-lambda `
    --runtime python3.10 `
    --role $ROLE_ARN `
    --handler firehose_driver_standings.lambda_handler `
    --zip-file fileb://firehose_driver_standings.zip `
    --timeout 60

Remove-Item firehose_driver_standings.zip

$LAMBDA_ARN = aws lambda get-function --function-name
f1-firehose-lambda --query 'Configuration.FunctionArn' --output
text
Write-Host "Lambda ARN: $LAMBDA_ARN" -ForegroundColor Yellow

Set-Location $ScriptsDir
Write-Host "[4/11] Creando Firehose Delivery Stream..."
-ForegroundColor Green

$firehoseConfig = @{
```

```
    BucketARN = "arn:aws:s3:::$BUCKET_NAME"
    RoleARN = $ROLE_ARN
    Prefix =
"raw/f1_driver_standings/partition_date=!{partitionKeyFromLambd
a:partition_date}/"
    ErrorOutputPrefix = "errors/!{firehose:error-output-type}/"
    BufferingHints = @{ SizeInMBs = 64; IntervalInSeconds = 60
}
    CompressionFormat = "UNCOMPRESSED"
    DynamicPartitioningConfiguration = @{
        Enabled = $true
        RetryOptions = @{
            DurationInSeconds = 300
        }
    }
    ProcessingConfiguration = @{
        Enabled = $true
        Processors = @(
            @{
                Type = "Lambda"
                Parameters = @(
                    @{
                        ParameterName = "LambdaArn"
                        ParameterValue = $LAMBDA_ARN
                    }
                )
            }
        )
    }
}

$firehoseConfig | ConvertTo-Json -Depth 10 | Set-Content
firehose_config.json

aws firehose create-delivery-stream `
    --delivery-stream-name f1-driver-standings-delivery-stream
`
    --delivery-stream-type KinesisStreamAsSource `
    --kinesis-stream-source-configuration
"KinesisStreamARN=arn:aws:kinesis:${AWS_REGION}:${ACCOUNT_ID}:s
tream/f1-driver-standings-stream,RoleARN=$ROLE_ARN" `
    --extended-s3-destination-configuration
file://firehose_config.json

Remove-Item firehose_config.json

Write-Host "Esperando a que Firehose este ACTIVE..."
-ForegroundColor Yellow
$waited = 0
while ($waited -lt $MAX_WAIT_TIMEOUT) {
    $firehoseStatus = aws firehose describe-delivery-stream
--delivery-stream-name f1-driver-standings-delivery-stream
--query 'DeliveryStreamDescription.DeliveryStreamStatus'
--output text 2>$null
    if ($firehoseStatus -eq "ACTIVE") {
```

```
        Write-Host "Firehose ACTIVE" -ForegroundColor Green
        break
    }
    Start-Sleep -Seconds 5
    $waited += 5
}
Start-Sleep -Seconds $WAIT_AFTER_COMPLETION

Write-Host "[5/11] Creando Glue Database..." -ForegroundColor
Green
$dbInput = @{ Name = "f1_db" } | ConvertTo-Json
$dbInput | Set-Content glue_db_input.json
aws glue create-database --database-input
file://glue_db_input.json
Remove-Item glue_db_input.json
Write-Host "[6/11] Creando Glue Crawler..." -ForegroundColor
Green

$crawlerTargets = @{ S3Targets = @( @{ Path =
"s3://$BUCKET_NAME/raw/f1_driver_standings" } ) }
$crawlerTargets | ConvertTo-Json -Depth 10 | Set-Content
crawler_targets.json

aws glue create-crawler `
    --name f1-driver-standings-raw-crawler `
    --role $ROLE_ARN `
    --database-name f1_db `
    --targets file://crawler_targets.json

Remove-Item crawler_targets.json

$crawlerTargets = @{
    S3Targets = @(
        @{ Path =
"s3://$BUCKET_NAME/processed/driver_standings_by_race" }
        @{ Path =
"s3://$BUCKET_NAME/processed/driver_standings_by_driver" }
    )
}
$crawlerTargets | ConvertTo-Json -Depth 10 | Set-Content
crawler_targets.json

aws glue create-crawler `
    --name f1-driver-standings-processed-crawler `
    --role $ROLE_ARN `
    --database-name f1_db `
    --targets file://crawler_targets.json

Remove-Item crawler_targets.json
Write-Host "[7/11] Subiendo scripts ETL a S3..."
-ForegroundColor Green

Set-Location (Join-Path $SrcDir "glue_jobs")
aws s3 cp driver_standings_aggregation_by_race.py
s3://$BUCKET_NAME/scripts/
```

```
aws s3 cp driver_standing_aggregation_by_driver.py
s3://$BUCKET_NAME/scripts/
Write-Host "[8/11] Creando Glue Jobs..." -ForegroundColor Green
Set-Location $ScriptsDir

$DATABASE = "f1_db"
$TABLE = "f1_driver_standings"

