Informe:
Práctica 5: Ingesta, almacenamiento y Serveless

Jordi Blasco Lozano
Infraestructuras y Servicios Cloud
Universidad de Alicante

17 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.

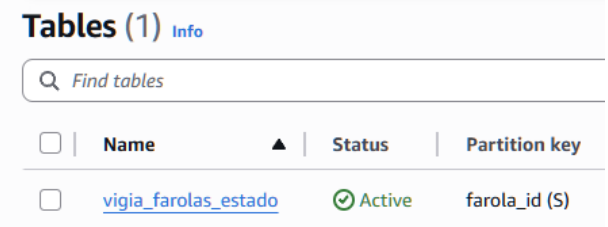

Figura 1: Tablas DynamoDB

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.

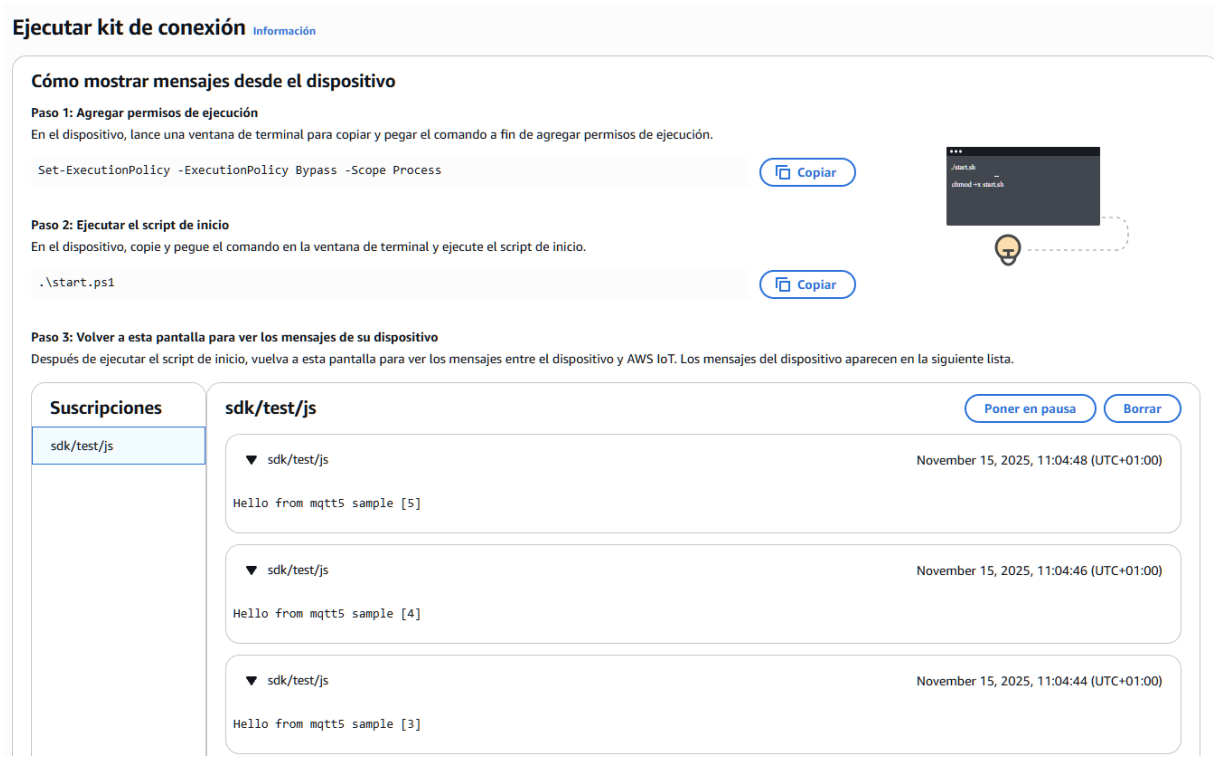

Figura 2: Test start.ps1

- **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.

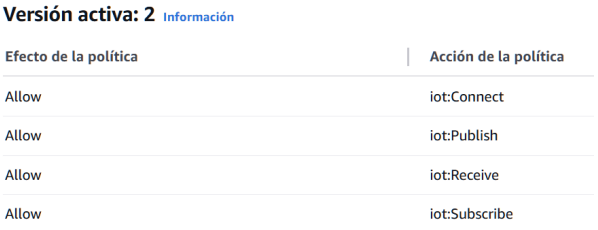

Figura 3: Política activa

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.

