AWS Análisis Arquitectura - AVANCE 4

Proyecto: FleetLogix – Sistema de Gestión de Transporte y Logística
Autor: Cristian García
Repositorio: https://github.com/cfgarciac/Module-2
Versión del documento: v1.0
Fecha: 13/11/2025

1. INTRODUCCIÓN Y CONTEXTO DEL PROYECTO

FleetLogix es una empresa de logística que ya cuenta con una base de datos transaccional en PostgreSQL y un conjunto de consultas SQL optimizadas que responden preguntas operativas del día a día. En los avances anteriores se diseñó un modelo dimensional en Snowflake y se construyó un pipeline ETL en Python que permite analizar de forma histórica el desempeño de las entregas: tiempos de entrega, retrasos, uso de combustible, ingresos estimados y eficiencia de rutas y conductores.

Sin embargo, para competir en un entorno cada vez más dinámico, FleetLogix necesita dar el siguiente paso: pasar de un análisis principalmente histórico a una arquitectura capaz de recibir, procesar y reaccionar a los datos de la flota en tiempo casi real, aprovechando servicios en la nube.

En este contexto, el Avance 4 se centra en diseñar (no implementar físicamente) una arquitectura serverless en AWS que permita:

- Recibir eventos de las aplicaciones móviles de los conductores mediante API Gateway.
- Procesar esos eventos con AWS Lambda, calculando métricas inmediatas como tiempo estimado de llegada (ETA), detección de entregas completadas y desvíos de ruta.
- Almacenar los datos históricos en Amazon S3, organizados por fecha, para análisis posteriores.
- Utilizar AWS RDS para alojar la base transaccional en la nube y DynamoDB para mantener el estado actual de las entregas.

De esta forma, la empresa pasa de una arquitectura puramente on-premise a una solución híbrida y preparada para tiempo real, que complementa el Data Warehouse ya construido en Snowflake.

2. OBJETIVOS

En este avance, el objetivo no es desplegar realmente todos los servicios en AWS, sino diseñar y documentar una arquitectura coherente, escalable y alineada con las necesidades del negocio de FleetLogix.

Los objetivos específicos son:

2.1. Objetivos de negocio

1. Habilitar el monitoreo casi en tiempo real de la operación de entrega
¿Por qué?
La dirección necesita saber no solo qué ocurrió en el pasado (histórico en Snowflake), sino también qué está ocurriendo ahora: entregas en curso, retrasos potenciales, desvíos de ruta y estimación de llegada a clientes.
¿Para qué?
Para tomar decisiones operativas rápidas: reasignar rutas, avisar al cliente de retrasos, investigar desvíos anómalos, etc.

2. Guardar un histórico completo de eventos de la flota en la nube
¿Por qué?
Los dispositivos móviles de los conductores pueden generar muchos eventos (ubicaciones, cambios de estado, confirmaciones). Estos datos no deben perderse, ya que alimentan tanto analíticas futuras como modelos de machine learning.
¿Para qué?
Para seguir enriqueciendo el Data Warehouse (Snowflake) con eventos a nivel granular y construir, a futuro, modelos de predicción de retrasos, optimización de rutas, etc.

3. Reducir la dependencia de infraestructura local
¿Por qué?
Mantener la base PostgreSQL únicamente en un servidor local limita la escalabilidad, la disponibilidad y la resiliencia de la solución.
¿Para qué?
Para que la plataforma pueda crecer bajo demanda, operar de forma más confiable y simplificar la administración de bases de datos mediante un servicio administrado como AWS RDS.

4. Disponer de un “estado actual” de la operación fácilmente consultable
¿Por qué?
La gerencia o un dashboard operativo necesitan ver rápidamente cuántas entregas están “EN CURSO”, “COMPLETADAS”, “RETENIDAS”, etc.

¿Para qué?
Para habilitar dashboards en tiempo real y alertas que muestren el pulso de la operación sin tener que recorrer tablas históricas muy grandes.

