Proyecto base de datos Transmilenio

Thomás Santiago Rivera Fonseca
Maestria en Ingenieria de sistemas
Pontificia Universidad Javeriana
Bogotá D.C
thomasrivera@javeriana.edu.co

Laura Sofia Sotto Perez
Maestria en Ingenieria de sistemas
Pontificia Universidad Javeriana
Bogotá D.C
laurasotto@javeriana.edu.co

Miguel Angel Rippe Pereira
Maestria en Analitica para la Inteligencia de Negocios
Pontificia Universidad Javeriana
Bogotá D.C
rippem@javeriana.edu.co

1. INTRODUCION

La movilidad urbana representa uno de los principales desafíos en las grandes ciudades latinoamericanas. En el caso de Bogotá, el sistema masivo de transporte TransMilenio constituye el eje central de la movilidad pública, movilizando diariamente millones de pasajeros. Sin embargo, su operación involucra una compleja red de estaciones, buses articulados, rutas troncales y portales, cuyos datos se encuentran dispersos en diferentes fuentes y formatos. Esta fragmentación dificulta la integración, el análisis histórico y la simulación de escenarios operativos en tiempo real.

En este proyecto se propone el diseño e implementación de una arquitectura de datos en la nube, capaz de simular la interacción del sistema TransMilenio a través de la ingesta, procesamiento y análisis de datos provenientes de tres fuentes principales: datos históricos, datos maestros y datos de simulación.

La solución se desarrolla sobre servicios de Amazon Web Services (AWS) y Atlas, utilizando una combinación de tecnologías de almacenamiento y procesamiento que permiten cubrir diferentes necesidades del ecosistema de datos.

1.1 Definición del problema

El sistema TransMilenio, aunque constituye la columna vertebral del transporte público de Bogotá, enfrenta limitaciones significativas en la gestión y monitoreo de su operación en tiempo real. Actualmente, no existe una integración efectiva de los datos de las rutas, estaciones y buses, lo que impide contar con una visión consolidada del estado operativo del sistema.

Esta fragmentación provoca que las entidades responsables de la operación carezcan de información actualizada sobre la ubicación, frecuencia y disponibilidad de los vehículos, así como sobre la situación de las estaciones en momentos de alta demanda o contingencias. En consecuencia, el sistema es vulnerable a colapsos operativos ante eventualidades como congestiones, desvíos, cierres de rutas o fallas mecánicas, sin una capacidad inmediata de reacción o predicción.

La falta de una infraestructura de datos integrada y analítica limita la posibilidad de anticipar comportamientos críticos y optimizar decisiones operativas. Además, la ausencia de mecanismos de simulación impide evaluar escenarios hipotéticos o validar estrategias de mejora bajo diferentes condiciones de tráfico.

Por tanto, el problema principal radica en la carencia de un ecosistema de datos unificado y en tiempo real que permita monitorear, analizar y simular el funcionamiento de TransMilenio, afectando directamente la eficiencia, la planeación y la experiencia de los usuarios.

1.2 Objetivos

- Diseñar e implementar en el período del semestre una arquitectura de datos que integre tres tipos de bases de datos relacional, geográfica y analítica con el fin de garantizar la disponibilidad y consistencia de la información operativa de TransMilenio.
- Desarrollar un simulador en Python que genere y almacene coordenadas geográficas de buses en tiempo real con una frecuencia configurable logrando éxito en las inserciones de datos en la base de datos geográfica.
- Construir un dashboard analítico interactivo sobre datos almacenados en la base de datos analítica que permita visualizar métricas clave del sistema. Y junto a ello un aplicativo web que alcance una latencia de consulta menor a 8 segundos y un nivel de disponibilidad del 99 % durante las pruebas finales.

1.3 Alcance

El presente proyecto abarca el diseño, implementación y evaluación de una arquitectura de datos en la nube orientada a la simulación operativa del sistema TransMilenio de Bogotá.

El alcance está definido bajo las siguientes actividades:

1. Implementación de tres diferentes tipos de bases de datos aprendidas durante el desarrollo del curso

2. Desarrollo de una capa de integración y procesamiento de datos

3. Desarrollo de una API Rest que facilita la comunicación de los componentes de datos y el cliente Web

4. Construcción de un dashboard analítico para la operación histórica y un aplicativo web para la capa operativa.

5. Configuración de los servicios de soporte.

Fuera del alcance

El proyecto no contempla integración directa con la red existente de Transmilenio, ni una simulación de todas las rutas actuales del sistema. Tampoco busca realizar la predicción del tráfico o análisis de comportamientos de pasajeros.

Así mismo, no busca realizar una simulación con equipos de GPS o posición geográfica en tiempo real, ni el consumo de datos provenientes de sensores o APIs oficiales.

2 ANALISIS Y MODELADO DE DATOS

En este punto nos enfocaremos en la descripción de los datos que vamos a almacenar (fuentes, volumen y características) y junto a ello el modelado de datos para cada base de datos seleccionada.

2.1 Identificacion y Exploración de datos

2.1.1 Aurora Postgres

Aurora PostgreSQL actúa como la base de datos transaccional del sistema. En este motor se almacena toda la información estructurada necesaria para operar el dominio del proyecto TransMilenio: rutas, estaciones, buses, tipos de bus y relaciones entre estos elementos. Los datos aquí manejados son de naturaleza estable, con una estructura clara, relaciones fuertes y cardinalidades bien definidas.

Las operaciones en esta base de datos son principalmente de lectura, ya que el sistema consulta con frecuencia la topología del sistema (por ejemplo, cuáles estaciones pertenecen a cada ruta o qué buses están asignados a cada línea). Las escrituras son bajas y ocurren únicamente cuando se registran nuevas rutas, se modifica la definición de un bus, se ajusta la estructura del sistema o se crea un nuevo usuario en la plataforma. Aunque en este proyecto no se utiliza la funcionalidad de usuarios finales, el modelo contempla entidades básicas asociadas a ellos.

Aurora PostgreSQL es adecuado para este tipo de información por las siguientes razones:

Consistencia garantizada:
Las relaciones entre rutas, estaciones y buses necesitan claves foráneas estrictas para evitar datos inválidos (por ejemplo, asignar un bus a una ruta inexistente).

- ✓ Consultas jerárquicas y navegaciones predecibles: El API consulta constantemente estructuras como “todas las estaciones de la ruta en orden” o “qué ruta opera este bus”. Estas consultas se benefician de índices relacionales y joins eficientes.

- ✓ Esquema estable: Los datos maestros no cambian constantemente, por lo que un modelo relacional tradicional encaja mejor que un esquema flexible.

- ✓ Volumen manejable: La cantidad de filas por tabla es reducida, lo que permite mantener alta disponibilidad y baja latencia.

2.1.2 Datos simulador

Los datos operativos en tiempo real representan la capa dinámica del sistema y están conformados por eventos geoespaciales generados por el simulador. Cada evento incluye la latitud, longitud, timestamp, identificador del bus, velocidad y su estado operativo. Se optó por almacenarlos en MongoDB Atlas utilizando documentos GeoJSON con índices 2dsphere.

El volumen estimado de estos datos depende directamente de la granularidad configurada en el simulador, para un bus en movimiento a través de una ruta, se generan entre diez y veinte posiciones por cada interpolación entre puntos de trayectoria dependiendo el escenario. Considerando que una ruta promedio puede contener entre cuarenta y sesenta segmentos, un recorrido completo produce entre cuatrocientos y mil doscientos eventos geoespaciales por bus. Al simular múltiples buses de manera simultánea, el volumen total puede alcanzar varios miles de documentos por ciclo operativo.

Finalmente, los datos históricos procesados constituyen la capa analítica del sistema. Estos datos son una versión consolidada y transformada de los registros operativos y maestros, incorporando agregaciones, cálculos temporales y métricas derivadas. Provienen del procesamiento realizado en AWS Glue y se almacenan en S3 en formatos columnar Parquet, lo que optimiza su posterior carga en Amazon Redshift.

2.2 Modelado de datos

2.2.1 Diagrama ER Postgres DB

El modelo entidad–relación (ER) representa la estructura lógica utilizada en Aurora PostgreSQL. Incluye las entidades: ruta, estación, estación_por_ruta, bus, tipo_bus, persona, tipo_documento y tipo_persona. Estas entidades reflejan la

topología del sistema y los datos maestros requeridos para soportar la simulación.

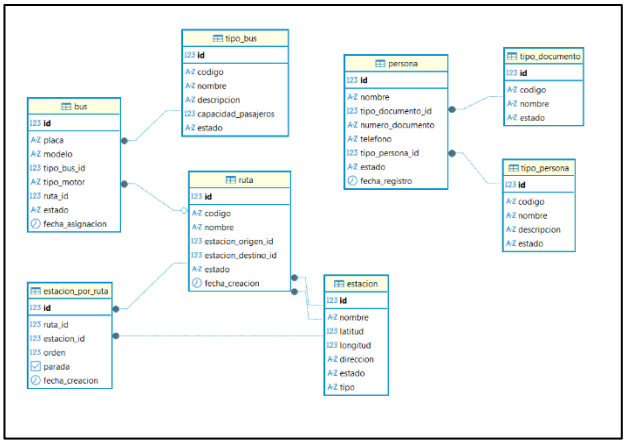

Figura 1: Diagrama Entidad relación Base Relacional

2.2.2 Modelo de datos MongoDB

- ***Estructura***

Para representar cada evento geoespacial generado por el simulador, se definió un documento en formato JSON estructurado según las especificaciones de GeoJSON, estándar utilizado ampliamente para el almacenamiento de información geográfica.