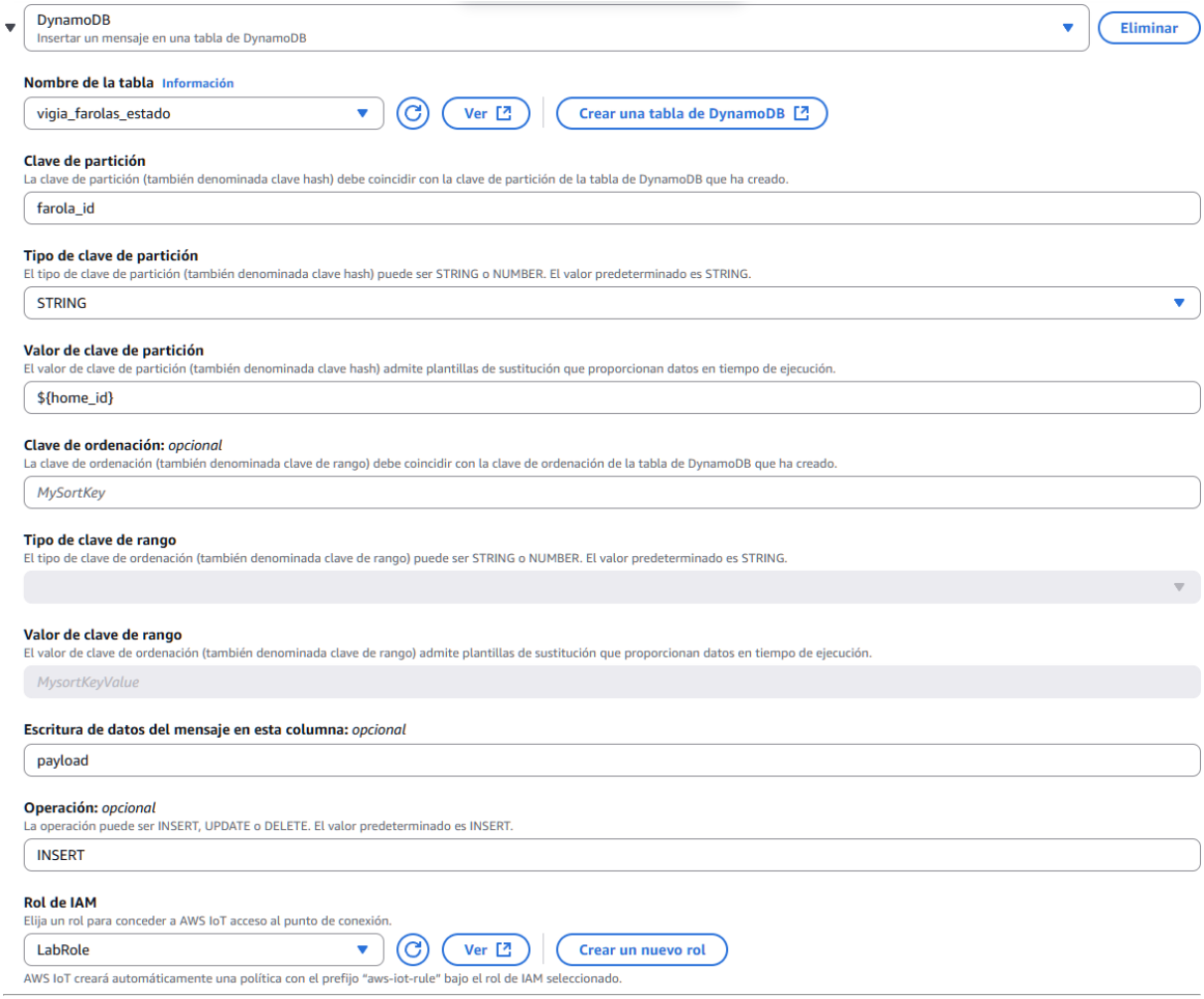

Figura 4: Configuración de la acción DynamoDB

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.

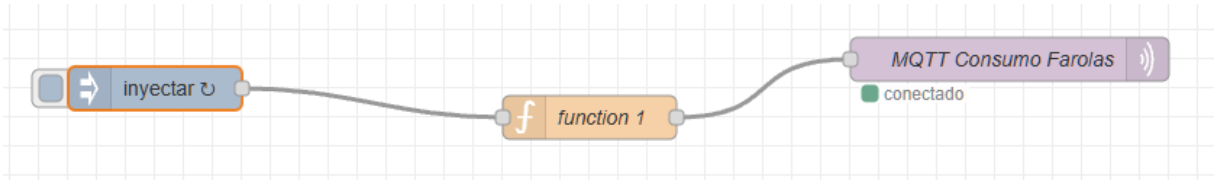

Figura 5: Flujo Node-RED

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.

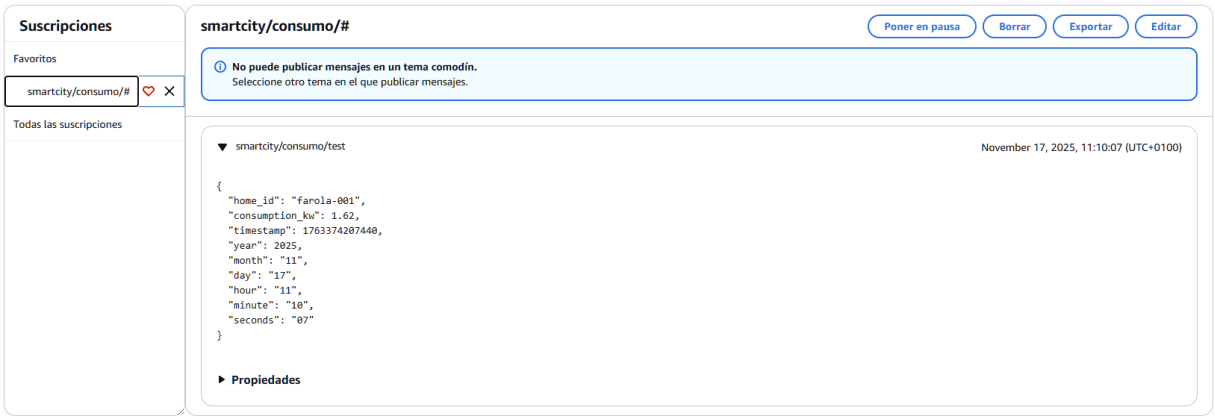

Figura 6: Cliente de prueba de MQTT

- **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.

