Informe:
Práctica 5: Ingesta, almacenamiento y Serveless

Jordi Blasco Lozano
Infraestructuras y Servicios Cloud
Universidad de Alicante

20 de noviembre de 2025

Resumen

Índice

1 Flujo IoT: Ingesta de Telemetría y Almacenamiento Dual

1.1 Preparación de Destinos y Configuración IoT

1.1.1 Creación de la Tabla DynamoDB

Se ha creado una tabla en DynamoDB llamada `vigía_farolas_estado` para almacenar el último estado reportado por cada farola junto con métricas del consumo y el momento de medir estas métricas. La clave de la tabla será la id de cada farola. Solamente indicando como queremos la clave de partición nuestra tabla será funcional.

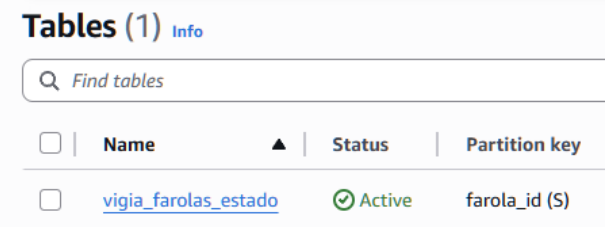


Figura 1: Tablas DynamoDB

1.1.2 Configuración de AWS IoT Core

- **Conexión del dispositivo:** Mediante el asistente de AWS IoT Core registré un dispositivo nuevo. Durante este proceso, se generaron y descargaron los certificados de seguridad (`farolasestado.cert.pem`, la clave privada correspondiente y publica correspondiente). Esto nos servirá para realizar la prueba de conexión y para conectarnos mediante Node-RED posteriormente.
- **Prueba de conexión:** Para verificar la conectividad, se ejecutó el script de prueba `start.ps1` proporcionado por AWS. El script instaló las dependencias necesarias y envió con éxito cinco mensajes de prueba, que fueron validados tanto en la terminal local como en el cliente de pruebas MQTT en el tema sdk/test/js en el mismo panel de conexión de dispositivos IoT.

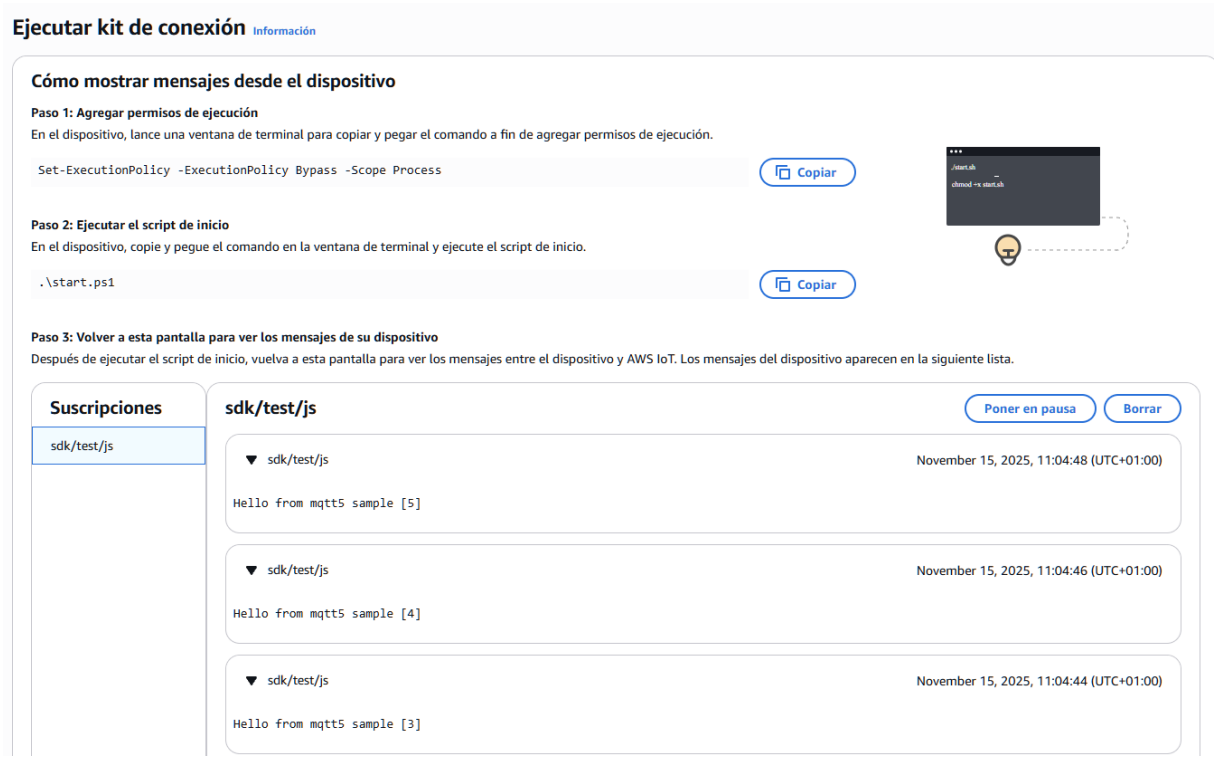


Figura 2: Test start.ps1

- **Cambio en la Política:**
  Automáticamente se asocia una política de seguridad, esta política es bastante restrictiva la cambiaremos para que se permitan la conexiones y publicaciones a cualquier tema, yo lo haré en el tema`"smartcity/consumo/test"`. Usamos 'test' porque en ningún momento serán farolas reales por lo que en un supuesto despliegue de la aplicación deberíamos de cambiar el tema. Cabe recalcar que para activar la política debemos de crear una nueva versión seleccionarla como activa y eliminar la versión anterior si no la vamos a volver a usar.