2.2. Objetivos técnicos

4.1. Definir una arquitectura serverless basada en servicios de AWS

- Uso de API Gateway como punto de entrada HTTP.
- Uso de AWS Lambda como capa de lógica de negocio disparada por eventos.
- Uso de Amazon S3 como almacenamiento de datos históricos (data lake).
- Uso de AWS RDS (PostgreSQL) para la base transaccional.
- Uso de Amazon DynamoDB como base NoSQL para estado actual de entregas.

4.2. Definir el diseño de 3 funciones Lambda clave

- Verificar si una entrega se completó.
- Calcular el tiempo estimado de llegada (ETA).
- Enviar una alerta cuando el camión se desvíe de la ruta esperada.

Para cada función se explicará:

- Trigger (cómo se activa).
- Entradas y salidas esperadas.
- Qué lógica implementa y con qué propósito de negocio.

4.3. Diseñar el flujo de ingesta y almacenamiento en S3 organizado por fecha

- Justificar la estructura de carpetas por fecha (por ejemplo year=/month=/day=) para facilitar la partición de datos en futuros procesos analíticos.
- Explicar cómo se integraría este “data lake” con el Data Warehouse en Snowflake.

4.4. Definir el esquema de bases de datos en la nube y backups automáticos

- Justificar la migración de PostgreSQL local a AWS RDS.
- Explicar el uso de backups automáticos (snapshots) y sus beneficios.
- Describir cómo DynamoDB se complementa con RDS y Snowflake.

4.5. Diseñar la API (Extra credit)

- Proponer 3 rutas en API Gateway, por ejemplo:
  POST /deliveries/update-status

GET /deliveries/{id}/eta
POST /deliveries/route-deviation

- Explicar cómo cada ruta se conecta con su Lambda correspondiente.
- Mostrar cómo la URL pública del API podría ser consumida desde la app móvil.

4.6. Preparar insumos claros para el diagrama de arquitectura AWS

- Que el diagrama aws_architecture_diagram.png muestre con claridad:
  Origen: app móvil del conductor.
  Entrada: API Gateway.
  Capa de lógica: Lambdas.
  Persistencia: S3, RDS, DynamoDB.
  Integraciones con Snowflake y herramientas de analítica.

3. VISIÓN GENERAL DE LA ARQUITECTURA EN AWS

3.1. Descripción de alto nivel del flujo de datos

En la propuesta de arquitectura, el flujo de datos sería el siguiente:

1. El conductor, desde una app móvil, envía eventos de la entrega (inicio de viaje, actualización de posición GPS, entrega completada, incidencias, etc.).
2. La app se comunica con un API Gateway público que expone una API REST con rutas como /deliveries, /eta y /alerts.
3. API Gateway invoca funciones AWS Lambda, que contienen la lógica de negocio:
   - Validar el payload.
   - Normalizar y enriquecer los datos.
   - Determinar si la entrega se completó.
   - Calcular ETA.
   - Verificar si hay desvíos de la ruta esperada.
4. Las Lambdas realizan dos tipos de escritura:
   - Almacenan el histórico “raw” en S3, en carpetas particionadas por fecha (s3://fleetlogix-raw/year=YYYY/month=MM/day=DD/...).
   - Actualizan el estado actual de la entrega en DynamoDB, para acceso rápido desde aplicaciones de monitoreo o dashboards operativos.
5. Periódicamente (por ejemplo, cada noche), se ejecutan procesos de integración que:
   - Extraen datos históricos desde S3 o desde RDS PostgreSQL.
   - Los integran con el Data Warehouse en Snowflake, alimentando la tabla de hechos fact_deliveries y sus dimensiones.

6. La dirección puede consultar:
    - Información operativa en tiempo casi real a través de DynamoDB / API.
    - Información histórica consolidada a través de Snowflake, con dashboards y reportes de largo plazo.

3.2. Relación con el Data Warehouse en Snowflake (Avance 3)

La arquitectura en AWS no reemplaza al Data Warehouse en Snowflake; lo complementa:

- Snowflake sigue siendo el repositorio central para análisis histórico, KPIs, tendencias por año, trimestre, región, vehículo, conductor, etc.
- AWS proporciona:
  La capa de ingesta en tiempo real (API + Lambda).
  El almacenamiento “raw” (S3) como data lake.
  El estado actual de las entregas (DynamoDB).
  La base operacional administrada (RDS).