Campo	Tipo	Descripción
bus_id	string	Identifica de forma única el bus (ej: T110)
ruta	string	Ruta activa (ej: J10 / B10)
escenario	string	Contexto de movilidad (hora_pico, hora_valle, etc.)
estado	string	Estado operacional (en_tramo, en_parada, detenido, etc.)
direccion	string	Dirección: ida / vuelta
timestamp	datetime	Tiempo original del evento
ingested_at	datetime	Tiempo en que la API lo procesó
location.type	string	Siempre “Point”
location.coordinates	double[lon, lat]	Obligatorio **en orden [lon, lat]**
velocidad_kmh	double	Velocidad instantánea
tramo.*	object	Información del tramo actual del bus
metrics_runtime.*	object	Métricas internas del simulador

Tabla 1: Campos de almacenamiento colecciones

Este documento constituye la unidad básica de almacenamiento en MongoDB Atlas y contiene todos los atributos necesarios para describir el estado instantáneo de un bus dentro del sistema. El componente central del documento es el campo location, el cual se ajusta estrictamente al estándar GeoJSON requerido por los índices espaciales 2dsphere de MongoDB.
Este campo se compone de dos propiedades: el atributo *type*, que en este caso corresponde a *Point*, y el arreglo *coordinates*, el cual sigue el orden obligatorio [longitud, latitud], acorde a la convención definida por GeoJSON y los operadores espaciales de MongoDB. Esta estructura permite ejecutar consultas eficientes basadas en cercanía, intersección o pertenencia a zonas delimitadas por polígonos complejos.

- ***Índices***

La base geoespacial implementa dos tipos de índices esenciales en las colecciones: un índice geoespacial tipo *2dsphere* y un índice de expiración (*TTL*). El índice *2dsphere* se aplica sobre el campo location en formato GeoJSON, permitiendo que MongoDB ejecute consultas eficientes basadas en geometrías y proximidad.
Por su parte, el índice TTL se utiliza en la colección histórica para eliminar automáticamente documentos cuya antigüedad supere 24 horas. Esto permite mantener controlado el crecimiento de la colección, evitando que los registros se acumulen indefinidamente y saturen el almacenamiento o degraden el rendimiento de las consultas.

- ***Otros campos complementarios***

El documento completo incluye no solo la posición geográfica del bus, sino también metadatos operativos que permiten contextualizar el evento dentro del flujo de la simulación. Se registra el identificador del bus *bus_id*, la ruta en la que se encuentra, y un *timestamp* que facilita ordenamientos temporales y análisis posteriores. El campo estado describe la condición operativa en ese instante como movimiento, detención o transición entre estaciones, mientras que *velocidad_kmh* almacena la velocidad calculada para ese punto de la trayectoria.

Variables adicionales como escenario permiten distinguir entre condiciones simuladas, por ejemplo, hora pico u hora valle, y permiten realizar análisis comparativos dentro del dashboard analítico. Asimismo, el campo dirección proporciona información sobre el sentido del recorrido, particularmente relevante para rutas bidireccionales.

Para complementar el análisis espacial, el documento incluye un bloque denominado tramo, que describe el segmento de la ruta en el que se encuentra el bus. Este segmento se define por los nombres de la estación de inicio y fin, junto con una fracción numérica que indica el progreso relativo entre ambos puntos.

Finalmente, el documento incorpora un bloque denominado *metrics_runtime*. Estos atributos incluyen la cantidad total de pasos interpolados para el segmento (*steps_per_segment*), el paso actual del recorrido (*current_step*) y los pasos restantes para completar el tramo.

- ***Colecciones***

En el diseño del modelo documental dentro de MongoDB Atlas, se estableció la necesidad de separar los datos operativos en dos colecciones complementarias: una colección dedicada a almacenar el historial completo de posiciones geoespaciales generadas por los buses a lo largo de la simulación, y otra destinada exclusivamente a mantener la última posición conocida de cada vehículo.
Esta separación responde tanto a razones de rendimiento como de optimización de las consultas, en especial consultas de proximidad a estaciones o a un punto dado.

La colección histórica funciona como un repositorio detallado, diseñado para capturar cada evento generado por el simulador.

Se creó una segunda colección mucho más liviana orientada exclusivamente a almacenar la última posición disponible de cada bus. Esta colección opera como una vista operacional simplificada: contiene un único documento por vehículo, el cual es actualizado cada vez que el simulador genera un nuevo punto de ubicación. En conjunto, la estrategia de separar los eventos en dos colecciones permite equilibrar rendimiento, escalabilidad y funcionalidad.

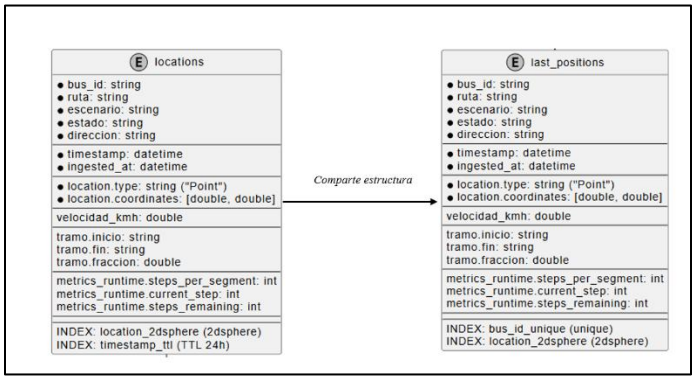

Figura 2: Diagrama base de datos MongoDB

2.2.3 Modelo de datos Redshift

El modelo analítico construido en Amazon Redshift organiza la información operativa proveniente del simulador dentro de una tabla de hechos denominada fact_bus_position_glue, diseñada para soportar consultas históricas. Dentro de esta tabla, dos llaves son importantes el DISTKEY y el SORTKEY. Ambas determinan cómo se distribuyen y ordenan físicamente los datos en Redshift, y por lo tanto influyen directamente en la velocidad de las consultas y en la eficiencia del procesamiento masivo.

En este modelo se eligió ruta como DISTKEY. Esta decisión se fundamenta en el hecho de que gran parte de las consultas analíticas del proyecto se basan en el análisis de una ruta específica o en la comparación entre rutas. Al utilizar ruta como clave de distribución, todos los registros asociados a una misma ruta quedan almacenados en el mismo nodo físico del clúster. Esto reduce de manera significativa el tráfico de datos entre nodos, evita operaciones de redistribución durante las consultas.

Por otro lado, se seleccionó ts_evento como SORTKEY, lo que determina que Redshift organice físicamente los bloques de almacenamiento en función del timestamp original del evento. Dado que la tabla representa datos de movimiento capturados en tiempo real, prácticamente todas las consultas analíticas consumen los datos en alguna forma de orden temporal; filtrando por rangos de tiempo, obteniendo posiciones recientes, agrupando por día y hora o calculando latencia y secuencias de movimientos.

Esta combinación es especialmente eficaz para un sistema como el simulado, en el que existe una fuerte correlación entre la consulta por ruta y la consulta por tiempo. El resultado es una tabla analítica que aprovecha las capacidades de Redshift, garantizando eficiencia en cargas históricas y velocidad en consultas de análisis.

2.3 Justificacion de BBDD y consultas

2.3.1 Base de datos Relacional: AWS Aurora

Dentro de la arquitectura propuesta, Amazon Aurora PostgreSQL fue seleccionada como la base de datos relacional principal para el manejo de la información estructurada y transaccional del sistema, específicamente la relacionada con estaciones, rutas, buses, portales y usuarios.

Realizamos una comparación técnica con diferentes alternativas como por ejemplo MySQL AWS RDS, que posee menor rendimiento en consultas paralelas y replicación más limitada. A partir de esta comparación, se determinó que Aurora PostgreSQL ofrece un equilibrio óptimo entre rendimiento, escalabilidad y compatibilidad con SQL estándar, permitiendo mantener la consistencia de datos críticos en la simulación del sistema TransMilenio.

Las razones claves para la elección de Aurora además de ser nativa en AWS son:

- Compatibilidad total con PostgreSQL estándar, lo cual facilita la implementación de consultas complejas, vistas materializadas y funciones almacenadas, indispensables para procesar relaciones entre rutas, buses y estaciones.

- Desempeño superior a una base de datos relacional tradicional, gracias a su motor distribuido optimizado.

- Escalabilidad automática de almacenamiento y capacidad, permitiendo crecer de forma transparente según el volumen de datos generados por la simulación o la expansión de entidades del sistema.

- Integración nativa con otros servicios AWS, como Lambda, Step Functions, Glue y Redshift, facilitando la orquestación de procesos ETL y el traspaso de información hacia la capa analítica.

- Costo optimizado por uso y rendimiento, ya que Aurora emplea un modelo de cobro basado en consumo efectivo y almacenamiento compartido

En el contexto del proyecto, Aurora PostgreSQL cumple el rol de repositorio central de datos maestros, garantizando integridad referencial dentro del sistema TransMilenio. Desde esta base se alimentan las operaciones de simulación, por ejemplo, la obtención de coordenadas de estacione y se provee información estructurada que complementa la capa analítica del dashboard final.

