Análisis de la Arquitectura de FleetLogix en AWS

Documentación del proceso

Los archivos *aws_setup.py* y *lambda_functions* definen la arquitectura de nube para la plataforma de logística de la empresa Fleetlogix.

aws_setup.py : actúa como el **plano de la infraestructura**, mientras que el lambda_functions.py contiene el **motor de la lógica de negocio**.

A.1) Recursos creados en aws_setup.py funcionalidad en FleetLogix.

Este script se encarga de provisionar la base de la **infraestructura, creando los recursos necesarios en la nube de** n AWS: RDS. Utiliza la librería boto3 de Python para comunicarse con las APIs de AWS. Es un script de **Infraestructura como Código (IaC)**.

1. **Instancia RDS con PostgreSQL (crear_rds_postgresql)**
    - **Recurso Creado**: Una **base de datos relacional** gestionada a través del servicio **Amazon RDS** (Relational Database Service), utilizando el motor PostgreSQL.
    - **Funcionalidad**: Actúa como el **almacén de datos principal y estructurado** de la aplicación. Esto ideal para guardar información histórica, registros de entregas finalizadas, datos de clientes, facturas, y cualquier otra información que requiera consistencia y relaciones complejas.
    - La configuración db.t3.micro la mantiene en la capa gratuita, y la accesibilidad pública (PubliclyAccessible=True) facilita la migración inicial desde un entorno local.
2. **Bucket en S3 (crear_s3_bucket)**
    - **Recurso Creado**: Un **Amazon S3 Bucket**, que es un servicio de **almacenamiento** de objetos altamente escalable y duradero.
    - **Funcionalidad**: Funciona como un **Data Lake** o repositorio central para datos no estructurados o semi-estructurados. En este caso, se utiliza para:
        - **raw-data/**: Almacenar datos crudos de GPS u otros sensores antes de ser procesados.

        - **processed-data/**: Guardar datos ya limpios y transformados, listos para análisis.
        - **backups/**: Guardar copias de seguridad, por ejemplo, de la base de datos RDS.
        - **logs/**: Centralizar los registros de las aplicaciones y funciones Lambda.
    - Una característica clave es la **política de ciclo de vida** que **mueve los datos de raw-data/ a Amazon Glacier** después de 90 días, optimizando costos al archivar datos antiguos que no se necesitan con frecuencia.
3. **Tablas en DynamoDB (crear_tablas_dynamodb)**
    - **Recursos Creados**: Cuatro tablas en **Amazon DynamoDB**, una base de datos NoSQL gestionada.
    - **Funcionalidad**: Diseñada para acceso de **alta velocidad y baja latencia**, es ideal para almacenar el **estado actual y en tiempo real** del sistema.
        - deliveries_status: Consulta rápida del estado de una entrega (en_camino, entregado).
        - vehicle_tracking: Almacena la última ubicación conocida de un vehículo. Su clave de ordenación (timestamp) permite consultar el historial de ubicaciones de un vehículo específico.
        - routes_waypoints: Guarda los puntos que componen una ruta planificada.
        - alerts_history: Un registro de todas las alertas generadas (desvíos, excesos de velocidad, etc.).
    - El modo de facturación PAY_PER_REQUEST es excelente para cargas de trabajo variables, ya que no se paga por capacidad aprovisionada, sino por uso real.
4. **Rol de IAM (crear_rol_iam_lambda)**
    - **Recurso Creado**: Un **rol de IAM (Identity and Access Management)**: FleetLogixLambdaRole.
    - **Funcionalidad**: : Este es un componente de seguridad fundamental. Es un conjunto de **permisos** que se asigna a las funciones Lambda.

En lugar de guardar credenciales de AWS en el código (lo cual sabemos que es una mala práctica), el rol le otorga a la función Lambda la autoridad para interactuar con otros servicios de AWSm específicamente a DynamoDB, S3 y SNS (servicio de notificaciones para mensajería en tiempo real), además del permiso básico para escribir logs.