$commandConfig = @{ Name = "glueetl"; ScriptLocation =
"s3://$BUCKET_NAME/scripts/driver_standings_aggregation_by_race
.py"; PythonVersion = "3" }
$defaultArgs = @{ "--database" = $DATABASE; "--table" = $TABLE;
"--output_path" =
"s3://$BUCKET_NAME/processed/driver_standings_by_race/";
"--enable-continuous-cloudwatch-log" = "true";
"--spark-event-logs-path" = "s3://$BUCKET_NAME/logs/" }
$commandConfig | ConvertTo-Json | Set-Content job_command.json
$defaultArgs | ConvertTo-Json | Set-Content job_args.json

aws glue create-job `
    --name driver-standings-by-race `
    --role $ROLE_ARN `
    --command file://job_command.json `
    --default-arguments file://job_args.json `
    --glue-version "4.0" `
    --number-of-workers 2 `
    --worker-type "G.1X"

Remove-Item job_command.json, job_args.json

$commandConfig = @{ Name = "glueetl"; ScriptLocation =
"s3://$BUCKET_NAME/scripts/driver_standing_aggregation_by_drive
r.py"; PythonVersion = "3" }
$defaultArgs = @{ "--database" = $DATABASE; "--table" = $TABLE;
"--output_path" =
"s3://$BUCKET_NAME/processed/driver_standings_by_driver/";
"--enable-continuous-cloudwatch-log" = "true";
"--spark-event-logs-path" = "s3://$BUCKET_NAME/logs/" }
$commandConfig | ConvertTo-Json | Set-Content job_command.json
$defaultArgs | ConvertTo-Json | Set-Content job_args.json

aws glue create-job `
    --name driver-standings-by-driver `
    --role $ROLE_ARN `
    --command file://job_command.json `
    --default-arguments file://job_args.json `
    --glue-version "4.0" `
    --number-of-workers 2 `
    --worker-type "G.1X"

Remove-Item job_command.json, job_args.json

Write-Host "\n[9/11] Ejecutando productor Kinesis..."
-ForegroundColor Green
Set-Location (Join-Path $SrcDir "producer")
```

```
if (Test-Path ".venv\Scripts\python.exe") {
    & .venv\Scripts\python.exe kinesis.py
} else {
    & python kinesis.py
}

if ($LASTEXITCODE -ne 0) {
    Write-Host "ERROR: El productor Kinesis falló con código de
salida $LASTEXITCODE" -ForegroundColor Red
    exit 1
}
Write-Host "Productor Kinesis completado exitosamente"
-ForegroundColor Green
Write-Host "Esperando $WAIT_AFTER_COMPLETION segundos para que
Firehose procese..." -ForegroundColor Yellow
Start-Sleep -Seconds $WAIT_AFTER_COMPLETION

Write-Host "\n[10/11] Ejecutando Crawler RAW..."
-ForegroundColor Green
aws glue start-crawler --name f1-driver-standings-raw-crawler

Write-Host "Monitoreando crawler RAW..." -ForegroundColor
Yellow
$waited = 0
while ($waited -lt $MAX_WAIT_TIMEOUT) {
    $crawlerState = aws glue get-crawler --name
f1-driver-standings-raw-crawler --query 'Crawler.State'
--output text
    if ($crawlerState -eq "READY") {
        Write-Host "Crawler RAW completado" -ForegroundColor
Green
        break
    }
    Write-Host "  Estado: $crawlerState" -ForegroundColor Gray
    Start-Sleep -Seconds 10
    $waited += 10
}
Start-Sleep -Seconds $WAIT_AFTER_COMPLETION

Write-Host "\n[11/11] Ejecutando Glue Jobs..." -ForegroundColor
Green

if ($JobToRun -eq "race" -or $JobToRun -eq "both") {
    Write-Host "`nLanzando job: driver-standings-by-race"
-ForegroundColor Yellow
    $jobRun1 = aws glue start-job-run --job-name
driver-standings-by-race | ConvertFrom-Json
    $runId1 = $jobRun1.JobRunId
    Write-Host "Job iniciado con Run ID: $runId1"
-ForegroundColor Cyan

    Write-Host "Monitoreando ejecucion del job..."
-ForegroundColor Yellow
    $waited = 0
    while ($waited -lt $MAX_WAIT_TIMEOUT) {
```

```
        $jobStatus1 = (aws glue get-job-run --job-name
driver-standings-by-race --run-id $runId1 |
ConvertFrom-Json).JobRun.JobRunState
        if ($jobStatus1 -in @("SUCCEEDED", "FAILED", "STOPPED",
"TIMEOUT")) {
            Write-Host "Job terminado con estado: $jobStatus1"
-ForegroundColor $(if($jobStatus1 -eq
"SUCCEEDED"){"Green"}else{"Red"})
            break
        }
        Write-Host "  Estado: $jobStatus1 (esperando...)"
-ForegroundColor Gray
        Start-Sleep -Seconds 15
        $waited += 15
    }

    if ($waited -ge $MAX_WAIT_TIMEOUT) {
        Write-Host "Timeout esperando al job race.
Continuando..." -ForegroundColor DarkYellow
    }
    Start-Sleep -Seconds $WAIT_AFTER_COMPLETION
}

if ($JobToRun -eq "driver" -or $JobToRun -eq "both") {
    Write-Host "`nLanzando job: driver-standings-by-driver"
-ForegroundColor Yellow
    $jobRun2 = aws glue start-job-run --job-name
driver-standings-by-driver | ConvertFrom-Json
    $runId2 = $jobRun2.JobRunId
    Write-Host "Job iniciado con Run ID: $runId2"
-ForegroundColor Cyan

