Avance 4: Arquitectura en la nube con ADW

a. Estructura general del proyecto:

El proyecto FleetLogix usa AWS para manejar datos de entregas provenientes de apps móviles de conductores. La arquitectura combina servicios serverless (sin servidores) y bases gestionadas.

El Diagrama se puede dividir en 4 capas para que se vea ordenado:

1. Capa de Entrada (Frontend / Ingreso de datos): 📱 Aplicación móvil de conductores.

Envía eventos como:

- “Entrega completada”
- “Posición GPS actual”
- “Inicio de viaje”

Todos estos mensajes entran al sistema a través de:

- Amazon API Gateway
- Expone endpoints HTTP (por ejemplo /deliveries/verify).
- Recibe los requests y los pasa a las funciones Lambda correspondientes.

2. Capa de Procesamiento (Lógica de negocio)

AWS Lambda (3 funciones):

a. lambda_verificar_entrega: Consulta DynamoDB para ver si una entrega se completó.
Trigger: API Gateway

b. lambda_calcular_eta: Calcula ETA usando coordenadas y velocidad.
Trigger: EventBridge (cada 5 min)

c. lambda_alerta_desvio: Detecta desvíos de ruta y envía alertas.
   Trigger: Kinesis (datos de GPS en tiempo real)

3. Capa de Almacenamiento

Amazon DynamoDB

- deliveries_status: estado de entregas
- vehicle_tracking: seguimiento y ETA
- routes_waypoints: rutas esperadas
- alerts_history: alertas generadas

Amazon S3

- Guarda los datos históricos exportados (copias de seguridad, reportes, logs, etc.)

Opcionalmente, si se agregan análisis después, se podrían incluir Snowflake o Redshift como capa analítica

4. Capa de Notificaciones y Automatización

Amazon SNS (Simple Notification Service)

- Envía alertas a supervisores o sistemas externos si hay desvíos.

Amazon EventBridge

- Programa tareas automáticas (como el cálculo ETA cada 5 minutos).

Amazon Kinesis

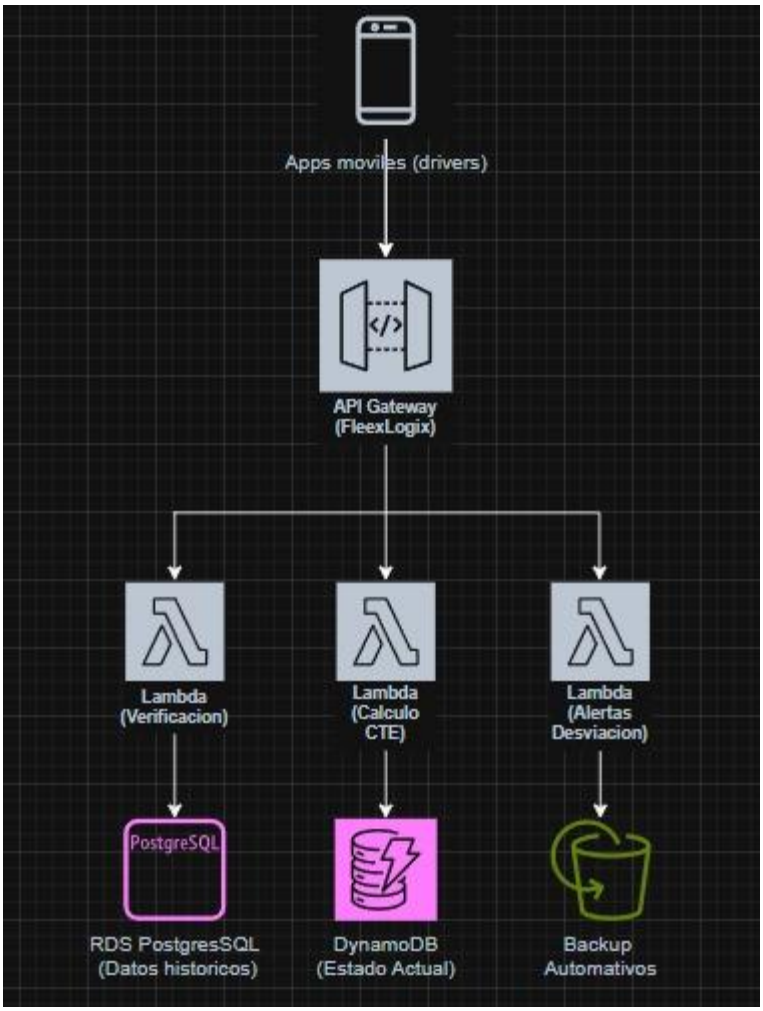
- Recibe el flujo de datos GPS en tiempo real.