Se escogió PostgreSQL porque el proyecto requiere consultas estructuradas y consistentes sobre los datos maestros del sistema. Este motor relacional permite mantener relaciones claras entre rutas, estaciones y buses, facilitando los tipos de consultas necesarias para la simulación, entre ellas:

- Validación para determinar si un bus se encuentra dentro de su ruta
- Total, de usuarios puede ser por tipo de documento o tipo de persona.
- Buses por ruta, estaciones por rutas.
- Consultas agregadas para alimentar el dashboard analítico

2.3.2 Mongo DB Altlas

Para el manejo de las posiciones geográficas inicialmente escogimos AWS DocumentDB que permitía integración completa con todos los servicios de Amazon y traía las capacidades completas de MongoDB. Sin embargo, en la fase de pruebas se identificaron costos más altos asociados a la operación continua del clúster, especialmente en escenarios de carga variable y procesos en tiempo real.

La base de datos geográfica es una de las más importantes en el proyecto ya que permite almacenar y procesar datos espaciales como ubicaciones, áreas, rutas (excelente para nuestro caso), fronteras y nos permiten responder preguntas de proximidad ejemplo entre buses o buses y estaciones. Adicional, soporta el formato estándar GeoJson compatible con sistemas GIS y API's de mapas. Entonces se adapta perfectamente a los requerimientos que necesitamos para almacenar coordenadas.

MongoDB nos ofrece además índices y operadores geoespaciales que nos permiten trabajar los puntos y rutas. Ideales para las ubicaciones de los buses y las estaciones. Así mismo, las coordenadas generadas por el sistema corresponden a eventos que pueden variar en estructura.

El modelo documental de MongoDB permite almacenar estos datos sin esquema rígido, adaptándose a cambios futuros en la simulación.

Los tipos de consultas que nos permite realizar esta base de datos geográfica son:

- Consultas de buses cercanas a una estación o portal
- Consultas por buses cercanos a un punto dado
- Validación si un bus está dentro de la ruta
- Recuperar el historial reciente de las posiciones

2.3.3 AWS Redshift

Nuestra tercera y última base de datos es AWS Redshift, un Dataware House con capacidades analíticas bastante avanzadas. Se seleccionó Amazon Redshift como el motor de almacenamiento y procesamiento columnar para consultas históricas y analíticas del sistema simulado de Transmilenio.

Redshift permite analizar grandes volúmenes de datos de manera eficiente, ejecutar agregaciones complejas y soportar cargas de trabajo típicas de BI.
Se evaluaron alternativas como Athena, Google BigQuery, Snowflake y motores relacionales tradicionales, pero Redshift ofreció el equilibrio ideal entre costo en su capa serveless, integración nativa con AWS y capacidad de análisis estructurado requerido por el proceso. Además, Amazon nos provee de 300 dólares de prueba para experimentación.

Las razones técnicas además de la integración nativa en la nube de AWS son que Redshift está diseñado para agregaciones, sumatorias y particionamiento por fechas lo que nos permite realizar el análisis de rutas y estaciones reduciendo el tiempo de consulta con respecto a la base de datos relacional.

El proyecto incluye un dashboard analítico que no está en tiempo real y es perfecto para la capa por que soporta consultas repetitivas de baja latencia y mantiene un modelo analítico optimizado. Las consultas que podemos identificar son

- Frecuencia de buses por hora y por ruta
- Ocupación promedio por estación
- Tiempo promedio entre estaciones

Las consultas de Aurora, MongoDB se encuentran documentadas en el Anexos 1 y 2.

3 DISEÑO DE LA ARQUITECTURA DE LA SOLUCION

3.3 Descripcion general

La arquitectura propuesta integra múltiples servicios de AWS y bases de datos para soportar la simulación operativa del sistema TransMilenio. El diseño sigue un enfoque modular, donde cada componente cumple un rol específico dentro del flujo de ingesta,

almacenamiento, procesamiento y visualización de datos. La arquitectura se divide en cinco capas principales: ingesta operativa, procesamiento ETL, almacenamiento, visualización, y servicios de soporte.

3.4 Diagrama

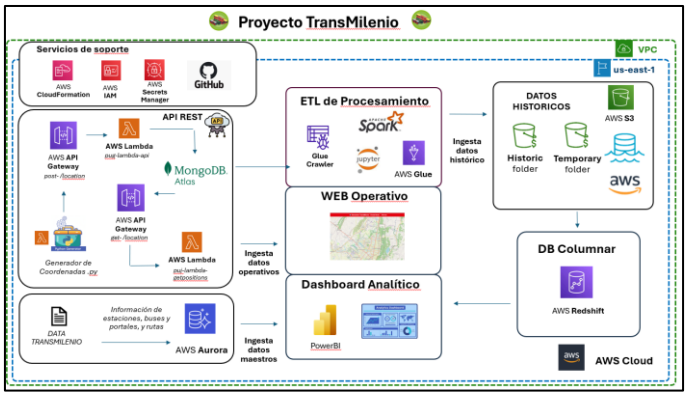

Figura 3: Diagrama Arquitectura

3.5 Tecnologias

Varias de las tecnologías se iran describiendo durante la explicación de la arquitectura. Es importante resaltar que la solución que planteamos usamos tecnologías recientes y escalables en el tiempo. Lo que permite que más adelante todo el sistema pueda evolucionar y lo pueda implementar Transmilenio.

3.5.1 Capa de ingesta operativa:

- ***Lenguajes y librerías***

La capa operativa del proyecto está construida sobre funciones AWS Lambda esta capa se seleccionó la última versión Python 3.13 como lenguaje principal, debido a su eficiencia, compatibilidad con AWS y amplia disponibilidad de librerías geoespaciales y de conexión a bases de datos.

Python ofrece una sintaxis clara y concisa que permite desarrollar funciones Lambda altamente legibles y fáciles de mantener. Esto es crítico en un entorno serverless donde el código debe ser liviano y modular.

Las librerías externas usadas fueron pymongo para conectar Python con mongo y psycopg2 esta librería permite ejecutar consultas SQL parametrizadas, actualizar registros maestros y recuperar configuraciones clave utilizadas por el simulador. Todas las librerías externas las encapsulamos en Layers para usarlas en Lambda.

Por su parte, la librería boto3, el SDK nativo de AWS para Python, fue importante para la interacción segura y programática con servicios como AWS Secrets Manager. Mediante boto3, las Lambdas recuperan credenciales almacenadas de forma cifrada, registran eventos operativos, coordinan tareas del pipeline y consumen servicios auxiliares sin necesidad de exponer información sensible en el código.

Además de estas dependencias externas, se utilizaron librerías nativas de Python como json, datetime, decimal y random, responsables del formateo de mensajes, manejo de fechas, serialización de objetos y generación de valores aleatorios que simulan la variabilidad del movimiento de los buses. La librería requests se empleó en el desarrollo y las pruebas del simulador para enviar datos hacia el API REST, así como para invocar servicios internos cuando fue necesario.

- ***AWS API Gateway***

Amazon API Gateway es un servicio completamente administrado que facilita la creación, publicación, mantenimiento, monitorización y protección de API a cualquier escala. Las API actúan como la puerta de entrada para que las aplicaciones accedan a las solicitudes hacia funciones AWS Lambda dedicadas.

Esto permite que la arquitectura soporte variaciones en la cantidad de solicitudes generadas por el simulador o por múltiples clientes simultáneamente, manteniendo tiempos de respuesta y sin comprometer la infraestructura.

Una característica especialmente útil para el proyecto es la capacidad de API Gateway de integrarse de forma nativa con AWS Lambda, lo que permite desacoplar completamente la lógica de negocio del procesamiento de solicitudes HTTP.

- ***AWS Lambda***

Inicialmente planteamos la arquitectura con un servidor backend sobre AWS EC2 (Elastic Compute), pero tuvimos varias limitaciones en el despliegue de este. Entonces decidimos usar AWS Lambda, AWS ofrece Lambda para enfocarse únicamente en el código, mientras el servicio se encarga de toda la administración de infraestructura, lo que acelera el desarrollo y optimiza los costos. Esto nos ayudó a no preocuparnos tanto en la administración y mantenimiento del servidor.

Las Lambdas ejecutan validaciones básicas, transformaciones ligeras y se conectan directamente a MongoDB Atlas, donde se almacenan las posiciones y eventos geoespaciales en tiempo real.

3.5.2 Capa de Almacenamiento y datos maestros

Los datos estructurados que no cambian frecuentemente como rutas, estaciones, portales y buses se almacenan en AWS Aurora PostgreSQL, la base de datos relacional del proyecto.

Aurora funciona como el repositorio consistente y transaccional para los datos maestros del sistema TransMilenio. A través de conectores dedicados hacia PowerBI, el dashboard va a ser capaz de leer la información alojada en el la Base de datos y alimentarse desde la misma.

Para establecer la conexión JDBC entre Power BI y AWS Aurora, fue necesario implementar un certificado raíz de Autoridad de Certificación (CA). Este certificado permite cifrar el tráfico de datos mediante protocolos SSL/TLS (PostgreSQL), garantizando la seguridad de la información en tránsito. La CA actúa como entidad de confianza al firmar el certificado del servidor, lo que permite a PowerBI validar la autenticidad de la instancia de base de datos antes de establecer la comunicación.

Atlas MongoDB

MongoDB Atlas es una plataforma de base de datos completamente administrada diseñada para ejecutar cargas operativas modernas en la nube sin necesidad de administrar servidores, parches, ni configuraciones manuales de clústeres. A diferencia del uso tradicional de MongoDB en infraestructura propia, Atlas automatiza todo el ciclo de vida del clúster aprovisionamiento, escalado, balanceo de cargas, seguridad, actualizaciones y monitoreo, permitiendo que el equipo de desarrollo se concentre únicamente en la aplicación.