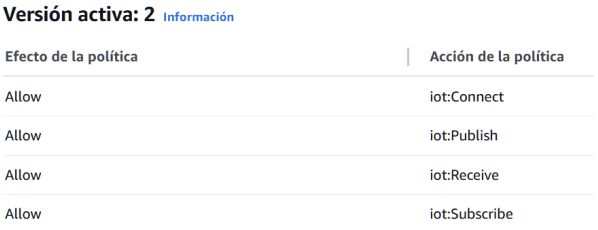


Figura 3: Política activa

1.2 Creación de la Regla IoT

1.2.1 Consulta SQL

Antes de definir la regla, se creó un bucket en S3 para el almacenamiento histórico de la telemetría. La regla de IoT se configuró para procesar los mensajes que llegan al tema `smartcity/consumo/test`. Para la consulta SQL tuve que cambiar un par de instrucciones ya que no me funcionaba correctamente el `processed_time`. Luego veremos lo que use en la función de `Node-RED` pero por ahora analizaremos la consulta SQL que he utilizado.

Bloque 1: Consulta SQL para la regla IoT

```
SELECT
 consumption_kw,
 timestamp,
 concat(day, '-', month, '-', year, ' ', hour, ':', minute, ':', seconds) AS
processed_time
FROM
 'smartcity/consumo/test'
```

Como vemos en el SQL hemos quitado el **home_id** y hemos cambiado el **processed_time**, esto se debe a que AWS no me dejaba utilizar DynamoDBv2 con plantilla y al usar DynamoDB legacy la `clave_id` la he mapeado directamente en el apartado correspondiente de DynamoDB, de esta forma ya no me apareciera doble en la tabla de Dynamo (como clave y como atributo). Posteriormente cambie el `proccesed_time` porque no hubo forma de procesar el tiempo con la función propuesta. Esta ha sido la forma más sencilla de obtener el tiempo procesado. Podría haberlo procesado al crear el mensaje cambiando la función del Node-RED o después al recibir el mensaje, y he decidido hacerlo después para que las carpetas del S3 pudieran mapear mucho mas facil el tiempo, y así crear las carpetas sin errores.

1.2.2 Acción S3 bucket

En esta acción también cambiamos un apartado, la "clave", en el enunciado se pedía utilizar un almacenamiento en carpetas clasificadas por año por mes por dia, el problema que le vi principalmente fue que en un sistema real podríamos tener diferentes farolas por lo que utilice una ultima clasificación que en la carpeta dias tuviera carpetas por "farola_id". Dentro de esta ultima carpeta ya tenemos cada archivo ordenado por la hora, minuto y segundo usando "hh:mm:ss-data.json" en vez de con timestamp (que a mi se me ponia en milisegundos). He usado la siguiente clave con los parámetros obtenidos directamente desde sus variables del mensaje (en vez de desde el timestamp).

Bloque 2: Clave del S3

```
 farolas/year=${year}/month=${month}/day=${day}/${home_id}/${hour}:${minute}:${
seconds}-data.json
```

1.2.3 Accion DynamoDB

En esta otra acción se pedía usar DynamoDBv2, despues de probar mil formas cambie a la versión a la de DynamoDB normal en la cual nosotros mismos debemos de vincular la variable clave para usarla como clave de particion de nuestra tabla. Lo hice manualmente porque no sabia como insertar la plantilla de atributos que proporciona el enunciado. Y no conseguí que las claves se vincularan automáticamente, por lo que lo hice manualmente con DynamoDB de la siguiente forma.