En términos de flujo:

1. Los datos nuevos nacen en API Gateway → Lambda → S3/DynamoDB.
2. Procesos posteriores (batch o micro-batch) integran esa información con el Data Warehouse en Snowflake, manteniendo la tabla fact_deliveries actualizada.

Esto permite a FleetLogix cubrir dos horizontes de análisis:

- Tiempo real / corto plazo: monitoreo de entregas activas, alertas, ETA (AWS).
- Histórico / largo plazo: análisis de desempeño, optimización de rutas, evaluación por cliente, conductor o vehículo (Snowflake).

4. PARTE A – INGESTA Y ALMACENAMIENTO DE DATOS

4.1. API Gateway como puerta de entrada

¿Por qué API Gateway?

- Porque permite definir un punto único de entrada para todas las apps móviles de la flota.
- Gestiona autenticación, autorización, limitación de peticiones (rate limiting) y generación de logs.
- Se integra de forma nativa con Lambda, IAM y otros servicios de AWS.

¿Para qué?

- Para recibir de manera estandarizada los eventos de entrega, sin que la app tenga que conectarse directamente a la base de datos.

- Para desacoplar el front (app móvil) del backend, facilitando la evolución de la arquitectura sin modificar los clientes.

A nivel conceptual, se define una API REST con rutas como:

- POST /deliveries/events → recepciona eventos (inicio de viaje, avance GPS, entrega completada).
- GET /deliveries/{delivery_id}/status → consulta del estado actual de una entrega.
- POST /deliveries/eta → cálculo de ETA (para conductor o cliente).
- POST /deliveries/route-deviation → notificaciones de posición para verificar desvíos.

Cada ruta se asocia a una integración Lambda.

4.2. Función Lambda de recepción y validación

La primera Lambda de la cadena tiene responsabilidades muy claras:

- Validar el payload recibido:
  Campos obligatorios: delivery_id, driver_id, vehicle_id, timestamp, location, status, etc.
  Reglas básicas de negocio (por ejemplo, tipos de estado válidos: IN_TRANSIT, DELIVERED, DELAYED).
- Enriquecer datos cuando sea posible (por ejemplo, normalizar formato de fechas, traducir códigos de estado legados a estados estándar).
- Registrar el evento en S3 para histórico y, opcionalmente, en DynamoDB si se trata de un estado final.

Desde el punto de vista técnico, Lambda recibe el evento de API Gateway, lo procesa y decide:

- Guardar un archivo JSON en S3 (una línea por evento o archivos pequeños por ventana de tiempo).
- Emitir un mensaje a otra Lambda (por ejemplo, vía SNS o EventBridge) para el cálculo de ETA o desvíos.x

4.3. Almacenamiento histórico en S3 por fecha (data lake “raw”)

¿Por qué S3?

- Es un almacenamiento altamente escalable y barato.
- Ideal para guardar históricos “raw”, sin transformar, que luego pueden ser usados por múltiples consumidores (ETL, Data Science, auditorías).

Organización propuesta:
s3://fleetlogix-raw/
  year=2025/
    month=11/
      day=13/
        deliveries_events_2025-11-13-00.json
        deliveries_events_2025-11-13-01.json
        ...

- Cada archivo puede contener múltiples eventos (por ejemplo, en formato JSON Lines).
- La partición por año/mes/día facilita:
  Filtrado eficiente para procesos batch.
  Integración futura con herramientas tipo Athena, Glue, o procesos ETL hacia Snowflake.

Desde el punto de vista de negocio, esto garantiza que ningún evento se pierde: aunque haya errores posteriores en otras capas, el histórico bruto está resguardado.

5. PARTE B – PROCESAMIENTO Y ANALÍTICA CASI EN TIEMPO REAL

5.1. Lambda: verificación de entrega completada

Esta función Lambda responde a preguntas como:

- "¿Esta entrega ya se completó?"
- "¿En qué momento se marcó como entregada?"
- "¿Hubo incidencias (reintentos, fallos en la firma)?"