B. Funciones *lambda_functions.py:*

Contiene la **lógica de negocio** de la aplicación. Son funciones serverless (se ejecutan sin servidor) y **responden a eventos** específicos, como una solicitud de un usuario o una actualización de datos.

Este archivo no *crea* recursos (infraestructura) sino que define el código que se ejecutará dentro de los recursos **AWS Lambda** que serían desplegados posteriormente.

1. **lambda_verificar_entrega**: Una función que actúa como un endpoint de API para consultar el estado de una entrega.
2. **lambda_calcular_eta**: Un procesador de datos que calcula el tiempo estimado de llegada (ETA) de un vehículo y actualiza su última ubicación conocida.
3. **lambda_alerta_desvio**: Un sistema de monitoreo en tiempo real que detecta si un vehículo se ha desviado de su ruta planificada y, de ser así, envía una notificación.

A. Profundización en la función lambda_verificar_entrega

Esta función es un ejemplo clásico de un microservicio serverless diseñado para una tarea específica. Su flujo de ejecución es el siguiente:

1. **Propósito y Disparador (Trigger)**: Está diseñada para ser activada por **Amazon API Gateway**. Esto la convierte en un endpoint HTTP(S) que puede ser llamado desde una aplicación móvil, una página web o cualquier otro servicio. Su objetivo es responder a la pregunta: "¿Se ha completado la entrega con ID X?"

2. **Entrada de Datos (event): Recepción y validación:**

La función recibe los datos de la solicitud en el parámetro evento. que contiene el delivery_id. Su primera acción es validar que el delivery_id necesario para la consulta esté presente. Si no lo está, devuelve inmediatamente un error 400 Bad Request, ahorrando procesamiento innecesario.

3. **Interacción con la Base de Datos(Consulta a DynamoDB)**

   Se conecta a la tabla deliveries_status y utiliza el método get_item. Esta es la operación más eficiente posible en DynamoDB, ya que busca un ítem directamente por su clave primaria. El resultado es casi instantáneo, incluso con miles de millones de registros.

4. **Lógica y Transformación** :

   - Si la base de datos devuelve un Item, la función comprueba si el valor del atributo status es 'delivered'. Esta simple comparación define el valor booleano de is_completed.
   - Si no se encuentra ningún Item, la función asume que la entrega no existe y prepara una respuesta de error 404 Not Found.
   - **Respuesta**: Finalmente, empaqueta el resultado en la estructura de respuesta que espera API Gateway, con un statusCode (200, 404, 500) y un body en formato JSON

5. **Manejo de Errores**: Todo el proceso está envuelto en un bloque try...except. Si ocurre cualquier otro error (problemas de conexión con DynamoDB, falta de permisos, etc.), se captura y se devuelve un error genérico 500 (Internal Server Error), evitando exponer detalles internos del sistema.

En resumen, esta función es un **backend** eficiente y escalable para una consulta muy común en un sistema de logística, aprovechando la **velocidad de DynamoDB** para dar respuestas instantáneas.

C. Profundización en la función migrar_datos_postgresql()

1. **Propósito**: Facilitar el proceso de mover los datos desde una base de datos PostgreSQL que se encuentra en un entorno de desarrollo local hacia la nueva instancia de base de datos RDS recién creada en AWS.
   - **Dependencia temporal**: La migración solo puede ocurrir *después* de que la instancia RDS esté completamente creada y disponible,

por eso se debe esperar 10 minutos luego de la creación de los servicios.

- **Credenciales y Endpoints**: El script de migración necesita el "endpoint" (la URL) de la nueva base de datos RDS, que solo se conoce una vez que ha sido creada. El desarrollador necesita copiar esta URL y pegarla en el script generado.

