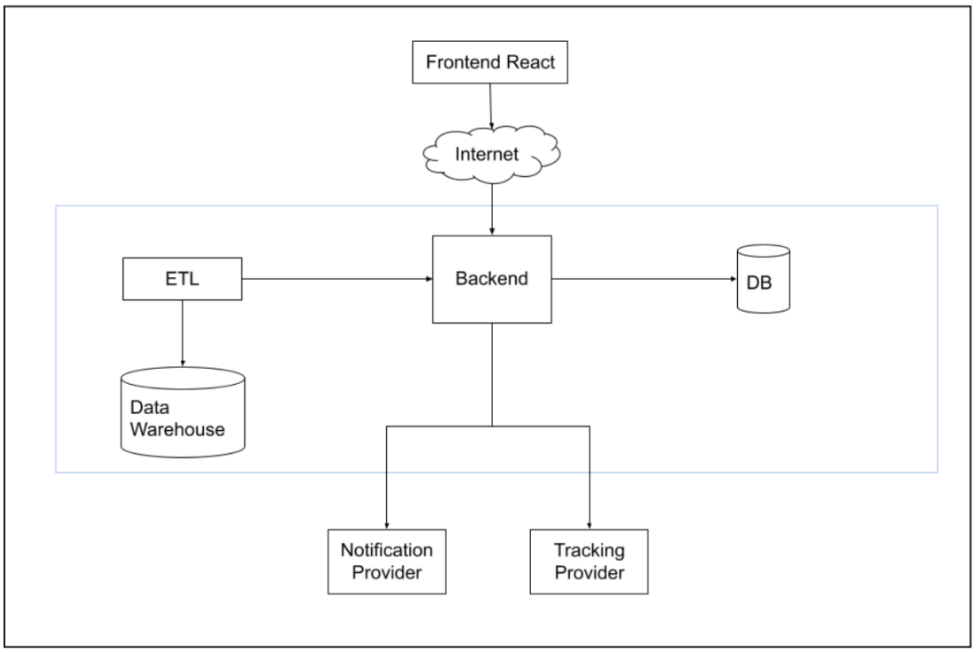
Migración a arquitectura basada en microservicios

1. Situación actual



- **Backend** (Monolítico), tienes las siguientes funcionalidades:
  - Autenticación y autorización según usuario / aplicación
  - Manejo de la lógica de negocio por medio de APIs que son consumidas por una aplicación web en react.
  - Manejo de las notificaciones en el sistema.
  - Manejo de seguimiento de acciones de usuarios. (User behavior analytics).
  - Tiene Apis para extracción de datos que son enviados a un Data Warehouse.

2. Identificación de la problemática actual

a. El servicio de backend tiene muchas responsabilidades, esto crea acoplamiento innecesario a la hora de dar mantenimiento a la aplicación.
b. Al ser un servicio monolítico el backend no puede escalar horizontalmente, por ende no puede atender más usuarios en momentos pico.
c. Si el backend está "caído" todo el sistema estaría inoperativo.
d. Las apis de datawarehouse afectan el performance general de la base de datos y del servicio backend, cuando invocan las apis para obtener data,

pues consumen los recursos del monolito, ¿Qué pasa si se invocan durante un pico de usuarios?

e. El despliegue de un cambio obliga a modificar todo, ¿Se cuenta con algún mecanismo de regresión automática?.
f. Actualmente se tiene una base de datos centralizada para almacenar información estructurada del negocio, si se necesita escalar por un pico de uso, este tipo de solución no es la adecuada.
g. Otro problema de la base de datos es que la escritura y la lectura de información se haría contra la misma fuente de datos, encontrándose momentos donde la Base estaría ocupada atendiendo escrituras y bloqueando lecturas y viceversa. Una solución podría ser tener bases de datos de réplica pero no se dispondría de información en tiempo real para ser consultada.