    Write-Host "Monitoreando ejecucion del job..."
-ForegroundColor Yellow
    $waited = 0
    while ($waited -lt $MAX_WAIT_TIMEOUT) {
        $jobStatus2 = (aws glue get-job-run --job-name
driver-standings-by-driver --run-id $runId2 |
ConvertFrom-Json).JobRun.JobRunState
        if ($jobStatus2 -in @("SUCCEEDED", "FAILED", "STOPPED",
"TIMEOUT")) {
            Write-Host "Job terminado con estado: $jobStatus2"
-ForegroundColor $(if($jobStatus2 -eq
"SUCCEEDED"){"Green"}else{"Red"})
            break
        }
        Write-Host "  Estado: $jobStatus2 (esperando...)"
-ForegroundColor Gray
        Start-Sleep -Seconds 15
        $waited += 15
    }

    if ($waited -ge $MAX_WAIT_TIMEOUT) {
        Write-Host "Timeout esperando al job driver.
Continuando..." -ForegroundColor DarkYellow
```

```
    }
    Start-Sleep -Seconds $WAIT_AFTER_COMPLETION
}

$shouldRunCrawler = $false

if ($JobToRun -eq "race" -and $jobStatus1 -eq "SUCCEEDED") {
    $shouldRunCrawler = $true
}
elseif ($JobToRun -eq "driver" -and $jobStatus2 -eq
"SUCCEEDED") {
    $shouldRunCrawler = $true
}
elseif ($JobToRun -eq "both" -and ($jobStatus1 -eq "SUCCEEDED"
-or $jobStatus2 -eq "SUCCEEDED")) {
    $shouldRunCrawler = $true
}

if ($shouldRunCrawler) {
    Write-Host "`nEjecutando crawler para tablas procesadas..."
-ForegroundColor Yellow
    aws glue start-crawler --name
f1-driver-standings-processed-crawler
    Write-Host "Crawler f1-driver-standings-processed-crawler
iniciado" -ForegroundColor Green

    Write-Host "Monitoreando crawler procesado..."
-ForegroundColor Yellow
    $waited = 0
    while ($waited -lt $MAX_WAIT_TIMEOUT) {
        $crawlerState = aws glue get-crawler --name
f1-driver-standings-processed-crawler --query 'Crawler.State'
--output text
        if ($crawlerState -eq "READY") {
            Write-Host "Crawler procesado completado"
-ForegroundColor Green
            break
        }
        Write-Host "  Estado: $crawlerState" -ForegroundColor
Gray
        Start-Sleep -Seconds 10
        $waited += 10
    }
    Start-Sleep -Seconds $WAIT_AFTER_COMPLETION
} else {
    Write-Host "`nNo se ejecutara el crawler procesado (ningun
job exitoso)" -ForegroundColor DarkYellow
}

Write-Host "`nEjecutando Athena queries..." -ForegroundColor
Green
Set-Location $ScriptsDir

if ($JobToRun -eq "race") {
    & .\athena_queries.ps1 -JobToRun "race"
```

```
} elseif ($JobToRun -eq "driver") {
    & .\athena_queries.ps1 -JobToRun "driver"
} else {
    & .\athena_queries.ps1 -JobToRun "both"
}

Write-Host "`n========================================"
-ForegroundColor Cyan
Write-Host "SETUP COMPLETADO" -ForegroundColor Green
Write-Host "========================================"
-ForegroundColor Cyan
Write-Host "`nEstado de los jobs:" -ForegroundColor Yellow

if ($JobToRun -eq "race" -or $JobToRun -eq "both") {
    aws glue get-job-runs --job-name driver-standings-by-race
--max-items 1
}

if ($JobToRun -eq "driver" -or $JobToRun -eq "both") {
    aws glue get-job-runs --job-name driver-standings-by-driver
--max-items 1
}

Write-Host "`nPipeline completado exitosamente"
-ForegroundColor Green
```

Código 1: Script setup.ps1

A[2]

```
# Script de limpieza - Elimina todos los recursos creados por
simple_script.ps1 y timed_script.ps1

$ScriptDir = Split-Path -Parent $MyInvocation.MyCommand.Path
$ProjectRoot = Split-Path -Parent $ScriptDir