Lógica de negocio:

1. Recibe un evento de entrega (por ejemplo, status = DELIVERED).
2. Valida que haya una transición válida de estados (por ejemplo, de IN_TRANSIT a DELIVERED, no desde un estado inválido).
3. Marca la entrega como completada en DynamoDB, guardando:
   - delivery_id
   - status = DELIVERED
   - delivered_datetime
   - driver_id, vehicle_id, route_id
   - metadata adicional (como ubicación final).

¿Para qué?
Para tener un registro rápido del estado actual y alimentar pantallas operativas que muestran:

- Entregas completadas hoy.

- Entregas pendientes.
- Entregas con retrasos.

Técnicamente, esta Lambda se dispara:

- Desde la Lambda de recepción (cuando el evento indica entrega completada).
- desde un mensaje en una cola o topic (SNS / SQS / EventBridge).

5.2. Lambda: cálculo de tiempo estimado de llegada (ETA)

Esta función responde a la necesidad de:

- Estimar cuándo llegará el pedido a su destino.
- Proveer información tanto al conductor como al cliente final (p. ej. en un portal web o notificación).

Inputs típicos:

- Ubicación actual (lat, lon) del vehículo.
- Destino de la entrega.
- Velocidad promedio histórica en esa ruta / franja horaria.
- Tiempo restante estimado calculado a partir de:
  Distancia restante.
  Condiciones de tráfico (eventualmente, integrable con APIs externas).

Lógica básica:

1. Calcular la distancia restante en la ruta.
2. Estimar el tiempo restante = distancia_restante / velocidad_promedio.
3. ETA = hora_actual + tiempo_restante_estimado.

Uso de resultados:

- Guardar la ETA en DynamoDB para consulta rápida.
- Devolver la ETA a la app que hizo la solicitud (API Gateway → Lambda → respuesta HTTP).
- Opcionalmente, publicar notificaciones (por ejemplo, cuando hay cambios significativos en la ETA).

5.3. Lambda: alerta por desvío de ruta

Objetivo de negocio:

Detectar cuando un vehículo se aleja de la ruta esperada, lo que puede indicar:

- Problemas de seguridad.
- Errores en navegación.
- Desvíos que afectarán tiempos de entrega.

Lógica técnica:

1. La ruta planificada se puede representar como una serie de puntos GPS o un “corredor” geográfico.
2. Lambda recibe periódicamente la ubicación actual del camión.
3. Calcula la distancia mínima entre la posición actual y la ruta planificada.
4. Si la distancia excede un umbral (por ejemplo, 1 km), se considera un desvío.
5. En caso de desvío:
   - Se registra el evento (S3 / DynamoDB).
   - Se dispara una alerta (por ejemplo, a SNS, email o push a un panel de monitoreo).

5.4. Uso de DynamoDB como estado actual de entregas

DynamoDB es ideal para:

- Consultas rápidas por clave primaria (delivery_id).
- Altos volúmenes de lectura/escritura con baja latencia.
- Esquema flexible (se pueden agregar atributos sin alterar una estructura rígida).

Ejemplos de items en DynamoDB:
Tabla deliveries_state con:

- PK: delivery_id
- Atributos: status, last_update, eta, driver_id, vehicle_id, current_location, is_deviated, etc.

Esto permite a FleetLogix tener un “single source of truth” operacional en tiempo real, separado del Data Warehouse histórico.

5.5. Notificaciones y alertas (SNS / email / push)

Para cerrar el ciclo de procesamiento en tiempo real:

- Las Lambdas pueden publicar mensajes en Amazon SNS cuando:
  Se completa una entrega.
  El ETA cambia drásticamente.
  Se detecta un desvío de ruta.

Estos mensajes pueden:

- Enviar emails a coordinadores.
- Integrarse con sistemas de notificaciones push.
- Alimentar herramientas de monitoreo interno.

Desde el negocio, esto se traduce en:

- Mayor visibilidad.
- Mejores tiempos de reacción ante incidentes.

- Mejor experiencia del cliente (información oportuna).