La plataforma opera sobre proveedores como AWS, GCP y Azure, permitiendo desplegar clústeres distribuidos globalmente y con réplica automática entre zonas de disponibilidad para garantizar alta disponibilidad. Atlas integra cifrado en tránsito y en reposo por defecto, autenticación gestionada, firewalls de red, controles de acceso granular y auditorías nativas, eliminando la necesidad de construir manualmente mecanismos de seguridad. En términos de rendimiento, la plataforma ofrece escalado vertical elástico y autoscaling automático, así como almacenamiento flexible basado en volúmenes administrados.

3.5.3 Capa de procesamiento ETL

El procesamiento batch y la consolidación histórica está a cargo de AWS Glue, ejecutando un script en PySpark que integra los datos provenientes de:

- ✓ MongoDB Atlas (datos geograficos)
- ✓ S3

Para mover la información entre MongoDB Atlas y AWS, usamos varias herramientas para conectarnos y descubrir datos de diferentes fuentes.

- ***AWS Glue Conexiones***

La conexión de AWS Glue es un objeto del Catálogo de datos que almacena credenciales de inicio de sesión, cadenas de URI, información de nube privada virtual VPC y otros datos de un determinado almacén de datos. Los rastreadores, trabajos y puntos de conexión de desarrollo de AWS Glue utilizan conexiones para acceder a ciertos tipos de almacenes de datos.

Creamos la conexión de Mongo DB en el catálogo de conexiones de Glue.

- ***AWS Crawler***

Un Rastreador de AWS Glue o AWS Glue Crawler un rastreador puede rastrear varios almacenes de datos en una única ejecución. Cuando finaliza, el rastreador crea o actualiza una o varias tablas del Catálogo de datos. Los trabajos de extracción, transformación y carga (ETL) que definimos en AWS Glue usan estas tablas del Catálogo de datos como orígenes y destinos. La ETL lee del almacén de datos que se especifica en la tabla de origen del Catálogo de datos.

Principalmente su trabajo es clasificar los datos para determinar el formato, el esquema y las propiedades asociadas de los datos sin procesar desde Mongo para agrupar los datos en tablas o particiones y finalmente escribir los metadatos en el Catálogo de datos

- ***AWS Jupyter Notebook Jobs y Apache Spark***

Para la capa de procesamiento de datos del proyecto se adoptó AWS Glue, aprovechando sus capacidades de ejecución de scripts ETL y su integración nativa con Jupyter Notebooks y el motor distribuido Apache Spark. Esta combinación permitió desarrollar, probar y desplegar transformaciones los datos históricos de Mongo DB.

AWS Glue ofrece un entorno interactivo basado en Jupyter Notebook, conocido como Glue Studio Notebooks, que nos permitió diseñar los jobs de procesamiento directamente en la nube. En este contexto, Apache Spark se convierte en un componente central. Spark permitió manipular los datasets de forma eficiente gracias a su arquitectura basada en RDDs y DataFrames distribuidos, y acelera tareas como limpieza, normalización, enriquecimiento, unión entre fuentes y generación en este caso de la tabla de hechos.

La capacidad de particionar datos por rangos temporales y de escribir salidas optimizadas en formato Parquet fue relevante para la alimentación de Amazon Redshift y para la persistencia en S3.

Además, Spark ejecutado dentro de Glue aseguró la capacidad serveless, sin requerir configuración manual de clústeres EMR ni administración de nodos. El servicio gestiona

automáticamente el aprovisionamiento de los recursos necesarios. El uso conjunto de Glue Notebooks más Spark permitió combinar la flexibilidad de un ambiente interactivo con la potencia de un motor distribuido.

3.5.4 Capa de Analitica

- ***Data WareHouse***

En el proyecto se implementó Amazon Redshift Serverless, dado que permite ejecutar cargas analíticas con un modelo de pago por uso, sin necesidad de gestionar clústeres ni nodos dedicados. Dentro de esta modalidad, la arquitectura exige la configuración de dos componentes fundamentales: el Namespace y el Workgroup, además de un endpoint JDBC para conexiones externas.

El Namespace representa la unidad lógica de almacenamiento y metadatos dentro de Redshift Serverless. Su rol principal es actuar como el contenedor del catálogo de bases de datos, esquemas y objetos SQL, configuraciones de seguridad y credenciales y políticas asociadas.

El Workgroup es la capa de cómputo elástica que se activa para ejecutar consultas SQL y cargas analíticas. En él se configuran los parámetros que controlan: Capacidad de procesamiento RPU, VPC, subnets y security groups, credenciales de conexión y auto escalado.

Finalmente, se configuró un endpoint JDBC en el workgroup, que funciona para realizar cargas, ejecutar consultas y conectar herramientas externas como PowerBI y AWS Glue a través de una nueva conexión JDBC en AWS Glue Connections.

- ***Dashboard***

En la capa de visualización analítica del proyecto se evaluaron inicialmente varias alternativas disponibles para conectarse a Amazon Redshift. La primera opción considerada fue Amazon QuickSight, debido a su integración nativa con el ecosistema AWS, su capacidad para conectarse directamente al clúster de Redshift

Sin embargo, durante las pruebas se identificaron limitaciones asociadas principalmente a la estructura de costos del servicio. QuickSight utiliza un modelo de cobro por usuario, el cual incluye tarifas mensuales y costos adicionales por lector. Para un proyecto académico o de prototipado, este esquema de facturación inviable y generaba un costo mayor al esperado.

Debido a estas restricciones, se optó por adoptar Power BI como plataforma principal para el dashboard analítico. Power BI ofrece un entorno muy completo para la construcción de reportes, con un conjunto robusto de visualizaciones y un motor de modelado altamente flexible. Una ventaja significativa es que permite desarrollar dashboards de manera local en Power BI Desktop sin costos adicionales. Para los requerimientos del proyecto de visualizar resultados analíticos derivados del procesamiento en Redshift y Power BI Desktop proporciona todas las funcionalidades necesarias sin generar costos operativos junto con el conector nativo a Redshift.

3.6 Consideraciones no funcionales

3.6.1 Servicios de soporte

- ***IAM***

IAM es el pilar de la seguridad dentro del proyecto. Permite gestionar identidades, permisos y políticas de acceso siguiendo el principio de privilegios mínimos. Cada Lambda, Glue Job, API Gateway y base de datos utiliza roles específicos para evitar accesos no autorizados.
El uso de políticas bien delimitadas garantiza que los servicios solo puedan interactuar con los recursos estrictamente necesarios, reduciendo el riesgo ante fallos o malas configuraciones.

Se aprovisionaron 3 usuarios para el desarrollo de todo el proyecto, cada uno con los permisos necesarios

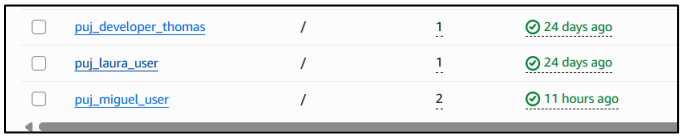

Figura 4: Usuarios IAM

Asi mismo, se configuraron los roles para los servicios con las políticas necesarias, algunos servicios usaron políticas directamente. Pero para la mayoría de ellos tuvo su propio rol.

Figura 5: Roles IAM

- ***Secrets manager***

Para proteger credenciales y cadenas de conexión a MongoDB Atlas, Aurora y Redshift, se empleó Secrets Manager. Este servicio almacena claves cifradas y otorga acceso a ellas únicamente a los roles Lambda autorizados, evitando incluir secretos en el código o en variables de entorno. Además, permite la rotación automática y reduce vulnerabilidades relacionadas con exposición accidental de credenciales. Lo anterior para cumplir con los estándares de seguridad del proyecto.

Figura 6: Secrets almacenados

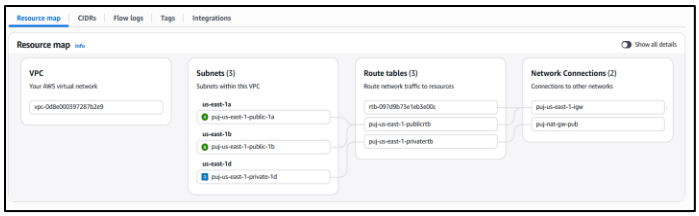

Figura 7: Flujo de VPC

- ***CloudFormation***

Toda la infraestructura del proyecto: API Gateway, Lambdas, Glue Jobs, Redshift, Aurora, los definimos como código mediante plantillas de CloudFormation. Esto facilita la reproducibilidad, permite crear ambientes idénticos, reduce inconsistencias y habilita versionamiento de la arquitectura, algo esencial para proyectos de nube

- ***GitHub***

GitHub permitió mantener un repositorio centralizado del código del simulador, Lambdas, scripts de Glue y plantillas de despliegue. Esto asegura trazabilidad, control de cambios, colaboración y facilidad para restaurar versiones previas o reproducir despliegues.

3.6.2 Seguridad

La arquitectura también incorpora una serie de consideraciones de seguridad que abarcan desde el cifrado en tránsito hasta la segmentación de red. Las conexiones hacia bases de datos externas como MongoDB Atlas se establecen mediante TLS, mientras que Aurora y Redshift operan dentro de subnets publicas. API Gateway actúa como la primera barrera al filtrar y autenticar solicitudes entrantes hacia Lambda, complementado por reglas específicas de security groups que restringen el tráfico únicamente a los servicios autorizados.

- ***VPC***

Para garantizar la conectividad segura y controlada entre los distintos componentes del proyecto, fue necesario diseñar una arquitectura de red basada en una Amazon VPC (Virtual Private Cloud), dentro de la cual se distribuyeron las subnets, tablas de enrutamiento y security groups utilizados por Redshift, Lambda y los demás servicios involucrados en el sistema. La VPC funciona como un entorno de red aislado dentro de AWS, permitiendo definir rutas, accesos y restricciones de tráfico de forma precisa.