Flujo de datos resumido

1) El conductor marca entrega → App móvil → API Gateway → Lambda 1 → DynamoDB

2) Cada 5 min → EventBridge ejecuta Lambda 2 → calcula ETA → guarda tracking
3) Cada actualización GPS → Kinesis → Lambda 3 → analiza desvío → SNS + DynamoDB
4) Todos los datos diarios se guardan en S3 (histórico o backups)

Diagrama:



1. **Archivo aws_setup.py:**

Este script automatiza la creación de los servicios base de AWS necesarios para el proyecto FleetLogix, usando la librería boto3 (el SDK oficial de AWS para Python). En otras palabras: en lugar de crear manualmente los recursos en la consola web de AWS, este script los crea programáticamente.

2. **Recursos creados y función de cada uno**

Recurso	Servicio AWS	Qué hace	Función en FleetLogix
RDS PostgreSQL	Amazon RDS	Crea una base de datos PostgreSQL administrada	Guarda la información transaccional y relacional (viajes, vehículos, entregas, conductores, etc.)
S3 Bucket	Amazon S3	Crea un bucket con subcarpetas y reglas de ciclo de vida	Almacena los datos históricos de las entregas, respaldos y logs
DynamoDB (4 tablas)	Amazon DynamoDB	Crea tablas NoSQL	Guarda datos operativos y en tiempo real (tracking, alertas, estados)

IAM Role para Lambda	AWS IAM	Crea un rol con permisos específicos	Permite que las funciones Lambda puedan leer/escribir en S3, DynamoDB y enviar notificaciones
Backup Snapshot RDS	Amazon RDS	Crea snapshot inicial y configura backups automáticos	Asegura disponibilidad y recuperación ante fallos
Script de migración	Shell script (migrate_to_rds.sh)	Crea un archivo para migrar datos desde una base local a RDS	Automatiza el traspaso de la base de datos local al entorno AWS

3. Explicación de funciones clave

a. crear_rds_postgresql(): tener una base de datos relacional persistente y escalable.

- Crea una instancia de **Amazon RDS** con el motor **PostgreSQL 15.4**, usando la clase gratuita db.t3.micro.
  Incluye configuraciones de:
- **Backups automáticos (7 días)**
- **Ventanas de mantenimiento**
- **Accesibilidad pública** (para pruebas)
- **Etiquetas (tags)** para identificar el entorno

b. crear_s3_bucket(): Almacena de forma segura y organizada todos los datos históricos del proyecto.

Crea el bucket fleetlogix-data y dentro de él genera la siguiente estructura:

raw-data/

processed-data/

backups/

logs/

También define una política de ciclo de vida (Lifecycle Configuration): Después de 90 días, los archivos de raw-data/ se mueven automáticamente al almacenamiento Glacier, más barato, para archivo histórico.

c. **crear_tablas_dynamodb():** Crea cuatro tablas NoSQL de acceso rápido, para manejar la parte en tiempo real del sistema (tracking, alertas, actualizaciones).