$AWS_REGION = "us-east-1"
$ACCOUNT_ID = aws sts get-caller-identity --query Account
--output text
$BUCKET_NAME = "datalake-f1-driverstandings-$ACCOUNT_ID"

Write-Host "`n========================================"
-ForegroundColor Red
Write-Host "SCRIPT DE LIMPIEZA - F1 Data Lake" -ForegroundColor
Red
Write-Host "========================================"
-ForegroundColor Red
Write-Host "Bucket a eliminar: $BUCKET_NAME" -ForegroundColor
Yellow
Write-Host "Account ID: $ACCOUNT_ID" -ForegroundColor Yellow
Write-Host "========================================`n"
-ForegroundColor Red

Write-Host "[1/8] Eliminando Glue Jobs..." -ForegroundColor
```

```
Cyan
Write-Host "`n[1/8] Eliminando Glue Jobs..." -ForegroundColor
Cyan
try {
    aws glue delete-job --job-name driver-standings-by-race
    Write-Host "  Job 'driver-standings-by-race' eliminado"
-ForegroundColor Green
} catch {
    Write-Host "  Job 'driver-standings-by-race' no existe o ya
fue eliminado" -ForegroundColor DarkYellow
}

try {
    aws glue delete-job --job-name driver-standings-by-driver
    Write-Host "  Job 'driver-standings-by-driver' eliminado"
-ForegroundColor Green
} catch {
    Write-Host "  Job 'driver-standings-by-driver' no existe o
ya fue eliminado" -ForegroundColor DarkYellow
}


Write-Host "`n[2/8] Eliminando Glue Crawlers..."
-ForegroundColor Cyan
try {
    aws glue delete-crawler --name
f1-driver-standings-raw-crawler
    Write-Host "  Crawler RAW eliminado" -ForegroundColor Green
} catch {
    Write-Host "  Crawler RAW no existe o ya fue eliminado"
-ForegroundColor DarkYellow
}

try {
    aws glue delete-crawler --name
f1-driver-standings-processed-crawler
    Write-Host "  Crawler processed eliminado" -ForegroundColor
Green
} catch {
    Write-Host "  Crawler processed no existe o ya fue
eliminado" -ForegroundColor DarkYellow
}

Write-Host "`n[3/8] Eliminando Glue Database..."
-ForegroundColor Cyan
try {
    aws glue delete-database --name f1_db
    Write-Host "  Database 'f1_db' eliminada" -ForegroundColor
Green
} catch {
    Write-Host "  Database no existe o ya fue eliminada"
-ForegroundColor DarkYellow
}

Write-Host "`n[4/8] Eliminando Firehose Delivery Stream..."
```

```
-ForegroundColor Cyan
try {
    aws firehose delete-delivery-stream --delivery-stream-name
f1-driver-standings-delivery-stream
    Write-Host "  Firehose eliminado (puede tardar unos minutos
en completarse)" -ForegroundColor Green
    Start-Sleep -Seconds 30
} catch {
    Write-Host "  Firehose no existe o ya fue eliminado"
-ForegroundColor DarkYellow
}

Write-Host "`n[5/8] Eliminando Lambda Function..."
-ForegroundColor Cyan
try {
    aws lambda delete-function --function-name
f1-firehose-lambda
    Write-Host "  Lambda eliminada" -ForegroundColor Green
} catch {
    Write-Host "  Lambda no existe o ya fue eliminada"
-ForegroundColor DarkYellow
}

Write-Host "`n[6/8] Eliminando Kinesis Stream..."
-ForegroundColor Cyan
try {
    aws kinesis delete-stream --stream-name
f1-driver-standings-stream
    Write-Host "  Kinesis Stream eliminado" -ForegroundColor
Green
} catch {
    Write-Host "  Kinesis Stream no existe o ya fue eliminado"
-ForegroundColor DarkYellow
}

Write-Host "`n[7/8] Vaciando y eliminando S3 Bucket..."
-ForegroundColor Cyan
try {
    Write-Host "  Vaciando bucket..." -ForegroundColor Yellow
    aws s3 rm s3://$BUCKET_NAME --recursive
    Write-Host "  Eliminando bucket..." -ForegroundColor Yellow
    aws s3 rb s3://$BUCKET_NAME
    Write-Host "  Bucket eliminado" -ForegroundColor Green
} catch {
    Write-Host "  Error al eliminar bucket (puede que no
exista)" -ForegroundColor DarkYellow
}

Write-Host "`n[8/8] Limpiando archivos temporales locales..."
-ForegroundColor Cyan
Set-Location $ProjectRoot
$tempFiles = @(
    "src\firehose_driver_standings.zip",
    "scripts\firehose_config.json",
    "scripts\glue_db_input.json",
```

```
    "scripts\crawler_targets.json",
    "scripts\job_command.json",
    "scripts\job_args.json"
)

foreach ($file in $tempFiles) {
    if (Test-Path $file) {
        Remove-Item $file -Force
        Write-Host "  Eliminado: $file" -ForegroundColor Green
    }
}