Subnets

Dentro de esta VPC se configuraron dos subnets públicas, distribuidas en distintas zonas de disponibilidad. La decisión de utilizar subnets públicas se fundamentó en un requerimiento específico de Amazon Redshift Serverless: para permitir conexiones externas a través de JDBC, especialmente desde herramientas de desarrollo como DBeaver y Power BI, el Workgroup de Redshift debía colocarse en subnets capaces de enrutar tráfico hacia internet a través de un Internet Gateway. Al ubicarse en estas subnets públicas, Redshift cuenta con la capacidad de establecer conexiones salientes necesarias para autenticación, inicialización del Workgroup y habilitación del endpoint JDBC.

Así mismo, se creó una subnet privada para la comunicación interna entre los otros servicios. La subnet privada tiene una tabla de enrutamiento para dirigir el trafico de salida a través de un Nat Gateway que es un servicio de traducción de direcciones de red. Utilizamos una puerta de enlace NAT para que las instancias de una subred privada puedan conectarse a servicios fuera de la VPC, pero los servicios externos no pueden iniciar una conexión con esas instancias.

Security groups

En paralelo, dentro de la misma VPC se definieron security groups especializados, que actúan como firewalls virtuales para controlar el tráfico entrante y saliente de cada componente. Cada security group fue configurado con reglas estrictas que limitan el acceso únicamente a los servicios autorizados. Por ejemplo, el security group asignado al Workgroup de Redshift permite conexiones JDBC exclusivamente desde nuestras IP's, evitando accesos no autorizados y garantizando que únicamente Lambdas internas, Glue Connections.

Las funciones Lambda utilizaron un security group propio, que restringe el tráfico entrante por completo y únicamente permite conexiones salientes a servicios puntuales: la API de AWS para recuperación de secretos, el endpoint del Workgroup de Redshift, el puerto correspondiente a MongoDB Atlas y otros recursos necesarios para la simulación.

3.6.3 *Escalabilidad*

En cuanto a la escalabilidad, la arquitectura se apoya casi por completo en servicios serverless o administrados. Las funciones AWS Lambda escalan automáticamente en función del número de solicitudes provenientes del simulador o del cliente web, sin necesidad de aprovisionamiento manual.

MongoDB Atlas ajusta su capacidad automáticamente para manejar incrementos en lecturas o escrituras geoespaciales. Redshift Serverless incrementa su capacidad de cómputo cuando la carga analítica lo requiere y se desacopla de la capa de almacenamiento para mantener costos controlados. S3 garantiza almacenamiento ilimitado tanto para datos históricos como para los archivos intermedios procesados por Glue, mientras que Spark provee distribución automática del procesamiento.

3.6.4 *Rendimiento*

El rendimiento es otro factor clave en el diseño del sistema. Gracias al uso de Python y librerías optimizadas dentro de Lambda, la comunicación entre API Gateway y MongoDB Atlas ocurre con latencias mínimas, adecuadas para escenarios en tiempo real. La presencia de índices geoespaciales en MongoDB Atlas permite responder consultas de posicionamiento de buses en milisegundos. Redshift Serverless, por su parte, es capaz de ejecutar agregaciones complejas sobre grandes volúmenes de datos en cuestión de segundos, lo que resulta esencial para el dashboard analítico. Glue y Spark aportan un motor distribuido capaz de procesar y transformar grandes datasets históricos con eficiencia, mientras que el uso del formato Parquet en S3 reduce tiempos de lectura y mejora la integración con Redshift.

4 IMPLEMENTACION Y DESPLIEGUE

4.1 *Descripción Aplicación Cliente*

4.1.1 *Lamda simulador aplicación cliente*

Esta Lambda no expone el API, es el cliente del API REST. Su función es simular el movimiento de un bus de TransMilenio, calcular su posición sobre la ruta y enviar en cada invocación un evento JSON al endpoint del API Gateway en una URL autenticado con una API Key. Ese API REST es el que luego se encarga de recibir el JSON y guardarlo en MongoDB Atlas. Empezamos describiendo primero el simulador para entender todo el flujo de la API Rest

- ***Obtención de estaciones y preparación del recorrido***

Al inicio de lambda_handler, la función lee del evento el bus_id, la lista de rutas (por ejemplo ["J10", "B10"]) y la direccion inicial. Con el primer código de ruta (RUTAS [0]) llama a get_estaciones_aurora, que se conecta a Aurora PostgreSQL usando credenciales obtenidas desde Secrets Manager. Esta consulta trae las estaciones de la ruta, en orden, únicamente aquellas marcadas como paradas.

```
SELECT
e.*
FROM transmi.ruta r
JOIN transmi.estacion_por_ruta epr
    ON epr.ruta_id = r.id
JOIN transmi.estacion e
    ON e.id = epr.estacion_id
where r.codigo = '""" + ruta + """' and epr.parada is true
ORDER BY r.codigo, epr.orden;
```

Figura 8: Query de simulador para estaciones

Para no consultar Aurora en cada invocación, el resultado se guarda en una caché global *cached_estaciones* y se reutiliza hasta por una hora. Si ya hay cache y no ha pasado ese tiempo, la Lambda usa las estaciones almacenadas en memoria.

Luego se determina el escenario de simulación con definir_escenario. Si el evento trae escenario, se respeta; si no, se elige aleatoriamente entre HORA_VALLE, HORA_PICO y BUS_VARADO con diferentes pesos. Según el escenario, se fija steps_per_segment:

- ✓ Hora valle: 10 pasos por tramo movimiento más fluido
- ✓ Hora pico: 20 pasos por tramo, movimiento más lento y granular
- ✓ Bus varado: 1 paso no se mueve

El número total de tramos *total_segments* es el número de estaciones menos uno.

- ***Estado interno del bus y avance por la ruta***

La Lambda mantiene variables globales *current_segment, current_step, direction, RUTA, last_fraccion* que conservan el estado del bus mientras el contenedor de la Lambda siga activo. En la primera ejecución se inicializan con el primer tramo, paso cero, dirección inicial IDA y la ruta de ida RUTAS [0].

En cada invocación, si el escenario no es BUS_VARADO, se incrementa *current_step*. Cuando *current_step* supera *steps_per_segment*, se reinicia a cero y se avanza al siguiente tramo. Si la dirección es IDA, el segmento sube; cuando llega al último, se invierte la dirección a VUELTA y se cambia la ruta a la de regreso RUTAS [1]. En VUELTA los segmentos decrecen y, cuando se llega al inicio, se vuelve a IDA y se regresa a la ruta de ida.

Con esa fracción y el escenario se llama a generar_estado_y_velocidad. Si el escenario es BUS_VARADO, el bus queda en DETENIDO con velocidad 0. En cualquier otro escenario si la fracción está muy cerca del inicio o del final del tramo (≤ 0.05 o ≥ 0.95), el estado es

EN_PARADA y la velocidad es baja. En el resto del tramo el estado es EN_RUTA y la velocidad se sortea en un rango distinto según sea hora pico más lenta u hora valle más rápida. Con esto se modelan paradas en estaciones y cambios de velocidad según el escenario.

- ***Interpolación de coordenadas***

La Lambda toma las estaciones actual y siguiente según current_segment. Convierte sus latitud y longitud de decimal a float y llama la función de *interpolar_coordenadas*, que hace una interpolación lineal entre ambos puntos. El resultado es un par lat, lon que representa la posición exacta del bus dentro del tramo en ese instante.

Con toda la información anterior se construye el payload que se enviará al API REST definido con la URL con la estructura que almacena la colección.

- ***Ejecución***

Durante el desarrollo del simulador se evaluó la posibilidad de programar su ejecución periódica mediante Amazon EventBridge. Sin embargo, en el proceso surgieron limitaciones del número de ejecuciones por segundo. Debido a esta restricción, se optó por utilizar un mecanismo alternativo que permitiera mantener la periodicidad del simulador.

La solución consistió en implementar un cliente ligero en Python que invoca la Lambda de manera programada utilizando la biblioteca boto3, el SDK oficial de AWS. Este cliente actúa como un disparador externo que ejecuta la función cada cinco segundos.

La invocación se realiza con el tipo asíncrono InvocationType= Event, lo que permite que el cliente no espere la respuesta de la Lambda y continúe inmediatamente con la siguiente iteración del ciclo. De esta manera, cada ejecución genera un nuevo evento operativo sin bloquear el flujo del programa, garantizando un envío continuo de coordenadas al API REST con intervalos fijos de cinco segundos.

Este enfoque presenta varias ventajas dentro del contexto del proyecto. Al ejecutarse a través del CLI, el simulador puede controlarse manualmente, iniciarse o detenerse según sea necesario para las pruebas, y permite ajustar dinámicamente la frecuencia de envío sin modificar servicios en la nube.

4.2 Descripcion Api Rest

4.2.1 Api Gateway

El API se construyó siguiendo un enfoque de integración directa, conocido como *Lambda Proxy Integration*, en el que API Gateway recibe el cuerpo de la solicitud y las cabeceras originales del cliente y las reenvía sin transformación a la Lambda correspondiente.

El punto principal del servicio es el recurso ***/location***, configurado con el método POST, cuyo propósito es recibir eventos de posición generados por el simulador. Cada invocación incluye un payload que contiene la ubicación del bus en formato GeoJSON junto con información operativa como velocidad, estado, tramo y escenario. API Gateway exige una API Key válida para autorizar esta operación, lo que asegura que solo el simulador pueda enviar datos al sistema.