2. **Desglose del Script migrate_to_rds.sh generado**:
    - **#!/bin/bash**: Indica que es un script de shell para sistemas tipo Unix (Linux, macOS).
    - **Variables**: Define placeholders para la configuración local y remota (nombres de bases de datos, usuarios y el RDS_ENDPOINT).
    - **Paso 1: pg_dump**: Para **exportar** la base de datos completa de PostgreSQL. Crea un archivo de texto (fleetlogix_dump.sql) que contiene todos los comandos SQL necesarios para recrear la estructura (tablas, vistas) y los datos (filas) de la base de datos local.
    - **Paso 2: psql ... -c "CREATE DATABASE ..."**: Se conecta a la instancia de RDS (usando el endpoint) y ejecuta un único comando SQL para **crear una base de datos vacía** con el nombre fleetlogix. Es un prerrequisito, ya que no se puede importar un volcado a una base de datos que no existe.
    - **Paso 3: psql ... -f fleetlogix_dump.sql**: Este es el comando de **importación**. Se conecta a la base de datos recién creada en RDS y ejecuta todos los comandos contenidos en el archivo fleetlogix_dump.sql, restaurando efectivamente la base de datos local en la nube.

En conclusión, esta función es una utilidad muy práctica que prepara y documenta el proceso manual de migración de datos, siguiendo las mejores prácticas al separar las tareas de aprovisionamiento de infraestructura de las tareas de gestión de datos.

Diagrama de Arquitectura y Flujo de Datos de FleetLogix

El diagrama adjunto visualiza cómo los componentes de AWS interactúan, desde la solicitud inicial hasta el almacenamiento de datos a largo plazo y la generación de alertas.

Descripción del Flujo y los Componentes del Diagrama

1. **Entrada de Datos y Solicitudes de Usuario**:
    - **Usuario / Aplicación Móvil**: Punto de partida. Realiza una solicitud, como verificar-entrega, a través de internet.
    - **API Gateway**: Actúa como el front-end seguro para las funciones Lambda. Recibe las solicitudes HTTP y las dirige a la función correcta. (Aunque no se crea en el script aws_setup.py, se menciona como el trigger lógico para la función verificar-entrega).
2. **Capa de Cómputo (Serverless)**:
    - **AWS Lambda**: Ejecuta el código de lambda_functions.py sin necesidad de gestionar servidores.
    - **IAM Role (FleetLogixLambdaRole)**: Es el componente de seguridad que envuelve a Lambda. Le otorga los permisos necesarios y explícitos para interactuar con otros servicios de AWS, como se ve con las flechas que salen de Lambda.
3. **Bases de Datos (Optimización por Caso de Uso)**:
    - **Amazon DynamoDB**: La base de datos para operaciones en **tiempo real**. Las Lambdas leen y escriben aquí para obtener respuestas de baja latencia (ej. consultar deliveries_status). La arquitectura aprovecha el modelo de facturación **Pay Per Request**, lo que significa que solo se paga por las lecturas y escrituras que se realizan, siendo extremadamente eficiente para cargas de trabajo variables.
    - **Amazon RDS (PostgreSQL)**: El sistema para **datos relacionales y transaccionales**. Almacena el histórico de datos y la información estructurada. La migración inicial de datos se realiza desde una base de datos local, como se indica en el script migrate_to_rds.sh generado.
4. **Sistema de Notificaciones**:

- **Amazon SNS (Simple Notification Service)**: Cuando una Lambda (como alerta-desvio) detecta un evento importante, publica un mensaje en un tema de SNS. Esto desacopla la lógica de la notificación.
- **Supervisores / Sistemas Externos**: Se suscriben al tema de SNS para recibir las alertas de forma inmediata (por email, SMS, etc.).

5. **Almacenamiento y Archivo de Datos (Data Lake)**:
    - **Amazon S3 Bucket**: Es el repositorio central para todo tipo de datos (raw-data, logs, backups). Es altamente durable y escalable.
    - **Política de Ciclo de Vida:** después de 90 días, los datos en la carpeta raw-data/ son movidos automáticamente de S3 a un almacenamiento de archivo de muy bajo costo.
    - **Amazon S3 Glacier**: Es el destino final para el archivo de datos a largo plazo. Acceder a estos datos es más lento, pero su costo de almacenamiento es significativamente menor, optimizando el presupuesto.