6. PARTE C – BASES DE DATOS ADMINISTRADAS Y PERSISTENCIA

6.1. Migración de PostgreSQL local a AWS RDS

¿Por qué RDS?
Manejo automático de:

- Parches, actualizaciones del motor.
- Backups automáticos.
- Alta disponibilidad (Multi-AZ).

Propuesta:

- Crear una instancia de Amazon RDS for PostgreSQL.
- Migrar el esquema y los datos existentes de la base local fleetlogix.
- Usar RDS como base operacional central para:
  Información estructurada de entregas, rutas, vehículos, conductores.
  Fuente de datos para el Data Warehouse en Snowflake (además de S3).

6.2. Uso de DynamoDB para estado en tiempo real

La coexistencia de RDS y DynamoDB permite:

- RDS: transacciones tradicionales, relaciones complejas, integridad referencial.
- DynamoDB: acceso rápido a estado actual de entregas, sin joins, con escalabilidad horizontal.

Desde el punto de vista de diseño:

- RDS se usa para los procesos batch, históricos y entidades “maestras”.
- DynamoDB se usa como “cache operacional persistente” para eventos recientes y estados vigentes.

6.3. Backups automáticos y recuperación ante desastres

RDS:

- Activar backups automáticos diarios con retención (ej. 7–30 días).
- Considerar snapshots manuales antes de cambios mayores.

DynamoDB:

- Activar Point-in-Time Recovery para poder restaurar la tabla a un punto anterior.

S3:

- Es inherentemente duradero (11 nueves), pero se pueden usar versionado y políticas de ciclo de vida para mover datos antiguos a clases de almacenamiento más baratas.

A nivel de negocio, esto significa que FleetLogix puede recuperarse de fallos sin pérdida significativa de información.

7. DISEÑO DE LA API (EXTRA CREDIT)

7.1. Rutas de API Gateway y métodos HTTP

Propuesta de endpoints:

1. POST /deliveries/events
   - Recibe cualquier evento relacionado con una entrega.
   - Cuerpo (ejemplo):

```
{
  "delivery_id": "DEL-123",
  "driver_id": "DRV-45",
  "vehicle_id": "VEH-10",
  "timestamp": "2025-11-13T12:34:56Z",
  "status": "IN_TRANSIT",
  "location": { "lat": 4.610, "lon": -74.082 }
}
```

2. GET /deliveries/{delivery_id}/status
   Devuelve el estado actual de la entrega desde DynamoDB.

3. POST /deliveries/eta
   Calcula y devuelve la ETA actual basada en la ubicación informada.

4. POST /deliveries/route-deviation
   Recibe la ubicación actual y verifica si existe desvío de la ruta esperada.

7.2. Integración con Lambdas correspondientes

- POST /deliveries/events → Lambda ingest_delivery_event
- GET /deliveries/{delivery_id}/status → Lambda get_delivery_status
- POST /deliveries/eta → Lambda calculate_eta
- POST /deliveries/route-deviation → Lambda check_route_deviation

Cada Lambda:

- Usa roles de IAM restringidos para acceder solo a los recursos necesarios (S3, DynamoDB, logging…).

- Devuelve respuestas en formato JSON con códigos HTTP adecuados (200, 400, 500…).

7.3. Ejemplo de URL y uso desde la app móvil

Tras desplegar la API en API Gateway, se obtiene una URL similar a:
https://abc123.execute-api.us-east-1.amazonaws.com/prod

La app móvil consumiría:

- POST https://abc123.execute-api.us-east-1.amazonaws.com/prod/deliveries/events

- GET https://abc123.execute-api.us-east-1.amazonaws.com/prod/deliveries/DEL-123/status

Incluyendo cabeceras de autenticación (por ejemplo, API Keys, JWT o Cognito).

8. SEGURIDAD, MONITOREO Y BUENAS PRÁCTICAS

8.1. IAM y control de acceso

Buenas prácticas:

- Definir roles de IAM específicos para:
  Lambdas (acceso mínimo a S3, DynamoDB, RDS si aplica).
  API Gateway (autorización, logs).
- Usar el principio de “least privilege”: cada servicio solo puede hacer lo estrictamente necesario.
- Proteger la API con:
  API Keys, o
  Autenticación basada en Cognito o tokens JWT emitidos por un Identity Provider.