3. Microservicios propuestos

Se propone crear una arquitectura totalmente serverless basada en microservicios, detallados a continuación:

a. **Servicio de seguridad**, estaría encargado de la autenticación y autorización de usuarios (grano fino), para el uso de los distintos servicios del sistema (apis, aplicación web, tracking de usuarios, etc.).
b. Set de apis de negocio para soportar la administración de cursos, clases, profesores, alumnos, padres y sus relaciones.
c. Lago de datos, Aquí se centraliza la información estructurada y no estructurada, permitiendo crear los datasets necesarios para atender los requerimientos del negocio, este tipo de soluciones escalan por su naturaleza y permitirían aplicar algoritmos de análisis y ciencia de datos para su explotación. Aquí también podríamos almacenar información de acciones de usuarios.
d. **Alimentador (feeder)**, este servicio es el bus de información que alimentará al lago de datos, se basará en servicios streaming para evitar el colapso en la escritura del lago.
e. Cluster de consulta, este cluster estará basado en un motor de búsqueda, que permitirá consultas pesadas en tiempo real, siendo la fuente de datos primaria que permitirá al "**Set de apis de negocio**" extraer información.
f. Servicio de observabilidad, conjunto de métricas, alarmas y dashboards para el monitoreo de los distintos servicios en el sistema.
g. Servicio de analíticas de negocio, este servicio consumirá parte de la información del "lago de datos" para permitirle a los analistas o científicos de datos extraer información valiosa.