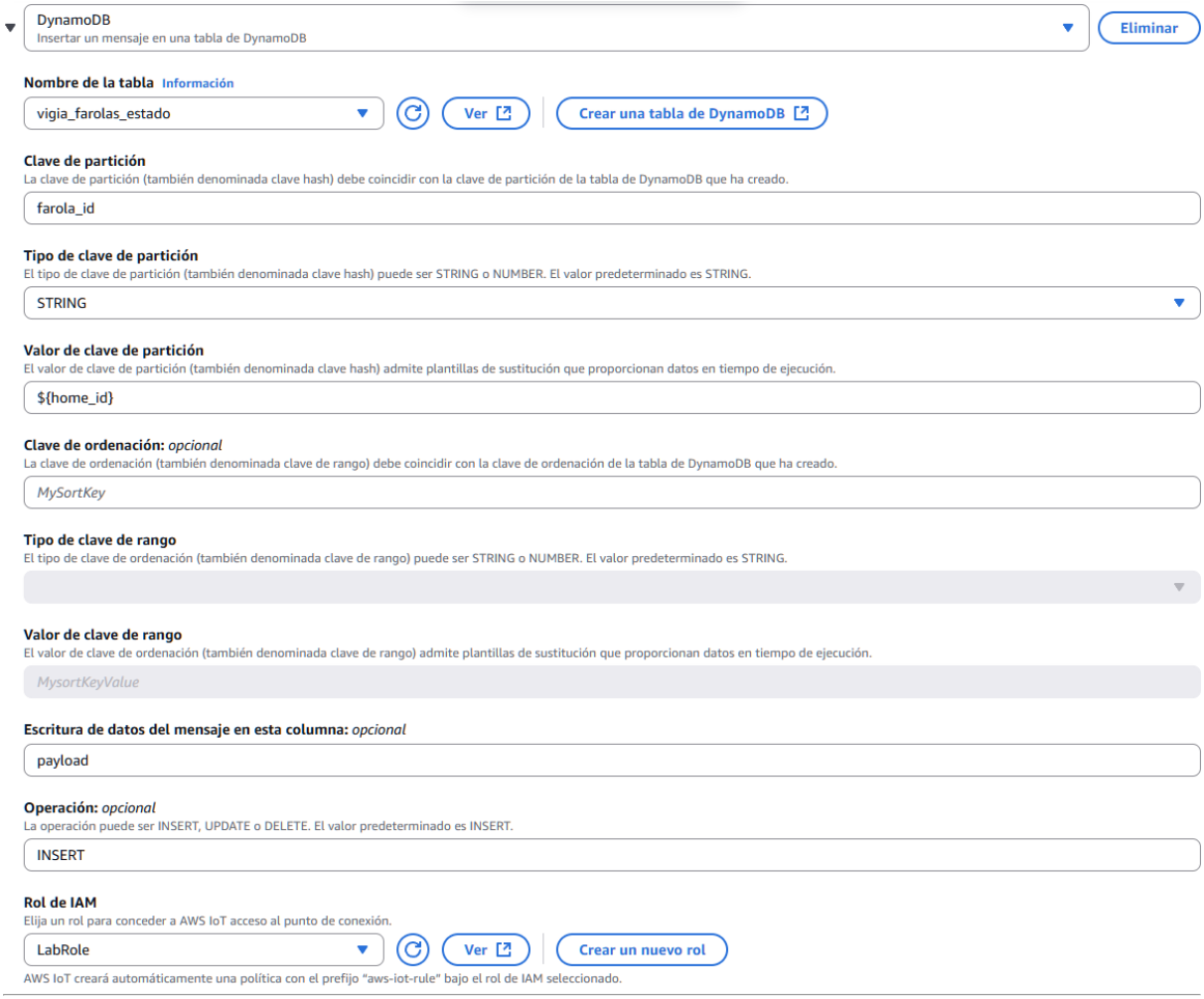


Figura 4: Configuración de la acción DynamoDB

1.3 Simulación de ingesta con Node-RED

Node-RED permite conectarse mediante MQTT a nuestro tema de 'thing' de IoT para simular el envío de datos de las farolas. El flujo consta de tres nodos:

- **Inject:** Un disparador manual para iniciar el flujo, cada 10 segundos se envia una señal al siguiente nodo para que mande un mensaje.
- **Function:** Un nodo que construye el mensaje JSON con los datos de la farola (ID, consumo y el tiempo) y establece el tema de destino en `smartcity/consumo/test`. Este nodo consta de una serie de variables que hemos randomizado para simular la farola. He utilizado 6 variables de más para el tiempo, de forma que tengamos en variables separadas el año, el mes, el dia, la hora, los minutos y los segundos. Consiguiendo que los pasos anteriores hayan sido más sencillos de implementar sabiendo que no tenemos que parsear ni usar funciones diferentes para obtener los datos del tiempo.

Bloque 3: Funcion constructora del mensaje

```
const now = new Date();
msg.topic = "smartcity/consumo/test";
msg.payload = {
    home_id: "farola-001",
    consumption_kw: +(Math.random() * 2).toFixed(2),
    timestamp: now.getTime(),
    year: now.getFullYear(),
    month: String(now.getMonth() + 1).padStart(2, "0"),
    day: String(now.getDate()).padStart(2, "0"),
    hour: String(now.getHours()).padStart(2, "0"),
    minute: String(now.getMinutes()).padStart(2, "0"),
    seconds: String(now.getSeconds()).padStart(2, "0"),
};
return msg;
```

- **MQTT Out:** Este nodo está configurado para publicar el mensaje en el endpoint de AWS IoT (a1paa0c5brn8mp-ats.iot.us-east-1.amazonaws.com) a través del puerto 8883 con TLS. La autenticación se realizó utilizando los mismos certificados de dispositivo que en la prueba de conexión inicial. Usamos TLS el cual contiene todos los certificados necesarios para conecarnos al endpoint que anteriormente hemos introducido.

Con estos tres nodos configurados tenemos ya el flujo funcional, y en darle a instanciar tendremos el sistema enviando mensajes cada 10 segundos al tema de MQTT de nuestro objeto. Para parar el sistema es tan simple como dehabilitar el nodo del disparador y volver a darle a instanciar.

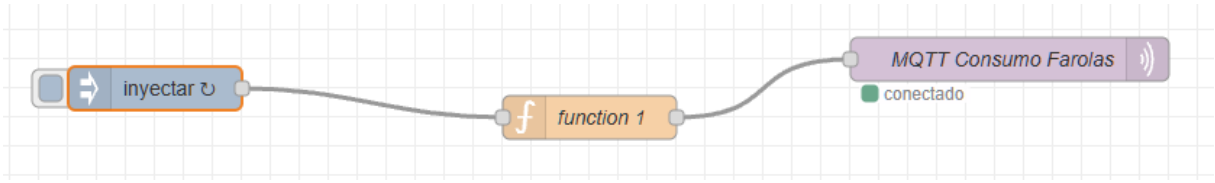


Figura 5: Flujo Node-RED

1.3.1 Pruebas

- **Cliente de prueba MQTT:** Para comprobar que cada una de las acciones anteriores de la regla funcionen debemos de comprobarlo lanzando el Node-RED. El primer paso es comprobar que los mensajes son enviados al tema correspondiente, para esto nos vamos al cliente de prueba de MQTT dentro de la pestaña de IoT de AWS y nos subscribimos a `smartcity/consumo/#` cuando nos subscribamos nos empezaran a salir mensajes como estos.