8.2. Logs y métricas (CloudWatch)

AWS CloudWatch Logs:

- Registra logs de Lambda y API Gateway.
- Permite crear filtros para detectar errores frecuentes.

CloudWatch Metrics:

- Métricas de invocaciones de Lambda, errores, duración.
- Métricas de API Gateway (latencia, errores 4xx y 5xx).

Se pueden configurar alarmas (por ejemplo, vía SNS) cuando:

- Aumentan los errores 5xx de la API.
- Lambda tiene tasas altas de fallo.
- La latencia promedio excede un umbral.

8.3. Análisis individual de cada consulta

- Lambda y API Gateway son servicios serverless: escalan automáticamente según la carga.
- S3 se adapta a volúmenes crecientes de datos con costo por GB almacenado.
- DynamoDB permite:
  Modo on-demand (pago por uso) o provisionado con auto-escalado.
- RDS requiere dimensionar la instancia, pero puede aumentarse en capacidad según crezca la flota.

Desde la perspectiva del negocio, esto permite empezar con un costo moderado y escalar a medida que aumentan los vehículos, entregas y volumen de datos.

9. DIAGRAMA DE ARQUITECTURA AWS

Descripción detallada del diagrama aws_architecture_diagram.html
El diagrama debe reflejar los componentes y flujos descritos:

1. App móvil de conductor
   Envía solicitudes HTTP (REST) hacia API Gateway.

2. API Gateway
   Recibe y enruta peticiones a diferentes Lambdas según la ruta:
   /deliveries/events → ingest_delivery_event
   /deliveries/eta → calculate_eta
   /deliveries/route-deviation → check_route_deviation
   /deliveries/{id}/status → get_delivery_status

3. Lambdas

   - Escriben eventos raw en S3.
   - Actualizan el estado actual en DynamoDB.
   - Opcionalmente, interactúan con RDS PostgreSQL para datos adicionales.
   - Publican alertas en SNS.

4. S3 (data lake raw)

- Es la fuente histórica para procesos de integración hacia Snowflake.

5. RDS (PostgreSQL)
   - Base transaccional en la nube, reemplazo del PostgreSQL local.

6. DynamoDB
   - Estado actual de las entregas, consultado por APIs o herramientas internas.

7. Snowflake
   - Data Warehouse externo donde ya existe el modelo estrella con fact_deliveries y dimensiones.

En el diagrama se pueden incluir flechas diferenciadas para:
- Flujo en tiempo real (App → API Gateway → Lambda → DynamoDB / SNS).
- Flujo histórico (Lambda → S3 → procesos ETL → Snowflake).

10. CONCLUSIONES Y APORTES DEL AVANCE 4 AL PROYECTO COMPLETO

El Avance 4 consolida la evolución de FleetLogix desde una base transaccional local hacia una arquitectura moderna en la nube, preparada para:
- Ingesta en tiempo real de eventos de la flota mediante API Gateway y Lambda.
- Almacenamiento robusto y barato de históricos en S3, organizado por fecha.
- Monitoreo del estado actual de las entregas utilizando DynamoDB.
- Base de datos administrada en RDS, que reemplaza la infraestructura local de PostgreSQL.
- Procesos automáticos de negocio como:
  Verificación de entregas completadas.
  Cálculo de ETA.
  Detección de desvíos de ruta y emisión de alertas.

En conjunto con el Data Warehouse en Snowflake implementado en el Avance 3, la organización obtiene:
- Una visión histórica de largo plazo (Snowflake) para análisis estratégico.
- Una visión operativa en tiempo real (AWS) para decisiones tácticas.

Aunque en este avance no se implementa la infraestructura real en AWS, la arquitectura propuesta es coherente, escalable y alineada con buenas prácticas cloud. Además, deja preparado el camino para:

- Automatizar la integración entre S3 / RDS y Snowflake.
- Extender las capacidades analíticas (por ejemplo, modelos de ML para predicción de retrasos o optimización de rutas).
- Incorporar mecanismos avanzados de seguridad y observabilidad.