Write-Host "  Limpiando archivos temporales de queries..."
-ForegroundColor Yellow
Set-Location $ScriptDir
Get-ChildItem -Filter "temp_query_*.sql" | Remove-Item -Force
-ErrorAction SilentlyContinue
Get-ChildItem -Filter "query_result_*.csv" | Remove-Item -Force
-ErrorAction SilentlyContinue
Write-Host "  Archivos temporales de queries eliminados"
-ForegroundColor Green

Write-Host "`n========================================"
-ForegroundColor Green
Write-Host "LIMPIEZA COMPLETADA" -ForegroundColor Green
Write-Host "========================================"
-ForegroundColor Green
Write-Host "`nTodos los recursos han sido eliminados."
-ForegroundColor Yellow
Write-Host "Nota: Algunos recursos como Firehose pueden tardar
unos minutos en eliminarse completamente.`n" -ForegroundColor
Gray
```

Código 2: script de limpieza cleanup.ps1

A[3]

```
# Script para ejecutar queries de Athena
# Parametros del script
param(
    [Parameter(Mandatory=$false)]
    [ValidateSet("race", "driver", "both")]
    [string]$JobToRun = "both"
)

# Variables
$ScriptDir = Split-Path -Parent $MyInvocation.MyCommand.Path
$AWS_REGION = "us-east-1"
$ACCOUNT_ID = aws sts get-caller-identity --query Account
--output text
$BUCKET_NAME = "datalake-f1-driverstandings-$ACCOUNT_ID"
$DATABASE = "f1_db"

Write-Host "`n========================================"
-ForegroundColor Cyan
```

```
Write-Host "F1 Driver Standings - Athena Queries"
-ForegroundColor Cyan
Write-Host "========================================"
-ForegroundColor Cyan
Write-Host "Database: $DATABASE" -ForegroundColor Yellow
Write-Host "Resultados en: s3://$BUCKET_NAME/queries/"
-ForegroundColor Yellow
Write-Host "Job seleccionado: $JobToRun" -ForegroundColor
Yellow
Write-Host "========================================`n"
-ForegroundColor Cyan

# Funcion para ejecutar query y esperar resultados
function Execute-AthenaQuery {
    param(
        [string]$QueryString,
        [string]$QueryName
    )

    Write-Host "`n[$QueryName]" -ForegroundColor Yellow
    Write-Host "Ejecutando query..." -ForegroundColor Gray

    # Guardar query en archivo temporal sin BOM
    $queryFile =
"temp_query_$([guid]::NewGuid().ToString()).sql"
    $QueryString | Out-File -FilePath $queryFile -Encoding
ASCII -NoNewline

    $execution = aws athena start-query-execution `
        --query-string file://$queryFile `
        --query-execution-context "Database=$DATABASE" `
        --result-configuration
"OutputLocation=s3://$BUCKET_NAME/queries/" | ConvertFrom-Json

    Remove-Item $queryFile -ErrorAction SilentlyContinue

    if (-not $execution.QueryExecutionId) {
        Write-Host "Error ejecutando query" -ForegroundColor
Red
        return
    }

    $executionId = $execution.QueryExecutionId
    Write-Host "Query ID: $executionId" -ForegroundColor Cyan

    # Esperar a que termine
    $maxWait = 60
    $waited = 0
    while ($waited -lt $maxWait) {
        $status = (aws athena get-query-execution
--query-execution-id $executionId |
ConvertFrom-Json).QueryExecution.Status.State

        if ($status -eq "SUCCEEDED") {
            Write-Host "Query completada exitosamente"
```

```
-ForegroundColor Green
            $resultPath =
"s3://$BUCKET_NAME/queries/$executionId.csv"
            Write-Host "Resultados: $resultPath"
-ForegroundColor Gray

            # Descargar y mostrar primeras lineas
            $localFile = "query_result_$executionId.csv"
            aws s3 cp $resultPath $localFile --quiet
            if (Test-Path $localFile) {
                Write-Host "`nPrimeras lineas del resultado:"
-ForegroundColor Cyan
                Get-Content $localFile -First 15 |
ForEach-Object { Write-Host "  $_" -ForegroundColor White }
                Remove-Item $localFile
            }
            break
        }
        elseif ($status -in @("FAILED", "CANCELLED")) {
            Write-Host "Query fallo con estado: $status"
-ForegroundColor Red
            break
        }

        Start-Sleep -Seconds 2
        $waited += 2
    }

    if ($waited -ge $maxWait) {
        Write-Host "Timeout esperando resultado"
-ForegroundColor DarkYellow
    }
}

Set-Location $ScriptDir

# Query 1: Top 10 pilotos por puntos totales (tabla raw)
$query1 = "SELECT driverId, forename, surname, SUM(points) as
total_points, COUNT(DISTINCT raceId) as num_races FROM
f1_db.f1_driver_standings GROUP BY driverId, forename, surname
ORDER BY total_points DESC LIMIT 10"

