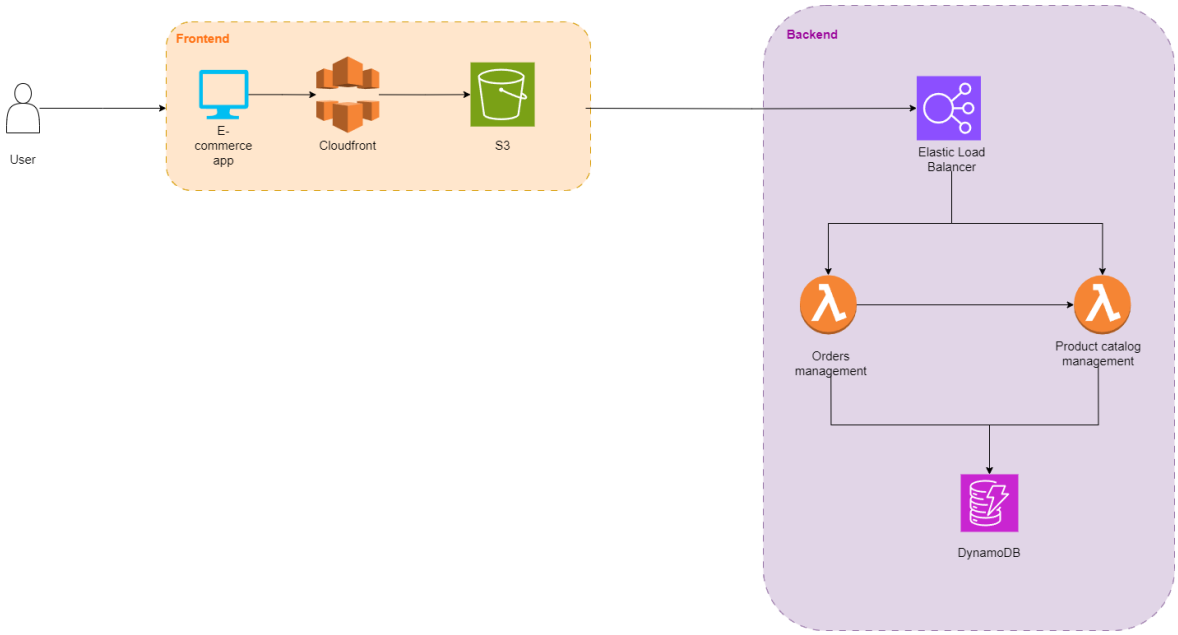
Arquitectura de E-commerce

Para la plataforma de comercio que se quiere desarrollar y a futuro implementar en una nube como AWS, se propone la siguiente arquitectura:



La elección de la arquitectura depende mucho de las prioridades del proyecto a futuro, como la complejidad del manejo de los datos, la escalabilidad o la mantenibilidad, o si, por otro lado, la prioridad es la seguridad, control de tráfico y monitoreo. En este caso, como la prueba no especificaba cuales eran estas prioridades, pero se mencionaba como puntos clave la escalabilidad y mantenibilidad, se tomó como base un escenario en el que se tiene una arquitectura de microservicios sencilla, pero que se espera que tenga una gran cantidad de transacciones por minuto a futuro.

A continuación, se sustentan las diferentes partes de la arquitectura:

1. **Frontend:**

    a. **Uso de Cloudfront + S3:**

    Con Cloudfront aseguramos que el contenido almacenado en S3, el cual suele ser el código fuente de las diferentes vistas de una página web, se muestre de una forma rápida, mejorando los tiempos de carga de las diferentes páginas, además de que en tiempos de temporada alta en el que el e-commerce tenga muchas visitas, podrá responder adecuadamente a pesar de los picos de tráfico. Por otro lado, también permitirá el acceso al contenido desde diferentes regiones.

2. **Backend:**

    a. **Elastic Load Balancer:**

    Debido a que la aplicación está enfocada en escalar, se escogió Load Balancer porque con éste se puede distribuir el tráfico entre servidores de manera eficiente y mejorar la disponibilidad.

    Otra alternativa pudo ser Api Gateway, pero este puede quedarse corto en temas de escalabilidad ya que todo el tráfico va a un solo servidor, y está más enfocado a temas como autenticación, autorización y monitoreo del tráfico. Por otro lado, Load Balancer es más mantenible y fácil de adaptar en caso de que se quieran añadir nuevos microservicios a futuro sin perder la escalabilidad.

    b. **Lambda:**

    Ya que es un proyecto de prueba, se considera que la carga que va a recibir cada servicio no va a ser mucha, además que resulta económico debido a que solo cobra por el tiempo consumido de cómputo y el número de peticiones realizadas. Por otro lado, su configuración es sencilla, ya que es el mismo servicio el que gestiona los recursos según sus necesidades.

    Sin embargo, aunque Lambda puede llegar a procesar miles de peticiones antes de tener un error de throttling (en los que la lambda ya no es capaz de procesar la cantidad de peticiones recibidas de forma concurrente), en caso de que la lambda se quedara corta en este aspecto, se podría considerar cambiar estos servicios por un ECS, en el cual el procesamiento de muchas peticiones simultáneas es mejor y puede aprovisionar o desaprovisionar tareas nuevas según la cantidad de tráfico, pero hay que tener en cuenta que el costo sería mayor.

    c. **DynamoDb:**

    DynamoDB es una base de datos NoSQL flexible, conocida por su rapidez en operaciones de escritura. Aunque en algunos escenarios las lecturas pueden ser más lentas, una de sus principales ventajas es que es administrada por AWS, que escala automáticamente la capacidad según sea necesario. La implementación de nuevas tablas es sencilla y no requiere una configuración extensa para su uso.

    Otra opción pudo ser Redis con ElasticCache, pero debido a que la capacidad de almacenamiento de Redis no es mucha no es la mejor opción, ya que está mas enfocada a guardar información en caché. También se pudo considerar alguna base de datos relacional en RDS, pero las tablas en el proyecto no tienen relaciones complejas entre sí que requieran el uso estricto de este tipo de base de datos, por lo que no se tomó como opción. Por otro lado, RDS requiere configuraciones

manuales para su funcionamiento y también para su escalado, además de que requiere estar encendida para su uso, mientras que DynamoDB es accesible en cualquier momento.