El segundo recurso expuesto es ***/get_positions***, implementado mediante el método GET, que permite a clientes externos principalmente la aplicación web de monitoreo consultar las posiciones más recientes de los buses. Al igual que en el caso del método POST, este endpoint exige una API Key para su consumo, asegurando que únicamente clientes autorizados puedan acceder a los datos en tiempo real. Cuando el método GET es invocado, API Gateway reenvía el request a la Lambda de consulta, que accede a la colección de últimas posiciones almacenada en MongoDB Atlas y devuelve un conjunto compacto de documentos geoespaciales que representan el estado actual del sistema.

4.2.2 Lambda Api

La función Lambda asociada al endpoint */location* es el componente responsable de recibir, validar y almacenar cada coordenada enviada por el simulador de buses. Su ejecución comienza cuando API Gateway reenvía el cuerpo de la solicitud HTTP utilizando el modo proxy. En la primera etapa, la función extrae el cuerpo del mensaje, lo interpreta. Si alguno de los valores críticos no está presente, por ejemplo, bus_id o las coordenadas la función responde inmediatamente con un error; de igual forma, verifica que la posición se encuentre dentro de un rango geográfico aproximado de Bogotá para evitar almacenar puntos inválidos.

Una vez validados los datos, la Lambda procede a establecer una conexión segura con MongoDB Atlas. Ya conectada, la Lambda accede a la base transmilenio y selecciona dos colecciones: locations, que almacena el historial completo de eventos geoespaciales, y last_positions, que mantiene la última posición actualizada de cada bus para consultas rápidas en tiempo real.

Antes de insertar datos, la función asegura que existan los índices necesarios. Si el índice geoespacial tipo 2dsphere no ha sido creado previamente, la Lambda lo define automáticamente sobre el campo location, tanto en la colección histórica como en la colección de últimas posiciones. De igual forma, en la colección histórica se crea si aún no existe un índice TTL sobre el campo timestamp, configurado para expirar documentos después de 24 horas.

Para almacenar la ubicación recibida, la Lambda construye un documento estructurado que respeta el formato GeoJSON requerido por MongoDB. Este documento incluye no solo la

coordenada interpolada, sino también el estado operativo del bus como se mencionó en la sección de Modelo de datos.

Al completar la inserción, la función genera una respuesta JSON que confirma el almacenamiento de la coordenada, devuelve el identificador del documento insertado en la colección histórica, y replica los principales valores enviados por el simulador. En conjunto, esta Lambda constituye la pieza central de la ingesta de datos en tiempo real del sistema, actuando como un puente confiable entre el API REST y la base de datos geoespacial.

4.2.3 Lambda get

La función Lambda asociada al endpoint ***/get_positions*** cumple el rol de servicio de consulta dentro de la arquitectura operativa. Mientras la *Lambda Api* se encarga de almacenar las coordenadas generadas por el simulador, esta Lambda actúa como el mecanismo mediante el cual el cliente web obtiene las posiciones más recientes y la información necesaria para visualizar el estado del sistema en tiempo real. Cada vez que un cliente realiza una solicitud GET hacia el endpoint, API Gateway reenvía el evento a esta Lambda, que a su vez procesa los parámetros recibidos, resuelve la solicitud y construye la respuesta correspondiente.

Una vez cargadas las estaciones, la Lambda establece una conexión segura con MongoDB. A partir de ese momento, el funcionamiento depende del parámetro mode especificado en el request. cada modo corresponde a un tipo de consulta predefinido sobre las colecciones locations y last_positions. Estos modos incluyen operaciones como recuperar todas las posiciones recientes, filtrar buses por ruta, estado o escenario, así como consultas geoespaciales basadas en operadores nativos de MongoDB, tales como $geoNear. Estas consultas permiten localizar buses cercanos a un punto o a una estación, medir distancias o recuperar fragmentos recientes del historial de movimiento de un bus. Las consultas están detalladas en el Anexo 2

El endpoint siempre devuelve la respuesta enriquecida con las estaciones obtenidas desde Aurora, esta integración permite que el cliente web represente no solo la ubicación actual de los buses, sino también la infraestructura del sistema como estaciones y puntos de referencia sin necesidad de realizar consultas adicionales.

De esta manera, la ***lambda get*** proporciona un mecanismo para consultar en tiempo real el estado de la flota simulada, cumpliendo un papel fundamental en la visualización del sistema TransMilenio operando dinámicamente.

4.3 Descripción ETL

Para integrar los datos generados por el simulador con la capa analítica del proyecto, se desarrolló un proceso ETL ejecutado en modo batch. Este proceso transforma los eventos crudos registrados en MongoDB, principalmente lecturas de posición de los buses, en un conjunto de datos estructurados y optimizados para análisis dentro de Amazon Redshift.

La ETL se ejecuta una vez al día y sigue tres etapas fundamentales.

4.3.1 Extracción:

Se leen desde MongoDB Atlas todos los registros generados en la última jornada desde el catálogo de datos de Glue, ya con los datos catalogados anteriormente desde el crawler. Estos documentos incluyen coordenadas, timestamps, velocidad estimada, estado del bus y metadatos relacionados con el recorrido.

4.3.2 Transformación:

En esta fase se realizan tareas como limpieza y validación de datos, cálculo de latencias entre evento e ingesta, normalización de campos de fecha y hora, estandarización de rutas y tramos, conversión de estructuras anidadas a columnas tabulares y enriquecimiento con atributos derivados para análisis, como fracción del tramo recorrido o pasos por segmento.

4.3.3 Carga:

Los datos transformados se escriben en la tabla fact_bus_position_glue dentro de Amazon Redshift a través de una conexión JDBC. Esta tabla está diseñada como un modelo de hechos que permite consultas rápidas, agregaciones y análisis histórico. Su estructura facilita la construcción de métricas, paneles analíticos y comparaciones entre escenarios.

4.4 Descripción Dashboard

El dashboard analítico desarrollado para el proyecto TransMilenio surge como una herramienta clave dentro del ecosistema de datos, orientada a transformar información compleja en indicadores claros y accionables. Su diseño responde a la necesidad de contar con una plataforma que permita evaluar el rendimiento operativo del sistema, consolidar datos históricos y facilitar análisis comparativos que apoyen tanto la toma de decisiones académicas como la validación técnica del simulador y la arquitectura implementada. Además, se concibe con una visión escalable, preparada para incorporar nuevas métricas y adaptarse a futuros requerimientos del proyecto.

4.4.1. Objetivos

- Evaluar el desempeño operativo del Sistema:
  Analizar métricas clave como velocidades promedio, tiempos de parada y recorridos para identificar patrones de eficiencia y áreas de mejora.

- Consolidar datos históricos frente a la capa en tiempo real:
  Integrar información optimizada en Amazon Redshift con datos operativos provenientes de MongoDB, garantizando una visión completa del sistema.
- Facilitar análisis comparativos entre rutas:
  Permitir la comparación de indicadores entre diferentes líneas y tramos para detectar variaciones en rendimiento y comportamiento.

- Soportar decisiones académicas y técnicas:
  Proveer información confiable para validar el correcto funcionamiento del simulador, los procesos ETL y la arquitectura general del proyecto.
- Permitir escalabilidad futura:
  Diseñar el dashboard con capacidad de incorporar nuevas métricas, como ocupación estimada, tiempos entre estaciones y análisis por hora o día.

El dashboard desarrollado en Power BI constituye la capa final de visualización dentro del ecosistema de datos del proyecto TransMilenio. Su objetivo principal es transformar la información procesada en Amazon Redshift en métricas operativas comprensibles, que permitan evaluar el comportamiento de rutas, estaciones y buses simulados. Se alimenta directamente de la tabla fact_bus_position_glue, generada mediante un proceso ETL diario ejecutado en AWS Glue. Esta tabla concentra información histórica sobre posiciones del sistema, incluyendo:

- ✓ Velocidad del bus.
- ✓ Estado operativo (EN_RUTA o EN_PARADA).
- ✓ Tramos recorridos.
- ✓ Latencia.
- ✓ Fechas y coordenadas procesadas.

Gracias al motor columnar de Redshift, es posible realizar análisis agregados, comparaciones y cálculos derivados de manera eficiente.

El dashboard se sustenta en una arquitectura integrada que garantiza la disponibilidad y calidad de los datos:

- ✓ MongoDB Atlas: actúa como fuente operativa del simulador.
- ✓ AWS Glue + Spark: ejecuta las transformaciones diarias para preparar la información.
- ✓ Amazon Redshift Serverless: funciona como repositorio analítico optimizado para consultas.
- ✓ Power BI: ofrece la capa final de análisis e interpretación.

Gracias a esta integración, se proporciona indicadores clave que permiten validar la efectividad del pipeline de datos y respaldar la toma de decisiones estratégicas.

4.5 Despliegue

El despliegue de la infraestructura completa del proyecto se realizó utilizando AWS CloudFormation y SAM desde la maquina local, adoptando el enfoque de Infrastructure as Code (IaC) para garantizar reproducibilidad, consistencia y una gestión eficiente de todos los recursos involucrados.

CloudFormation permitió describir de manera declarativa los componentes necesarios para la operación del sistema TransMilenio simulado, desde las funciones Lambda hasta la topología de red, pasando por API Gateway, roles IAM, buckets S3 y las configuraciones de integración con MongoDB Atlas y Redshift.