# Query 2: Estadisticas por carrera (tabla raw)
$query2 = "SELECT raceId, COUNT(DISTINCT driverId) as
num_drivers, SUM(points) as total_points, AVG(points) as
avg_points FROM f1_db.f1_driver_standings GROUP BY raceId ORDER
BY raceId LIMIT 10"

# Query 3: Pilotos con mas victorias
$query3 = "SELECT forename, surname, nationality, MAX(wins) as
total_wins FROM f1_db.f1_driver_standings GROUP BY forename,
surname, nationality HAVING MAX(wins) > 0 ORDER BY total_wins
DESC LIMIT 10"

# Query 4: Top pilotos por puntos agregados (si existe tabla
```

```
by_driver)
$query4 = "SELECT driverId, forename, surname, total_points,
posicion_promedio_historica, num_races FROM
f1_db.driver_standings_by_driver ORDER BY total_points DESC
LIMIT 10"

# Query 5: Carreras con mas participantes (si existe tabla
by_race)
$query5 = "SELECT raceId, total_drivers, total_points,
top_driver_forename, top_driver_surname, top_driver_victories
FROM f1_db.driver_standings_by_race ORDER BY total_drivers DESC
LIMIT 10"

# Ejecutar queries comunes (siempre se ejecutan)
Execute-AthenaQuery -QueryString $query1 -QueryName "Query 1:
Top 10 pilotos por puntos totales (raw)"
Execute-AthenaQuery -QueryString $query2 -QueryName "Query 2:
Estadisticas por carrera (raw)"
Execute-AthenaQuery -QueryString $query3 -QueryName "Query 3:
Pilotos con mas victorias"

# Ejecutar queries segun el parametro JobToRun
if ($JobToRun -eq "driver" -or $JobToRun -eq "both") {
    Write-Host "`n--- Query en tabla procesada by_driver ---"
-ForegroundColor DarkYellow
    Execute-AthenaQuery -QueryString $query4 -QueryName "Query
4: Top pilotos (tabla by_driver)"
}

if ($JobToRun -eq "race" -or $JobToRun -eq "both") {
    Write-Host "`n--- Query en tabla procesada by_race ---"
-ForegroundColor DarkYellow
    Execute-AthenaQuery -QueryString $query5 -QueryName "Query
5: Carreras con mas participantes (tabla by_race)"
}

Write-Host "`n========================================"
-ForegroundColor Cyan
Write-Host "Queries completadas" -ForegroundColor Green
Write-Host "========================================"
-ForegroundColor Cyan
$queriesPath = "s3://$BUCKET_NAME/queries/"
Write-Host "Todos los resultados estan en: $queriesPath"
-ForegroundColor Yellow
Write-Host "Puedes listarlos con el comando aws s3 ls
$queriesPath" -ForegroundColor Gray
```

Código 3: Script de queries de Athena athena_queries.ps1

A[4]

```
import boto3
import csv
import json
import sys
import time
```

```
from pathlib import Path
from loguru import logger

STREAM_NAME = 'f1-driver-standings-stream'
REGION = 'us-east-1'
BATCH_SIZE = 100
BATCH_DELAY = 2.1

# Ruta relativa al proyecto root (sube 2 niveles desde
src/producer/)
INPUT_FILE = Path(__file__).parent.parent.parent / 'data' /
'driver_standings_with_info.csv'
kinesis = boto3.client('kinesis', region_name=REGION)

def load_csv_data(file_path):
    data = []
    with open(file_path, 'r', encoding='utf-8') as f:
        reader = csv.DictReader(f)
        for row in reader:
            data.append(row)
    return data

def to_int(val, default=0):
    """Convierte cadenas numéricas a int de forma segura
(acepta '10.0')."""
    try:
        if val is None or val == '':
            return default
        return int(float(val))
    except Exception:
        return default

def run_producer():
    if not INPUT_FILE.exists():
        logger.error("Fichero no encontrado: %s", INPUT_FILE)
        logger.error("Genera el CSV con: python
src/merge_driver_standings.py\nLuego ejecuta: python
src/kinesis.py")
        sys.exit(1)

    data = load_csv_data(INPUT_FILE)
    total_records = len(data)
    records_sent = 0

    logger.info(f"Iniciando transmision al stream:
{STREAM_NAME}...")
    logger.info(f"Total de registros a enviar:
{total_records}")
    logger.info(f"Usando batches de {BATCH_SIZE} registros con
{BATCH_DELAY}s de espera entre batches")

    for i in range(0, total_records, BATCH_SIZE):
        batch = data[i:i+BATCH_SIZE]
        records = []
```