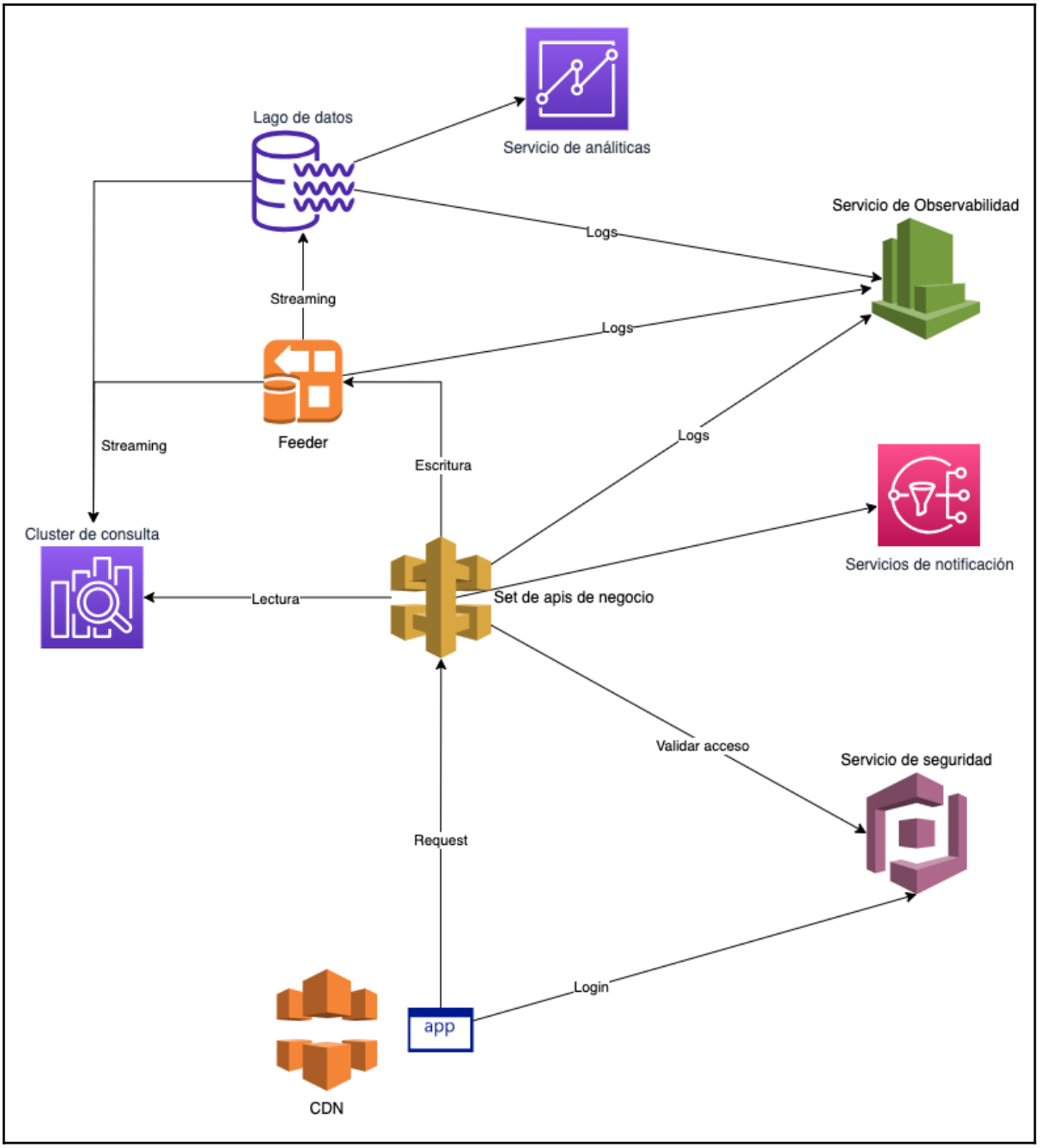


Figura 1: Arquitectura serverless basada en microservicios

4. Plan de implementación/despliegue de servicios

a. **Primera fase: creación de lago de datos, feeder y cluster de consulta,** esta primera fase creará el lago de datos y el cluster de consulta, sin alterar la operativa del sistema actual. Todas las escrituras que se hacen hacia la base de datos, deben de ir notificadas a través del Feeder (Streaming de datos por ejemplo usando kinesis firehose), la fase terminará cuando el lago de datos soporte a todas las entidades del negocio almacenadas en la base

de datos (cursos, clases, profesores, alumnos, padres y sus relaciones), y esas entidades también puedan ser consultadas en el cluster de consultas.

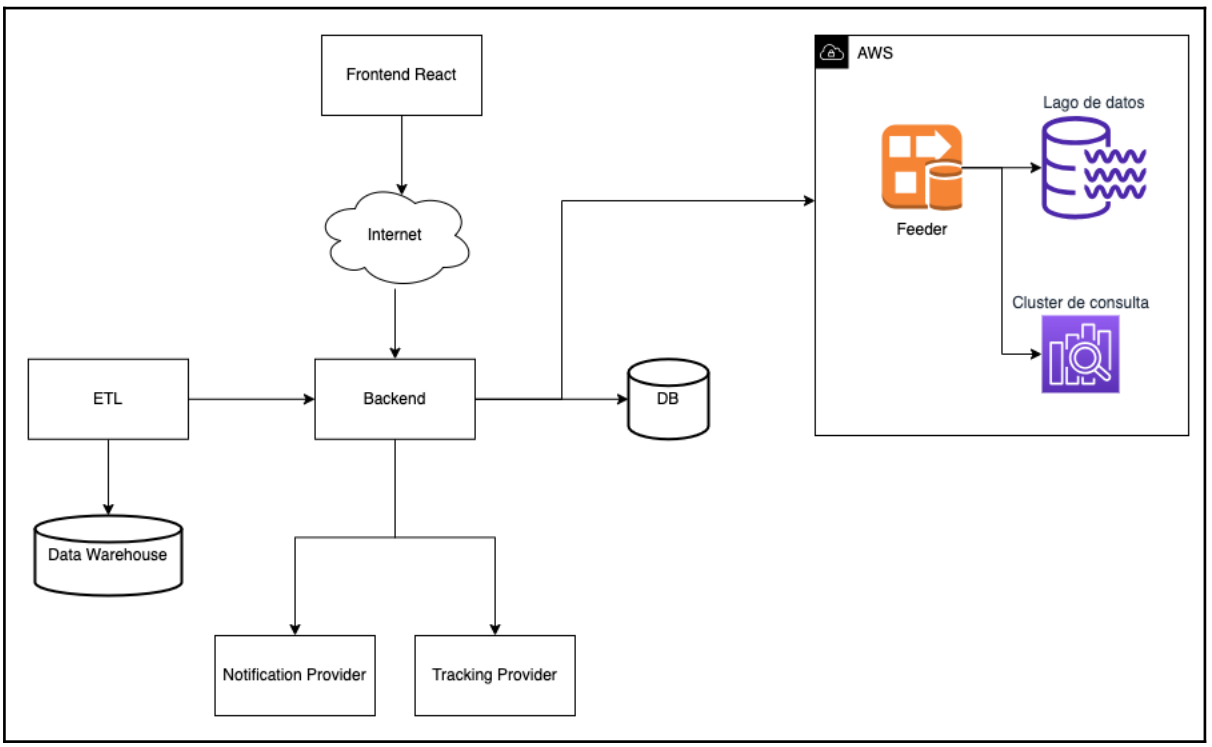


Figura 2: Primera fase

b. **Segunda fase: creación del servicio de seguridad,** Creación del servicio de análiticas y observabilidad. Con estos servicios podemos eliminar el ETL, el Data warehouse y el tracking provider, ya que estos servicios son reemplazados por el lago de datos y el servicio de analíticas.