Tabla	Clave primaria	Qué almacena
deliveries_status	delivery_id	Estado actual de cada entrega
vehicle_tracking	vehicle_id + timestamp	Posición geográfica del vehículo en tiempo real
routes_waypoints	route_id	Puntos de la ruta o camino a seguir
alerts_history	vehicle_id + timestamp	Historial de alertas y desviaciones

d. configurar_backups_automaticos(): garantiza la recuperación ante desastres. Crea un snapshot inicial de la base de datos RDS y confirma la política de backups automáticos (ya definida en la función anterior).

e. **migrar_datos_postgresql():** mueve tus datos locales (por ejemplo, de localhost:5432) al nuevo entorno cloud sin perder nada.
Genera un script de bash (migrate_to_rds.sh) con los pasos para migrar una base PostgreSQL local al RDS creado.
El script:

- Hace un pg_dump de la base local
- Crea la base vacía en RDS
- Restaura los datos en RDS con psql

f. **crear_rol_iam_lambda()** Permite que las funciones Lambda se integren con todos los servicios necesarios.
   Crea un rol de permisos llamado **FleetLogixLambdaRole** con las siguientes políticas:
   - **AWSLambdaBasicExecutionRole**: Permite ejecutar Lambdas y escribir logs en CloudWatch
   - **AmazonDynamoDBFullAccess**: Acceso a las tablas operativas
   - **AmazonS3FullAccess**: Leer y escribir datos en el bucket.
   - **AmazonSNSFullAccess**: Enviar notificaciones (por ejemplo, alertas de desvío)

g. **main()**
   Ejecuta todo el flujo completo:
   1. Crea RDS
   2. Crea S3
   3. Crea DynamoDB
   4. Configura backups
   5. Crea el script de migración
   6. Crea el rol IAM
   7. Guarda todo en aws_config.json

Cómo se vincula esto con las funciones Lambda

Este aws_setup.py solo prepara la infraestructura.
Las funciones Lambda (por ejemplo verificar-entrega, calcular-eta, alerta-desvio) se crean aparte, y usan los recursos definidos aquí:

- Guardan entregas procesadas en DynamoDB
- Escriben registros históricos en S3
- Se autentican mediante el rol IAM creado
- Envían notificaciones o alertas mediante SNS

Funciones Lambda: Representan el “cerebro” del flujo operativo de FleetLogix en AWS.

Explicación de la función lambda_verificar_entrega:

Validar si una entrega fue completada, consultando la tabla deliveries_status en DynamoDB. Esta Lambda actúa como intermediario entre el sistema móvil y la base de datos, asegurando que las apps móviles obtengan la información correcta sin acceder directamente al almacenamiento.

Flujo paso a paso:

1. Recibe un evento desde la app móvil a través del API Gateway, con el delivery_id y opcionalmente tracking_number.
2. Valida que haya un delivery_id, si no lo hay, devuelve un error 400.
3. Consulta DynamoDB (deliveries_status) buscando esa entrega.
4. Si existe:
    - Revisa si status es "delivered".
    - Devuelve JSON con el estado actual, fecha de entrega y si está completada.

    Si no existe:
    - Devuelve error 404 (“Entrega no encontrada”).

    Si ocurre algún fallo de conexión o lectura:
    - Devuelve error 500 con el detalle del error.

Explicación de lambda_calcular_eta:

Esta Lambda se ejecuta automáticamente cada 5 minutos (EventBridge rule), ayudando a actualizar el estado en tiempo real de todos los camiones.

Flujo paso a paso:

1. Recibe desde EventBridge o una app los datos:
2. Calcula la **distancia restante** usando una fórmula simple de Haversine (aprox. 111 km por grado).
3. Divide la distancia por la velocidad actual: obtiene **horas restantes**.
4. Calcula el ETA (datetime.now() + timedelta(hours)).
5. Guarda en la tabla vehicle_tracking de DynamoDB:

1. Ubicación actual y destino
2. Distancia restante
3. ETA
4. Velocidad actual

6. Devuelve una respuesta con la distancia y ETA estimada.

Explicación de lambda_alerta_desvio: Detecta si un vehículo se desvía de su ruta predefinida y enviar una alerta.

Flujo paso a paso:

1. Recibe evento.
2. Busca en DynamoDB (routes_waypoints) los puntos de ruta esperada.
3. Calcula la distancia mínima entre el vehículo y la ruta.
4. Si se aleja más de 5 km del recorrido entonces considera que hay un desvío.
5. En caso de desvío:
    - Envía un mensaje al SNS topic fleetlogix-alerts.
    - Guarda el evento en la tabla alerts_history para registro.
6. Devuelve si hubo o no desvío, y a qué distancia.