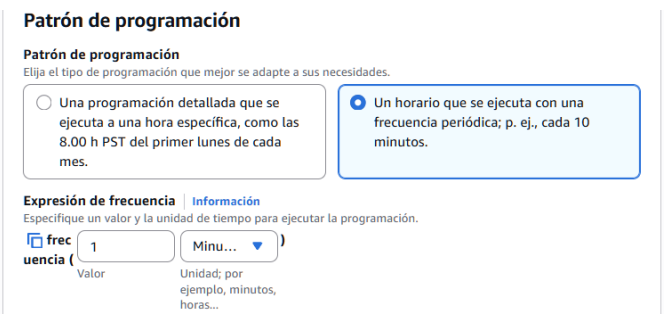

Figura 10: Frecuencia Event Bridge

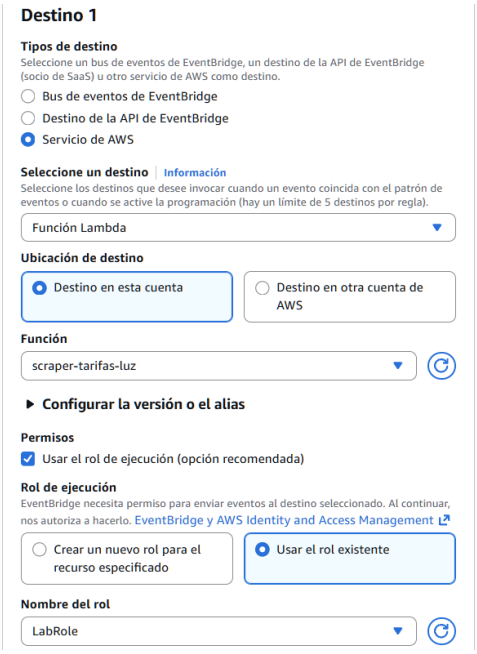

Figura 11: Destino Event Bridge

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.

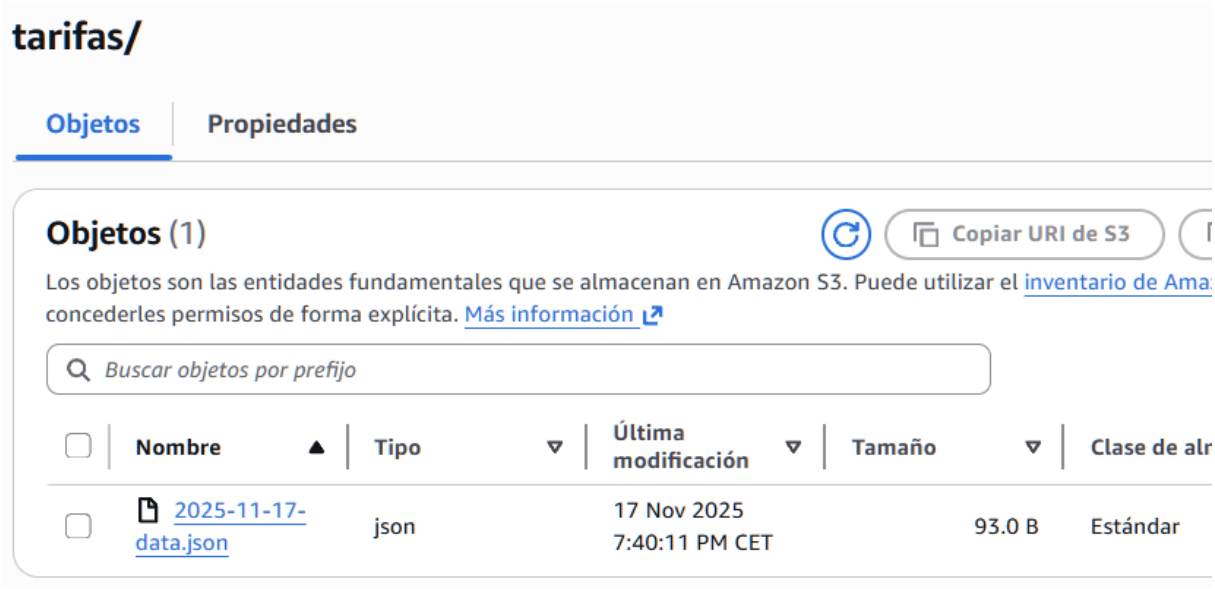

Figura 12: S3 tarifas