4.5.1 Despliegue VPC

El proceso comenzó por la definición estructurada de la VPC, subnets públicas, tablas de ruteo e Internet Gateway, asegurando que los recursos de Redshift, Lambda y Glue contaran con los elementos de red requeridos para operar correctamente. Estos componentes se especificaron en la plantilla como recursos dependientes, de forma que CloudFormation construyera el entorno en el orden adecuado, aplicando políticas de creación y sustitución sin intervención manual. Dentro de esta misma plantilla se declararon los Security Groups utilizados por Lambda, Redshift y el API Gateway, garantizando que cada uno quedara configurado con las reglas apropiadas de entrada y salida según su función.

4.5.2 Despliegue Aurora

Dentro de la misma plantilla se desplegó Amazon Aurora PostgreSQL, que constituye la base relacional maestra del sistema. La declaración incluyó el clúster Aurora, su instancia asociada, las subnets de la VPC para su operación privada y el respectivo security group que restringe el acceso a servicios autorizados, principalmente las Lambdas.

4.5.3 Despliegue IAM

Se configuraron los roles IAM necesarios para habilitar permisos mínimos y restringidos a cada servicio. Estos roles incluyen acceso a Secrets Manager para la recuperación de credenciales, permisos de ejecución para las Lambdas, políticas de invocación para API Gateway y permisos de escritura hacia S3 para los procesos ETL.

4.5.4 Despliegue Lambdas

Dentro del despliegue también se definieron las funciones Lambda, incluyendo sus variables de entorno, asignación de roles y dependencias de runtime mediante Lambda Layers. Aunque el código de las Lambdas fue desarrollado externamente, CloudFormation referenció las rutas de

despliegue en el repositorio, integrando perfectamente la lógica del simulador, la Lambda de ingesta y la Lambda de consultas.

Del mismo modo, el API REST se declaró completamente en CloudFormation: los recursos /location y /get_positions, sus métodos GET y POST, los enlaces con las Lambdas, los mecanismos de autenticación mediante API Keys y las etapas de despliegue del API.

4.5.5 Despliegue ETL

El pipeline ETL también fue incluido mediante la creación declarativa de AWS Glue Jobs y AWS Glue Crawlers, los cuales se enlazaron con buckets de S3 y con el catálogo de datos. Se incorporaron recursos asociados al almacenamiento histórico, definiendo las carpetas historic y temp

4.5.6 Despliegue Redshift

Finalmente, Amazon Redshift Serverless incluyendo su Namespace, Workgroup, subnets y endpoint JDBC fue definido en la plantilla. Esto permitió integrar de manera automática el almacén analítico con el pipeline ETL sin requerir configuraciones manuales posteriores. La separación declarativa entre el Namespace como capa de almacenamiento y el Workgroup como capa de cómputo quedó asegurada dentro de CloudFormation, facilitando escalamientos o modificaciones posteriores sin afectar al resto del sistema.

5 PRUEBAS Y EVALUACIÓN

5.1 Metodologia de pruebas y resultados:

Las pruebas se centraron en validar el correcto funcionamiento del API, la integración entre los componentes y la consistencia de los datos almacenados en las diferentes bases de datos. Para esto se emplearon solicitudes de prueba enviadas desde Postman, con el fin de verificar que los endpoints respondieran adecuadamente, que la estructura del JSON fuera correcta y que los datos recibidos fueran procesados sin errores.

Posteriormente, se complementó la validación mediante consultas directas en las bases de datos para confirmar que la información enviada por el API fuera almacenada y procesada como se esperaba.

5.1.1 Pruebas Funcionales

Estas pruebas confirmaron que cada endpoint cumplía la lógica de negocio definida.

- *Prueba de ubicación de buses*

Se enviaron solicitudes de prueba al endpoint:
POST /buses/{id}/location

Desde Postman se validó:

- ✓ Código HTTP de respuesta.
- ✓ Tiempo de respuesta.
- ✓ Estructura del JSON devuelto por el API.

Luego se ejecutó en Aurora PostgreSQL:

```
select bus_id, ts_evento, ST_AsText(geom)
from transmi.bus_posicion
order by ts_evento desc
limit 5;
```

Para verificar que las coordenadas fueran registradas correctamente y construidas como geometrías válidas.

- *Prueba de eventos de entrada y salida*

Se enviaron solicitudes a:

- ✓ POST /presence/enter
- ✓ POST /presence/exit

Con Postman se verificaron las respuestas del API y posteriormente se revisó en Atlas MongoDB que los cambios en el estado del bus se reflejaran correctamente.

5.1.2 Pruebas de Integración

Las pruebas de integración permitieron validar el funcionamiento del flujo completo de la arquitectura tal como fue diseñada. El proceso inició con el envío manual del evento desde Postman, donde se construyó un payload equivalente al generado por el simulador para verificar que el API REST respondiera correctamente sin depender aún del generador automático.

Una vez ejecutado el envío, API Gateway procesó la solicitud y la Lambda de ingesta almacenó los datos tanto en Aurora PostgreSQL, para mantener coherencia con los datos maestros, como en MongoDB Atlas, donde se consolidan los eventos geoespaciales en tiempo real. Posteriormente, los datos almacenados en Atlas fueron integrados dentro del pipeline de procesamiento ejecutado mediante AWS Glue, lo que permitió consolidarlos en la tabla de hechos del clúster analítico de Redshift.

Finalmente, las métricas resultantes tales como latencias, patrones de movimiento, velocidades y secuencias temporales fueron validadas mediante consultas analíticas sobre Redshift, confirmando que tanto la estructura del modelo como la lógica de procesamiento reflejaban fielmente los datos enviados en la prueba inicial. De esta manera, la cadena completa desde la entrada del evento hasta la explotación analítica quedó verificada y en funcionamiento.

Ejemplo usado para validar la carga del día en Redshift:

```
select count(*)
from fact_bus_position_rs
where fecha = current_date;
```

5.2.3 *Pruebas de Rendimiento*

Aunque no se ejecutaron pruebas de carga formales, durante las pruebas funcionales se registraron los tiempos de respuesta entregados por Postman.

Esto permitió confirmar que:

- ✓ El API respondía de manera estable.
- ✓ Ningún endpoint superaba los tiempos definidos en los objetivos del proyecto.
- ✓ No se observaron errores HTTP ni interrupciones en la comunicación.

5.2 Resultados:

5.2.1 *Cliente web*

Como resultado final de la integración entre el simulador, el API REST y la base de datos geoespacial, se desarrolló una aplicación web que permite visualizar en tiempo real el movimiento de los buses dentro del sistema TransMilenio simulado. La interfaz se construyó utilizando Leaflet como motor de mapas, integrado con los datos enviados por el simulador y consultados mediante los endpoints del API Gateway. Esta aplicación representa gráficamente la ciudad de Bogotá y actualiza de forma dinámica la posición de cada vehículo, mostrando su desplazamiento continuo sobre la ruta establecida.

El sistema implementa un panel de filtros que permite seleccionar los buses por ruta, por estado operativo o por escenario, facilitando el análisis visual de distintas condiciones de simulación como hora pico, hora valle o eventos de detención. Cada bus se representa mediante un ícono dinámico y se actualiza automáticamente con la información más reciente proveniente de la colección last_positions en MongoDB Atlas. Asimismo, el mapa muestra cada estación como un marcador georreferenciado, permitiendo interpretar con claridad la progresión del bus a lo largo de los tramos durante el ciclo de simulación.

Durante las pruebas se comprobó que el sistema responde en tiempo real, con latencias mínimas entre la emisión del evento por parte del simulador y su representación en el mapa. Los buses avanzan de acuerdo con los valores interpolados generados por la Lambda simuladora, lo que permite observar cambios de velocidad, paradas intermedias y variaciones entre los dos sentidos de la ruta. El comportamiento gráfico refleja fielmente los datos almacenados, confirmando que las capas API, Lambda y base de datos operan de manera coordinada y estable.

En conjunto, la aplicación web constituye una visualización operativa completa del sistema, permitiendo observar de forma clara la dinámica de movimiento de los buses y validando el funcionamiento integral del flujo de datos en tiempo real. Este resultado demuestra la eficacia del diseño arquitectural y la interoperabilidad entre los distintos componentes de la solución.

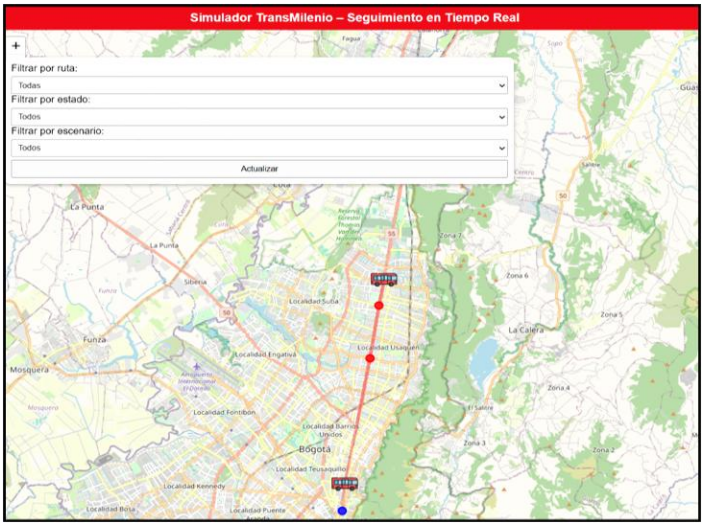

Figura 9: Simulador del Transmilenio

5.2.2 *Dashboard*

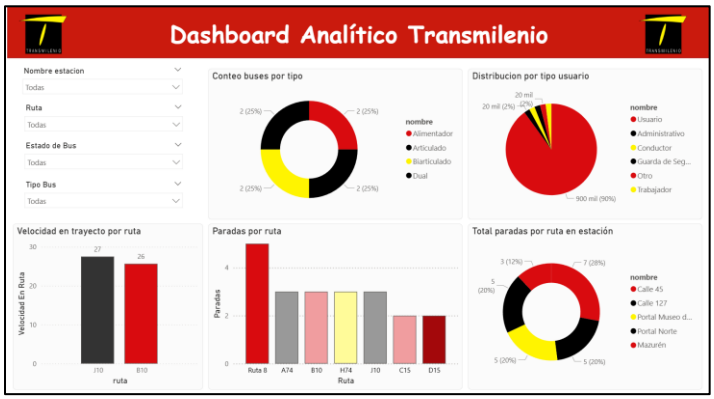

Figura 10: Dashboard Analítico

El dashboard en Power BI consolida los datos provenientes de Aurora PostgreSQL y Redshift para ofrecer una visión general del comportamiento del sistema. A partir de esta integración se generan métricas operativas y descriptivas que permiten analizar las rutas y las estaciones de forma centralizada.