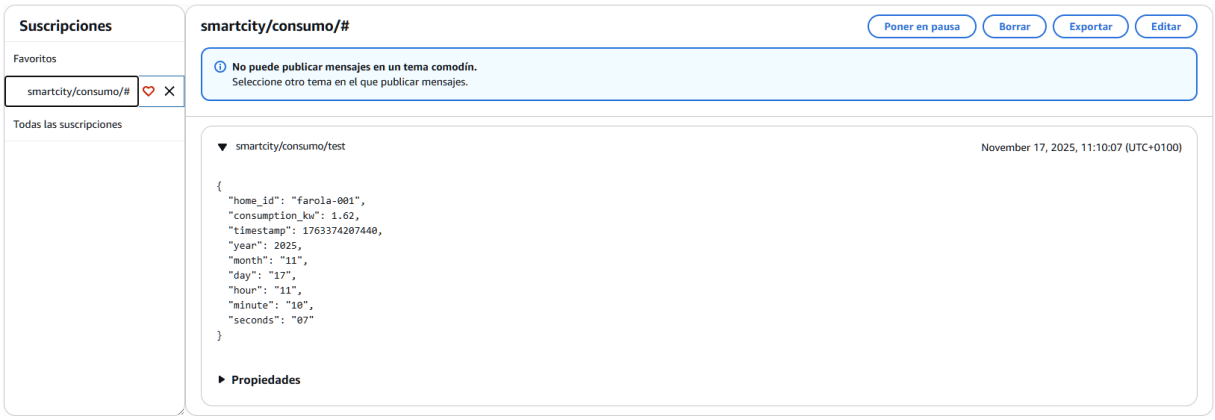


Figura 6: Cliente de prueba de MQTT

- **S3:** Respecto al guardado de mensajes historicos debemos de comprobarlo dentro del S3 de la practica, nos dirigimos a la sucesiones de carpetas que tenemos configurada y dentro de la ultima vemos como tenemos estos datos. Los datos de dentro de cada archivo tambien corresponden con el .json definido en el SQL, ej:

```
{"consumption_kw":0.3,"timestamp":1763374077358,"processed_time":"17-11-2025
 11:07:57"}
```

Figura 7: S3 bucket

- **DynamoDB** Finalemente debemos de comprobar la tabla de DynamoDB, dentro de `vigia_farolas_estado`, y en explorar elementos de la tabla vemos como nuestra tabla contiene una `farola001` con el json correspondiente. Este json se va actualizando cada 10 segundos y muestra siempre el estado actual de la farola. Si tuviéramos mas farolas se mostraría una tabla con el estado actual de cada una de las farolas.

Figura 8: Tabla DynamoDB elementos

2 Flujo de Contexto: Web Scraping con Lambda Serverless (RA2)

2.1 Creación de la Función Lambda

Para crear la función Lambda debemos de ingresar al servio Lambda y crear una función de python desde cero con los permisos de Lab-Role para poder ejecutar correctamente el proceso y realizar el PutObject al S3. Una vez dentro se nos abrirá un editor de código, que en si es un fork web de visual Studio Code. Debemos de pasar la variable por entorno, para esto dentro del panel de configuración ingresamos a variables de entorno y copiamos el nombre del bucket con el nombre de la variable del código proporcionado.

Figura 9: Variable de entorno

Cuando tengamos la variable de entorno copiamos el código que simulará nuestra ingesta de datos de contexto aletorizando la tarifa de luz del día `tarifakwh`. Tras copiar el código ya podemos desplegarlo.

2.2 Orquestación Serverless (EventBridge)

Al igual que hemos hecho en el apartado anterior con la ingesta de telemetría mediante la simulación por Node-RED, en este apartado también tenemos que simular una entrada programada cada cierto tiempo. En este caso mandaremos una señal a nuestra función Lambda cada 1 minuto. Esto lo haremos mediante el servicio de Amazon EventBridge, este nos permitirá crear un evento programado al destino de AWS que nosotros queramos. Lo configuraremos de la siguiente forma eligiendo nuestra función Lambda y poniendo una programación que se repita cada minuto.

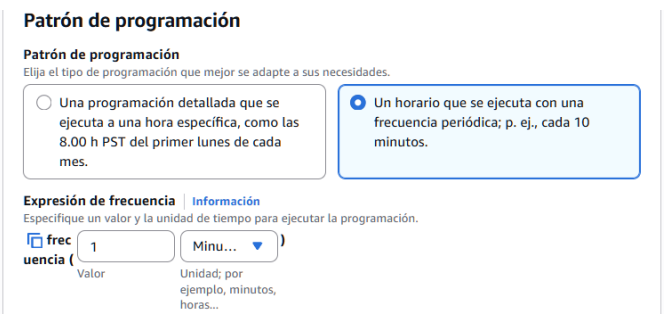


Figura 10: Frecuencia Event Bridge

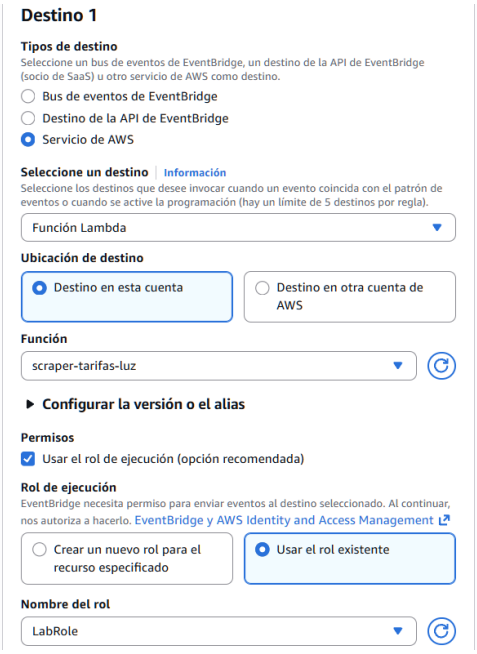


Figura 11: Destino Event Bridge

Si nos fijamos en el código de nuestra función, el guardado de datos se sobrescribira al poner como nombre DD-MM-YYYY-data.json. Pero podremos comprobar que se sobrescriba correctamente comprobando el contenido del archivo del S3 cada minuto. Nuestros archivos `-data.json` tendrán la estructura de este ejemplo.

```
{"timestampscraper": "2025-11-17T18:20:11.307175", "tarifakwh": 0.1833, "ciudad": "
 Neo-Tech"}
```

Nuestra carpeta dentro del bucket dentra esta otra forma.