```
        for registro in batch:
            payload = {
                'driverStandingsId':
to_int(registro.get('driverStandingsId')),
                'raceId': to_int(registro.get('raceId')),
                'driverId': to_int(registro.get('driverId')),
                'points': to_int(registro.get('points')),
                'position': to_int(registro.get('position')),
                'positionText': registro.get('positionText'),
                'wins': to_int(registro.get('wins')),
                'forename': registro.get('forename'),
                'surname': registro.get('surname'),
                'dob': registro.get('dob'),
                'nationality': registro.get('nationality')
            }

            records.append({
                'Data': json.dumps(payload),
                'PartitionKey': str(registro['driverId'])
            })

        try:
            response = kinesis.put_records(
                StreamName=STREAM_NAME,
                Records=records
            )

            failed_count = response['FailedRecordCount']
            if failed_count > 0:
                logger.warning(f"Batch {i//BATCH_SIZE + 1}:
{failed_count} registros fallaron")

            records_sent += len(records) - failed_count
            logger.info(f"Progreso:
{records_sent}/{total_records} registros enviados
({(records_sent/total_records)*100:.1f}%)")

            if i + BATCH_SIZE < total_records:
                time.sleep(BATCH_DELAY)

        except Exception as e:
            logger.error(f"Error enviando batch: {e}")
            continue

    logger.success(f"Transmision completada! Total registros
enviados: {records_sent}/{total_records}")

    if records_sent < total_records:
        logger.warning(f"Algunos registros no se enviaron:
{total_records - records_sent} fallaron")

if __name__ == '__main__':
    run_producer()
```

Código 4: Kinesis.py

A[5]

```
import json
import base64
from datetime import datetime

def lambda_handler(event, context):
    output = []
    # Obtener fecha actual en formato YYYY-MM-DD
    partition_date = datetime.utcnow().strftime('%Y-%m-%d')

    for record in event['records']:
        payload =
base64.b64decode(record['data']).decode('utf-8')

        # El payload viene como string de dict, convertirlo a
dict real
        data_dict = json.loads(payload)

        # Eliminar positionText (campo redundante)
        data_without_redundant = {k: v for k, v in
data_dict.items() if k not in ['positionText']}

        # Convertir a JSON string válido
        data_json = json.dumps(data_without_redundant)

        output_record = {
            'recordId': record['recordId'],
            'result': 'Ok',
            'data': base64.b64encode((data_json +
'\n').encode('utf-8')).decode('utf-8'),
            'metadata': {
                'partitionKeys': {
                    'partition_date': partition_date
                }
            }
        }
        output.append(output_record)

    return {'records': output}
```

Código 5: Firehose_driver_standings.py(código lambda)

A[6]

```
import sys
import logging
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from pyspark.sql.functions import col, sum as spark_sum, count,
countDistinct, first, when
from pyspark.sql.window import Window
from awsglue.dynamicframe import DynamicFrame

logging.basicConfig(level=logging.INFO, format='%(asctime)s -
```

```
%(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

def main():
    args = getResolvedOptions(sys.argv, ['database', 'table',
'output_path'])
    database = args['database']
    table = args['table']
    output_path = args['output_path']

    logger.info(f"Database: {database}, Table: {table}, Output:
{output_path}")

    sc = SparkContext()
    glueContext = GlueContext(sc)

    dynamic_frame =
glueContext.create_dynamic_frame.from_catalog(
        database=database,
        table_name=table
    )

    df = dynamic_frame.toDF()
    df.printSchema()
    logger.info(f"Registros leídos: {df.count()}")

    df_with_wins = df.withColumn(
        "is_winner",
        when(col("position") == 1, 1).otherwise(0)
    )

    wins_by_driver = df_with_wins.groupBy("raceId", "driverId",
"forename", "surname") \
        .agg(
            spark_sum("is_winner").alias("num_victories")
        )

    window =
Window.partitionBy("raceId").orderBy(col("num_victories").desc(
))
    from pyspark.sql.functions import row_number

    top_driver = wins_by_driver \
        .withColumn("rank", row_number().over(window)) \
        .filter(col("rank") == 1) \
        .select("raceId", "driverId", "forename", "surname",
"num_victories")

    race_df = df.groupBy("raceId") \
        .agg(
            countDistinct("driverId").alias("total_drivers"),
            spark_sum("points").alias("total_points"),
            count("*").alias("total_records")
        )
```

```
    result_df = race_df.join(top_driver, "raceId", "left") \
        .select(
            "raceId",
            "total_drivers",
            "total_points",
            "total_records",
            col("driverId").alias("top_driver_id"),
            col("forename").alias("top_driver_forename"),
            col("surname").alias("top_driver_surname"),
            col("num_victories").alias("top_driver_victories")
        ) \
        .orderBy("raceId")

    output_dynamic_frame = DynamicFrame.fromDF(result_df,
glueContext, "output")

    logger.info(f"Registros agregados:
{output_dynamic_frame.count()}")

    glueContext.write_dynamic_frame.from_options(
        frame=output_dynamic_frame,
        connection_type="s3",
        connection_options={
            "path": output_path,
            "partitionKeys": ["raceId"]
        },
        format="parquet",
        format_options={"compression": "snappy"}
    )

    logger.info(f"Completado. Registros: {result_df.count()}")

if __name__ == "__main__":
    main()
```

Código 6: driver_standings_aggregation_by_race.py(ETL race)

A[7]