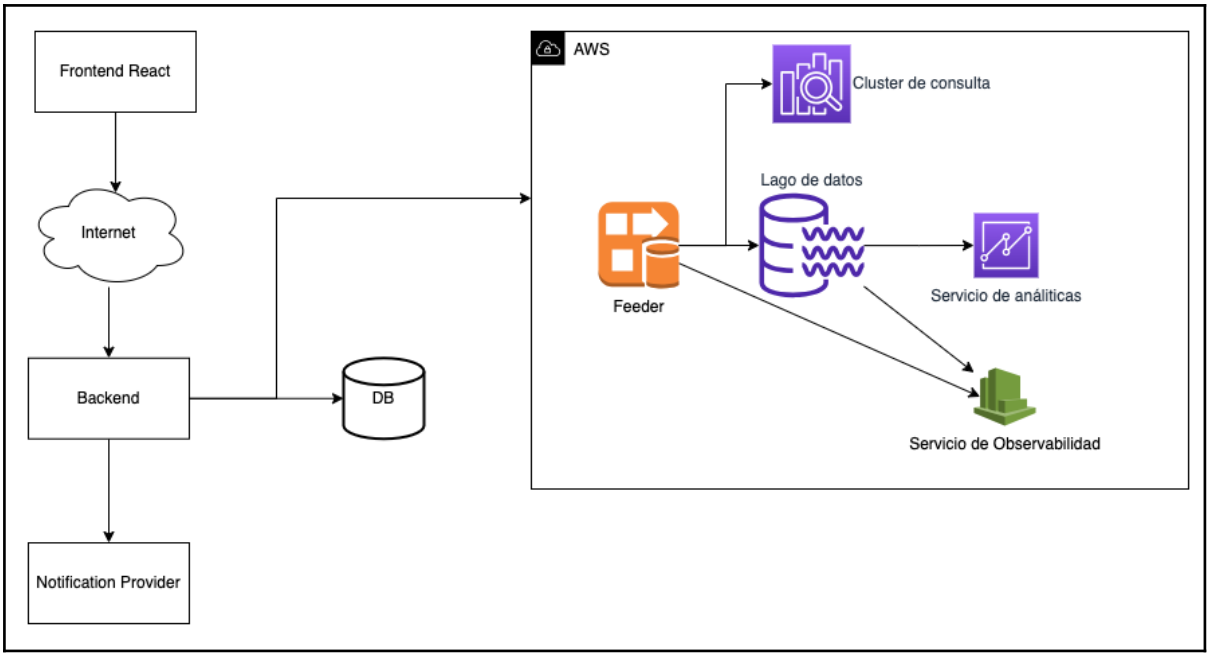


Figura 3: Segunda fase

c. **Tercera fase:** Reemplazo completo del backend, aquí se deben de crear los endpoints en la nube de AWS (api gateway), el servicio de seguridad y los servicios de notificación. Por último se debe de conectar el front end a este nuevo set de servicios usando canary deployments para no afectar la operativa de los usuarios del sistema.

5. Stack tecnológico

Proveedor de cloud services (AWS):

- **Servicio de seguridad**: Cognito, IAM.
- **Set de apis de negocio:** Api gateway, lambda.
- **Lago de datos:** S3, lake formation, Glue.
- **Alimentador (feeder)**, lambda, kinesis firehose.
- **Cluster de consulta,** Opensearch.
- **Servicio de observabilidad,** Cloudwatch.
- **Servicio de analíticas de negocio,** Quick Sight, Athena.

Lenguajes de programación:

- Javascript o Typescript para frontend y backend.
- Python para Lago de datos.
- Python o Typescript para infraestructura mediante código.

6. Características que se deberían tener en cuenta

- Soporte de CI/CD.
- Uso de ambientes de desarrollo, testing y producción separados.
- Testing automatizado.
- Uso de infraestructura mediante código (CDK)