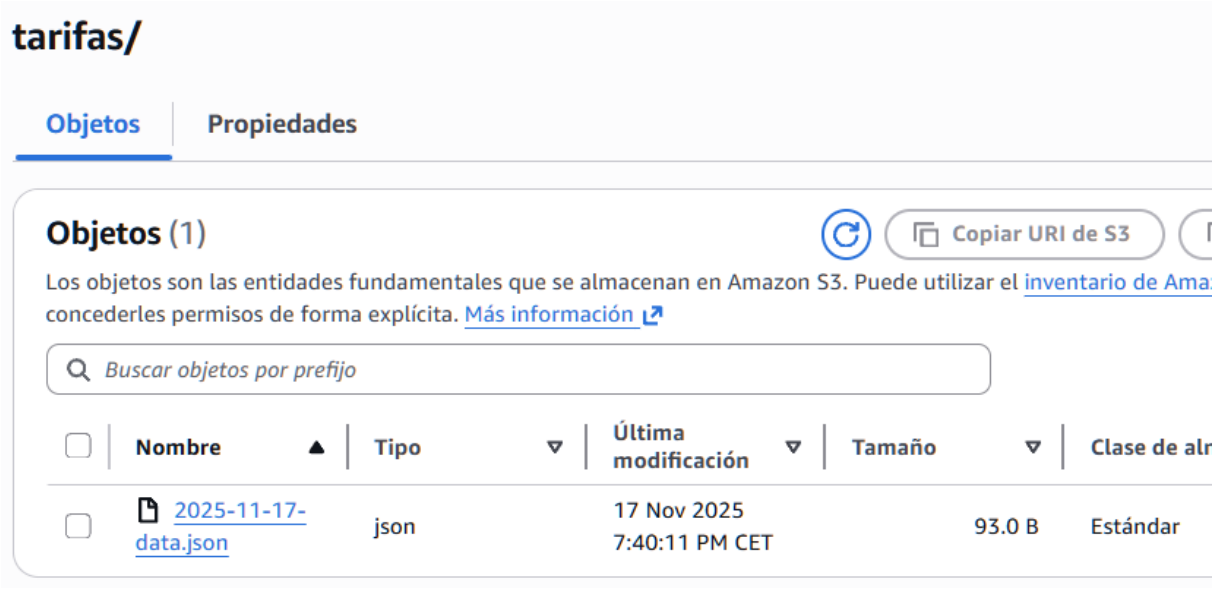


Figura 12: S3 tarifas

Finalmente en los registros de cloudWatch podemos ver las invocaciones a la función mas recientes.

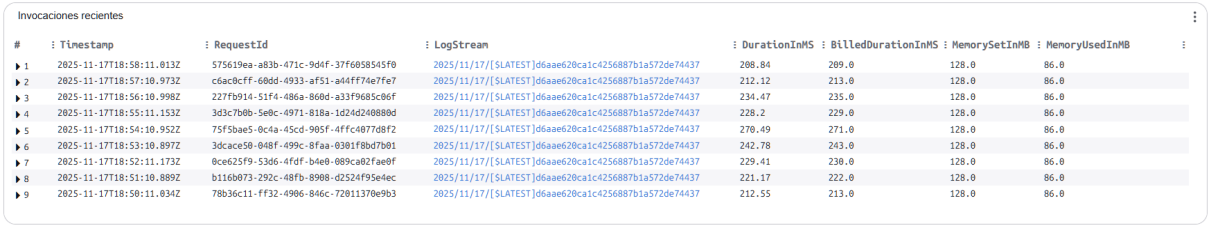


Figura 13: S3 tarifas

3 Flujo Masivo: Transformación y Análisis Serverless

3.1 Carga y Catalogación (RA 1)

Para cargar el dataset historico de datos de movilidad hice uso de una carpeta nueva en nuestro S3 de la practica, la llame `/historicomovilidad` e introduje el `.csv` proporcionado en la practica, utilice el de 10 filas para evitar consumos. Antes de subirlo lo abrí en visual studio code y me di cuenta de que me saltaba el linter porque al haber eliminado la categoria repetida `Airport_fee` no me había fijado que tambíen debía eliminar las comas de esta columna.

3.1.1 Carga Masiva

Con el `.csv` listo y la carpeta creada cargué el archivo.

Figura 14: csv historico movilidad

3.1.2 AWS Glue Crawler

Mediante AWS Glue Crawler, automatizamos el proceso de descubrimiento y catalogación de los datos almacenados en el S3. El Crawler escanea los archivos en la carpeta, infiere su esquema (nombres de columnas, tipos de datos y particiones) y crea las Glue Tables dentro de nuestra base de datos.

Es importante destacar que las tablas creadas en el Data Catalog no contienen los datos en sí, sino metadatos: apuntan a la ubicación física de los archivos en S3 y definen su estructura. Esto actúa como una capa de abstracción que permite a servicios como Amazon Athena consultar los archivos CSV (u otros formatos) directamente usando SQL estándar, como si se tratara de una base de datos relacional tradicional, sin necesidad de mover o cargar los datos previamente. El crawler lo configuramos de la siguiente forma:

- **Creación del Crawler:** en este apartado debemos de definir el nombre del crawler (el mio vigia-crawler) y dentro de su panel de configuración debemos de añadir nuestra fuente de datos (la dirección S3 de nuestra carpeta `/historicomovilidad` que creamos anteriormente). Posteriormente asignamos nuestro IAM role (el LabRole) y finalmente creamos una base de datos vacia para que el Crawler genere las tablas dentro de ella. Al acabar estos pasos debemos de tener un Crawler como este:

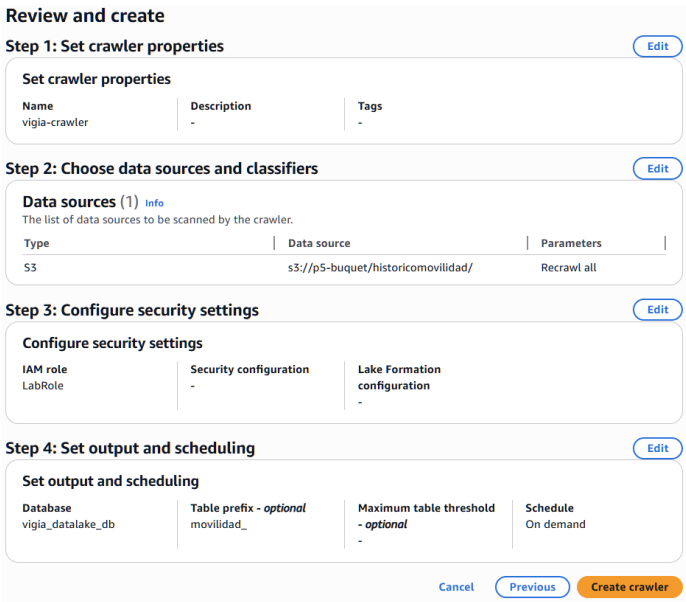
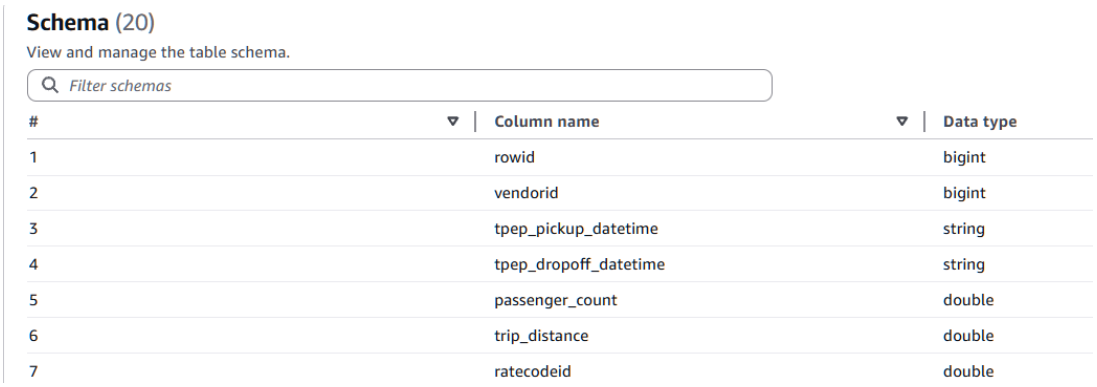


Figura 15: Crawler

- **Ejecución del Crawler:** tras terminar la configuración del Crawler debemos de ejecutarlo. Durante este proceso de ejecución el crawler lo podemos observar en los logs de ejecución: el crawler `vigia-movilidad-crawler` inicia el proceso de clasificación de los datos. Una vez analizada la estructura de los archivos en S3, escribe los resultados en la base de datos `vigia_datalake_db`, creando exitosamente la tabla `historicomovilidad`. Finalmente, actualiza el Data Catalog (indicando `ADD: 1` en los logs) y vuelve al estado `READY`, dejando los datos listos para ser consultados. Para comprobar si ha el crawler ha conseguido analizar correctamente la escructura accedemos a las tablas de nuestro Data Catalog y dentro de la tabla `historicomovilidad` deberá de aparecernos este esquema:



Schema (20)

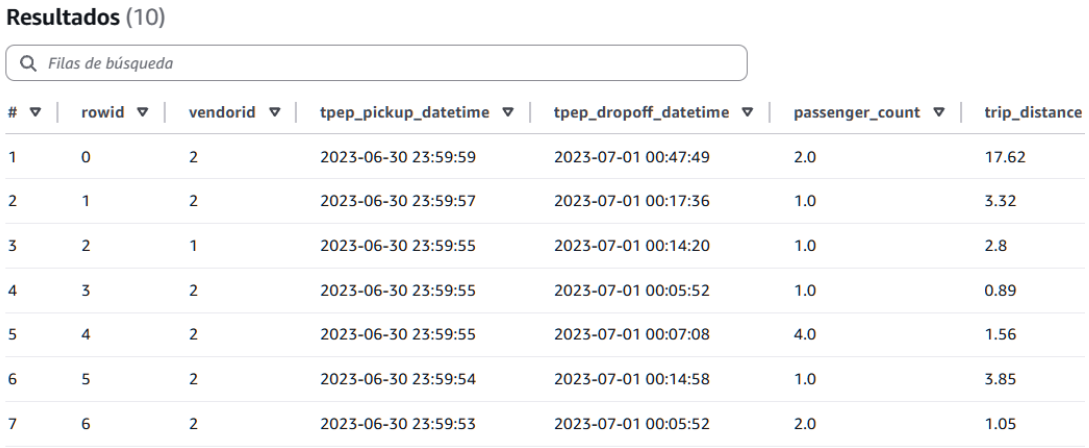
View and manage the table schema.