```
import sys
import logging
from pyspark.context import SparkContext
from awsglue.context import GlueContext
from awsglue.utils import getResolvedOptions
from pyspark.sql.functions import col, min as spark_min, max as
spark_max, avg, count, stddev
from awsglue.dynamicframe import DynamicFrame

logging.basicConfig(level=logging.INFO, format='%(asctime)s -
%(levelname)s - %(message)s')
logger = logging.getLogger(__name__)

def main():
    args = getResolvedOptions(sys.argv, ['database', 'table',
'output_path'])
```

```
    database = args['database']
    table = args['table']
    output_path = args['output_path']

    logger.info(f"Database: {database}, Table: {table}, Output:
{output_path}")

    sc = SparkContext()
    glueContext = GlueContext(sc)

    dynamic_frame =
glueContext.create_dynamic_frame.from_catalog(
        database=database,
        table_name=table
    )

    df = dynamic_frame.toDF()
    df.printSchema()
    logger.info(f"Registros leídos: {df.count()}")

    # Agregación por piloto: estadísticas de cada driverId
    # Usamos first() para obtener forename y surname (son
constantes por driverId)
    from pyspark.sql.functions import first, sum as spark_sum

    driver_df = df.groupBy("driverId") \
        .agg(
            first("forename").alias("forename"),
            first("surname").alias("surname"),

spark_min("position").alias("mejor_posicion_historica"),

avg("position").alias("posicion_promedio_historica"),

stddev("position").alias("desviacion_estandar_posicion"),
            count("raceId").alias("num_races"),
            spark_sum(col("points")).alias("total_points"),
            avg("points").alias("avg_points")
        ) \
        .orderBy("driverId")

    output_dynamic_frame = DynamicFrame.fromDF(driver_df,
glueContext, "output")

    logger.info(f"Registros agregados:
{output_dynamic_frame.count()}")

    # Escribir sin particionKeys ya que driverId es la clave de
agregación
    glueContext.write_dynamic_frame.from_options(
        frame=output_dynamic_frame,
        connection_type="s3",
        connection_options={
            "path": output_path
        },
```

```
        format="parquet",
        format_options={"compression": "snappy"}
    )

    logger.info(f"Completado. Registros: {driver_df.count()}")

if __name__ == "__main__":
    main()
```

Código 7: driver_standings_aggregation_by_driver.py(ETL driver)

A[8]

```
import pandas as pd
from pathlib import Path

# Obtener la ruta del directorio del script
script_dir = Path(__file__).parent
project_dir = script_dir.parent
dataset_dir = project_dir / 'dataset'

# Leer los archivos CSV
driver_standings = pd.read_csv(dataset_dir /
'driver_standings.csv')
drivers = pd.read_csv(dataset_dir / 'drivers.csv')

# Seleccionar solo las columnas necesarias de drivers
drivers_subset = drivers[['driverId', 'forename', 'surname',
'dob', 'nationality']]

# Hacer el merge (join) por driverId
merged_df = pd.merge(
    driver_standings,
    drivers_subset,
    on='driverId',
    how='left'
)

# Guardar el resultado en un nuevo CSV
merged_df.to_csv(dataset_dir /
'driver_standings_with_info.csv', index=False)

print(f"Archivo creado exitosamente:
driver_standings_with_info.csv")
print(f"Total de filas: {len(merged_df)}")
print(f"\nPrimeras filas del resultado:")
print(merged_df.head())
```

Código 8: merge_driver_standings.py (script que une el csv de los drivers y de los standings)

A[9]

```
driverStandingsId,raceId,driverId,points,position,positionText,w
```

```
ins,forename,surname,dob,nationality
1,18,1,10.0,1,1,1,Lewis,Hamilton,1985-01-07,British
2,18,2,8.0,2,2,0,Nick,Heidfeld,1977-05-10,German
3,18,3,6.0,3,3,0,Nico,Rosberg,1985-06-27,German
4,18,4,5.0,4,4,0,Fernando,Alonso,1981-07-29,Spanish
5,18,5,4.0,5,5,0,Heikki,Kovalainen,1981-10-19,Finnish
6,18,6,3.0,6,6,0,Kazuki,Nakajima,1985-01-11,Japanese
7,18,7,2.0,7,7,0,Sébastien,Bourdais,1979-02-28,French
8,18,8,1.0,8,8,0,Kimi,Räikkönen,1979-10-17,Finnish
9,19,1,14.0,1,1,1,Lewis,Hamilton,1985-01-07,British
10,19,2,11.0,3,3,0,Nick,Heidfeld,1977-05-10,German
```

Código 9: Fragmento del Dataset generado

A[10]

Uso de IA Generativa:

- Adaptación de código entre lenguajes Bash y PowerShell
- Redacción y revisión de la memoria técnica.
- Corrección de fallos

Todo el código/redacciones generado o modificado con asistencia de IA ha sido revisado, refactorizado y probado en el entorno y validado.