El panel utiliza los datos geográficos, documentales para mostrar tendencias, distribuciones y comparaciones entre rutas y tipos de vehículos. Además, incluye filtros interactivos que

permiten explorar la información sin necesidad de ejecutar consultas directas.

En conjunto, el dashboard resume el estado del sistema y facilita la interpretación de los datos almacenados en las diferentes bases de datos usadas en la arquitectura.

6 CONCLUSIONES

6.1 Resumen:

El proyecto logró integrar de forma efectiva diversos tipos de datos, modelos de almacenamiento y componentes de arquitectura cloud para construir una plataforma analítica orientada al comportamiento de TransMilenio. Se implementó una solución que combina datos estructurados, semiestructurados y datos analíticos derivados, los cuales fluyen a través de servicios que permiten capturar, procesar, transformar y visualizar información operativa simulada con un enfoque moderno de ingeniería de datos.

En la capa de ingestión se utilizaron datos semiestructurados en formato JSON provenientes del simulador, ingresados a través de una API expuesta mediante Amazon API Gateway. Esta API se conectó con una función AWS Lambda que actuó como puente entre el flujo de eventos y la base de datos NoSQL, permitiendo enviar cada mensaje directamente a MongoDB Atlas. Este mecanismo permitió simular el comportamiento de un backend de transporte que recibe lecturas frecuentes de posición y estado de los buses sin necesidad de infraestructura adicional, beneficiándose del modelo serverless y del procesamiento bajo demanda.

En paralelo, se trabajó con datos completamente estructurados para representar la red troncal de TransMilenio. Estos datos maestros incluyeron rutas, estaciones, tramos y relaciones topológicas que permiten interpretar cualquier evento del sistema. Su función fue proveer contexto y consistencia, facilitando que los datos operativos pudieran vincularse correctamente a ubicaciones del recorrido.

Posteriormente, el proceso ETL desarrollado en AWS Glue consolidó los datos operativos y generó un dataset analítico optimizado para consultas históricas. Este proceso realizó transformaciones, cálculos derivados, validaciones básicas y organización temporal, para luego almacenar los resultados en Amazon Redshift Serverless, un motor columnar especializado en cargas analíticas. El uso de Redshift permitió realizar consultas agregadas sobre grandes volúmenes de datos y habilitó la creación de métricas históricas como comportamiento por tramo, velocidades promedio, patrones de detención y distribución de movimiento por rutas.

La combinación de API Gateway, Lambda, MongoDB Atlas, S3, Glue y Redshift permitió demostrar que una arquitectura basada en servicios desacoplados y modelos de datos distintos puede funcionar de manera integrada y coherente. Cada componente cumplió un rol clave: la API y Lambda proporcionaron un mecanismo ligero y seguro para la ingestión de datos, MongoDB ofreció flexibilidad para recibir eventos dinámicos, los datos estructurados garantizaron integridad y Redshift aportó capacidad analítica a gran escala. Esta integración permitió finalmente construir un dashboard en Power BI que refleja el comportamiento del sistema de transporte y permite analizar patrones operativos relevantes.

En conjunto, el trabajo demuestra que es posible articular diferentes tipos de datos y múltiples tecnologías cloud para construir una solución integral de movilidad. La arquitectura final sienta bases sólidas para evolucionar hacia análisis predictivos, procesamiento en streaming, mayor automatización y una plataforma de monitoreo operativo en tiempo real.

6.2 Limitaciones

A pesar de los resultados alcanzados, el proyecto presenta varias limitaciones derivadas tanto del alcance académico como de las restricciones tecnológicas:

- *Simulación en lugar de datos reales*: El sistema se probó con datos generados artificialmente en el simulador de tramos. Aunque estos datos son consistentes y estructurados, no capturan variabilidad real del sistema (congestiones, desvíos, fallos mecánicos, bloqueos, tráfico impredecible, etc.).
- *Uso exclusivo del Free Tier y créditos gratuitos de AWS*
  El desarrollo fue implementado aprovechando los recursos gratuitos de Amazon Web Services (Redshift Serverless trial, Glue Free Tier, S3 y VPC básico). Si bien esto permitió construir la arquitectura completa sin incurrir en costos operativos, también impuso varias restricciones:

  - ✓ No se pudieron ejecutar jobs de Glue con mayor capacidad de cómputo (DPUs), limitando el desempeño del ETL.
  - ✓ Se evitó el uso de servicios avanzados como Glue Streaming, Kinesis, MSK, SageMaker o Redshift Spectrum por sus costos asociados.
  - ✓ No fue posible realizar pruebas de estrés a gran escala porque el consumo adicional de RPU en Redshift Serverless genera costos.

- *ETL con periodicidad diaria*:
  La arquitectura implementada contempla una carga batch, ejecutada una vez al día. Esto restringe la capacidad de realizar monitoreo operativo en tiempo real o cuasi–tiempo real, característica fundamental si se quisiera predecir colapsos operativos o gestionar contingencias.
- *Modelo simplificado de rutas y estaciones:* El modelo maestro de rutas contiene únicamente rutas

troncales seleccionadas, estaciones ficticias y una estructura simplificada de origen–destino. En un entorno real, TransMilenio posee múltiples servicios por ruta, variantes, extensiones y cambios operativos frecuentes.

- *Dependencia del crawler de Glue:* El proceso de catalogación depende del Glue Crawler, lo cual introduce tiempos adicionales antes de poder consultar los datos. Para escenarios operativos reales sería necesaria una gestión más inmediata del esquema y del lineage.
- *Limitaciones de Power BI Free*: La versión utilizada no permite funciones avanzadas como refrescos automáticos, gateways empresariales, integración con RLS, ni dashboards colaborativos en equipo. Para el caso real se requeriría Power BI Pro o Premium.
- *Escalabilidad no evaluada completamente*: El proyecto no sometió la arquitectura a pruebas de rendimiento con grandes volúmenes de datos, como los que generaría un sistema con miles de buses y millones de posiciones diarias. Redshift y Glue pueden escalar, pero no se validó su comportamiento bajo carga masiva.

Estas limitaciones no afectan la comprensión del enfoque propuesto, pero sí muestran que la solución debería fortalecerse para ser adoptada en escenarios operativos reales.

6.3 Líneas futuras

La arquitectura desarrollada representa una primera aproximación funcional a un sistema de analítica operacional para TransMilenio. Sin embargo, su verdadero potencial se manifiesta cuando se consideran las posibilidades de evolución hacia una plataforma completa de movilidad inteligente. La integración de servicios en la nube, modelos predictivos, procesamiento en tiempo real, gobierno de datos y visualizaciones avanzadas permite proyectar una solución similar a la utilizada en centros de control modernos. Con la base diseñada, que incluye el simulador, la ingestión, el proceso ETL, el almacenamiento analítico y la capa de visualización, el proyecto queda en un punto ideal para ampliar su alcance técnico y explorar nuevas capacidades de escalabilidad, automatización, predicción y análisis geoespacial. Estos aspectos permitirían convertir el sistema en una herramienta robusta y adaptable para soportar escenarios reales de operación y planificación del transporte masivo.

7 Referencias

[1] Amazon Web Services, *"Using SSL/TLS to Encrypt a Connection to a DB Instance."* Amazon RDS Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/es_es/AmazonRDS/latest/UserGuide/UsingWithRDS.SSL.html

[2] Amazon Web Services, *"Introducing MongoDB Atlas metadata collection with AWS Glue crawlers."* AWS Big Data Blog, 2023. [Online]. Available: https://aws.amazon.com/blogs/big-data/introducing-mongodb-atlas-metadata-collection-with-aws-glue-crawlers/

[3] Amazon Web Services, *"Amazon API Gateway – Build, Deploy, and Manage APIs."* AWS Services Overview, 2024. [Online]. Available: https://aws.amazon.com/es/api-gateway/

[4] Amazon Web Services, *"API Gateway Developer Guide."* AWS Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/es_es/apigateway/latest/developerguide/welcome.html

[5] Amazon Web Services, *"Amazon VPC NAT Gateway."* AWS VPC User Guide, 2024. [Online]. Available: https://docs.aws.amazon.com/es_es/vpc/latest/userguide/vpc-nat-gateway.html

[6] Amazon Web Services, *"Connecting to Amazon Redshift Serverless."* Amazon Redshift Documentation, 2024. [Online]. Available: https://docs.aws.amazon.com/es_es/redshift/latest/mgmt/serverless-connecting.html

[7] MongoDB Inc., "MongoDB Atlas Documentation." MongoDB Product Documentation, 2024. [Online]. Available: https://www.mongodb.com/docs/atlas/