Filter schemas

#	Column name	Data type
1	rowid	bigint
2	vendorid	bigint
3	tpep_pickup_datetime	string
4	tpep_dropoff_datetime	string
5	passenger_count	double
6	trip_distance	double
7	ratecodeid	double

Figura 16: Table Schema

- **Consulta SQL:** Al tener la tabla ya creada, podemos consultar el archivo csv directamente con SQL mediante el servicio de Athena que proporciona amazon. Para las guardar las consultas lo optimo será crear un S3 nuevo que en mi caso he llamado `resultados-p5-backet` que guardará tanto las consultas que hagamos con esta tabla (de datos en bruto) como las que hagamos con la tabla (analítica). He obtenido estos resultados al hacer la consulta:



Resultados (10)

Filas de búsqueda

#	rowid	vendorid	tpep_pickup_datetime	tpep_dropoff_datetime	passenger_count	trip_distance
1	0	2	2023-06-30 23:59:59	2023-07-01 00:47:49	2.0	17.62
2	1	2	2023-06-30 23:59:57	2023-07-01 00:17:36	1.0	3.32
3	2	1	2023-06-30 23:59:55	2023-07-01 00:14:20	1.0	2.8
4	3	2	2023-06-30 23:59:55	2023-07-01 00:05:52	1.0	0.89
5	4	2	2023-06-30 23:59:55	2023-07-01 00:07:08	4.0	1.56
6	5	2	2023-06-30 23:59:54	2023-07-01 00:14:58	1.0	3.85
7	6	2	2023-06-30 23:59:53	2023-07-01 00:05:52	2.0	1.05

Figura 17: Table Schema

3.2 Optimización del Data Lake (AWS Glue ETL) (RA 2)

En este paso, implementaremos un proceso ETL (Extract, Transform, Load) utilizando un Job de AWS Glue. El objetivo principal es transformar los datos crudos que tenemos en formato CSV a un formato optimizado para analítica (Parquet) y particionarlos adecuadamente. Esto nos servirá para mejorar drásticamente el rendimiento de las consultas en Athena y reducir los costos, ya que Parquet es un formato columnar comprimido que permite leer solo las columnas necesarias, y el particionado permite escanear solo los datos relevantes para una consulta (por ejemplo, filtrar solo por un año y mes específicos).

El Job realizará las siguientes acciones clave:

- **Extracción:** Leerá los datos de la tabla original catalogada en el Data Lake.
- **Transformación:** Convertirá los tipos de datos (casteo de strings a double/timestamp), limpiará el formato de fecha y seleccionará las columnas críticas para el análisis.
- **Carga:** Escribirá los datos procesados en una nueva carpeta `/analitica` del S3 en formato Parquet, particionados por año y mes.

3.2.1 Creación y Propiedades del Job

Para crear nuestro job nos dirigimos a AWS glue, a ETL Jobs y creamos uno nuevo. Este paso es sencillo ya que al tener ya el código python tan solo tendremos que pegarlo en la pestaña de scripts eligiendo antes la version de python 3, Spark 3.5, amazon glue 5.0 y el LabRole.

3.2.2 Configuración del Script de Transformación

Al acabar configurar el entorno Serverless debemos de pegar el archivo python cambiando los valores de nuestra tabla y base de datos a `historicomovilidad` y `vigia_datalake_db` respectivamente. También debemos de cambiar el nombre de la variable S3 para que apunte a nuestra carpeta de analisis (que debemos de crear) dentro de nuestro S3. En esta carpeta es donde se realizarán las particiones y se guardaran los archivos Parquet que genere nuestro job.

Sabremos que nuestro job ha concluido con exito cuando en runs nos salga lo siguiente.

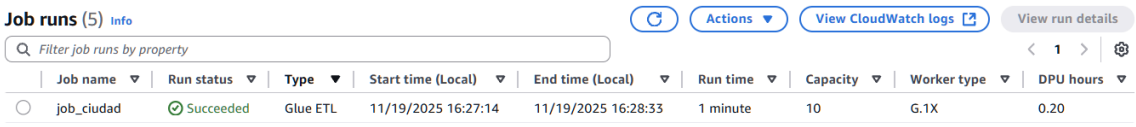


Figura 18: Job run

Dentro de la carpeta `/analisis` de nuestro S3 debemos de comprobar si realmente ha escrito los archivos `Parquet` correspondientes.

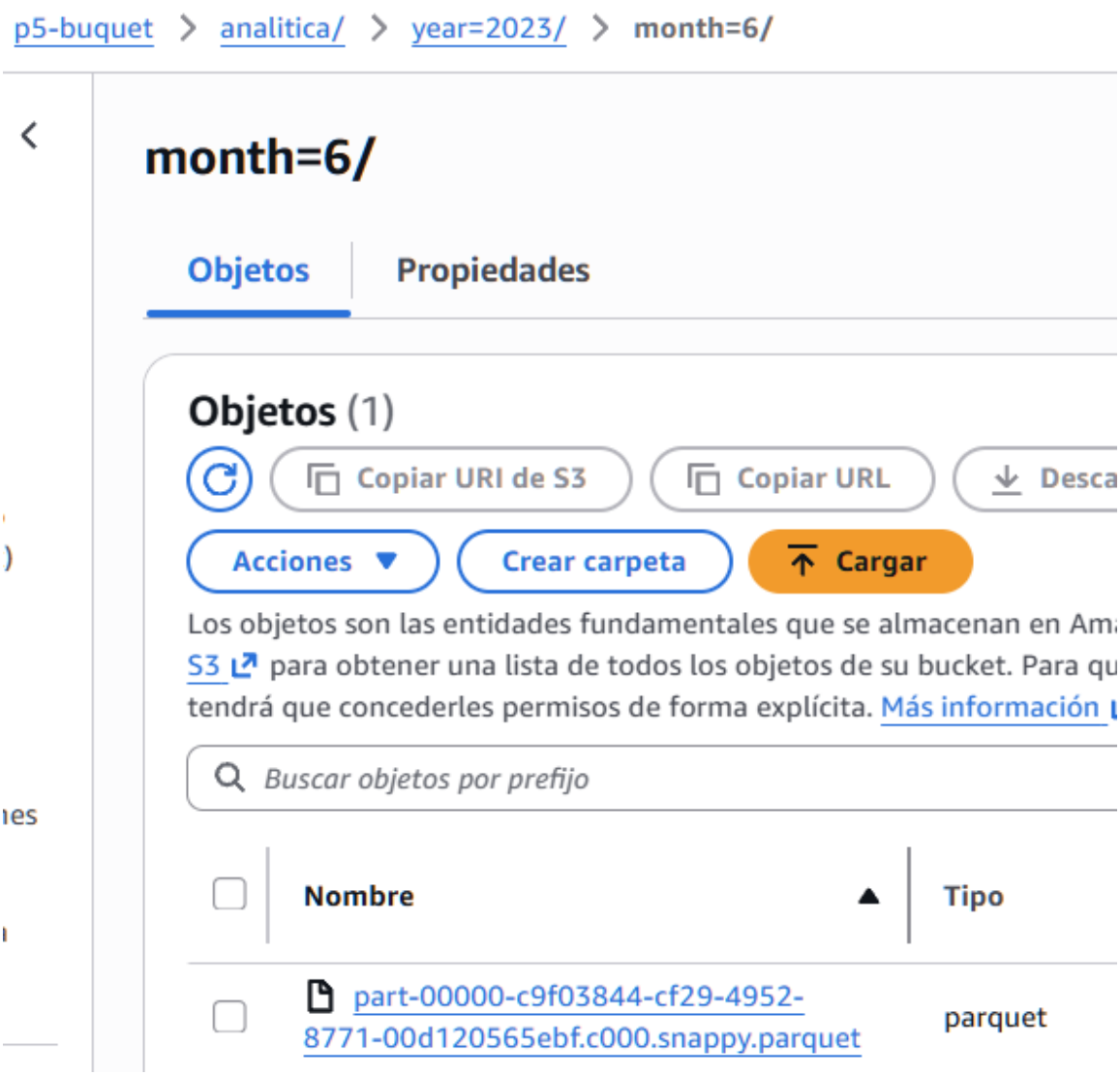


Figura 19: Archivo parket

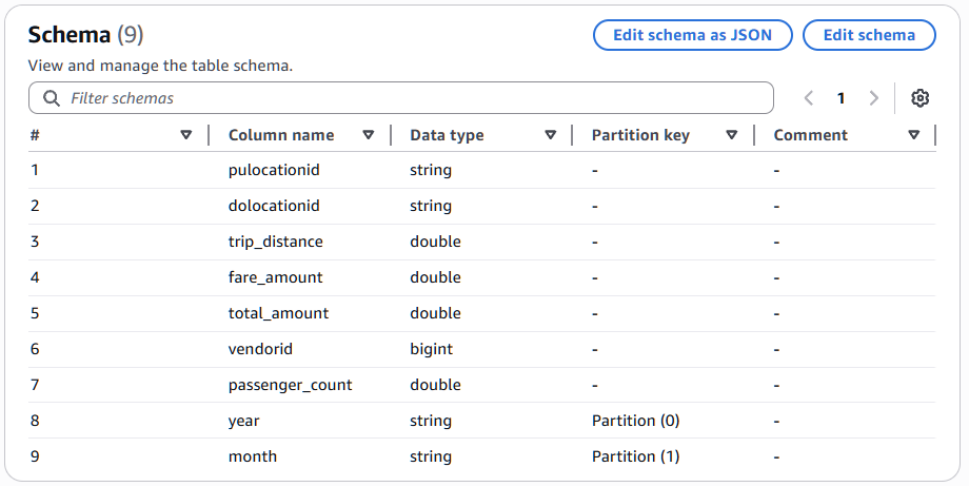
Al inspeccionar la salida en S3, observamos que se ha generado una estructura de carpetas jerárquica correspondiente a las claves de partición definidas en el script (`year` y `month`). En este caso específico, solo vemos la ruta `year=2023/month=6/` conteniendo un único archivo Parquet. Esto se debe a que nuestro dataset de prueba (las 10 filas del CSV) es minúsculo y contiene exclusivamente registros del 30 de junio de 2023.

Es importante aclarar que en un entorno de producción con muchos datos, es normal y deseable encontrar múltiples archivos Parquet dentro de una misma carpeta de mes. Esto ocurre porque Glue procesa los datos en paralelo; cada proceso "worker.[es]cribe su propia parte de los datos simultáneamente para maximizar el rendimiento. Aunque Athena leerá automáticamente todos los archivos que encuentre dentro de la carpeta de la partición como si fueran una única tabla.

3.3 Creación del Crawler PROCESADO

Para finalizar este flujo y poder explotar estos datos optimizados, es necesario catalogar esta nueva estructura en el Glue Data Catalog. Esto se realiza creando y ejecutando un segundo Crawler que apunte específicamente a la carpeta `/analitica`. Una vez completado, tendremos una nueva tabla `analitica_movilidadanalitica` en la base de datos `vigia_datalake_db`. En esta nueva tabla podemos ver como el crawler ha analizado la carpeta `/analitica` y a parte de extraer las columnas con los nuevos

tipos de datos, ha obtenido las particiones de año y mes.



Schema (9)

Edit schema as JSON | Edit schema

View and manage the table schema.

Filter schemas

1

#	Column name	Data type	Partition key	Comment
1	pulocationid	string	-	-
2	dolocationid	string	-	-
3	trip_distance	double	-	-
4	fare_amount	double	-	-
5	total_amount	double	-	-
6	vendorid	bigint	-	-
7	passenger_count	double	-	-
8	year	string	Partition (0)	-
9	month	string	Partition (1)	-

Figura 20: Schema analitica_movilidadanalitica

3.3.1 Configuración de Athena

3.3.2 Consultas SQL sobre los Datos Originales

3.3.3 Consultas SQL sobre los Datos Transformados

3.3.4 Análisis de Resultados y Optimización