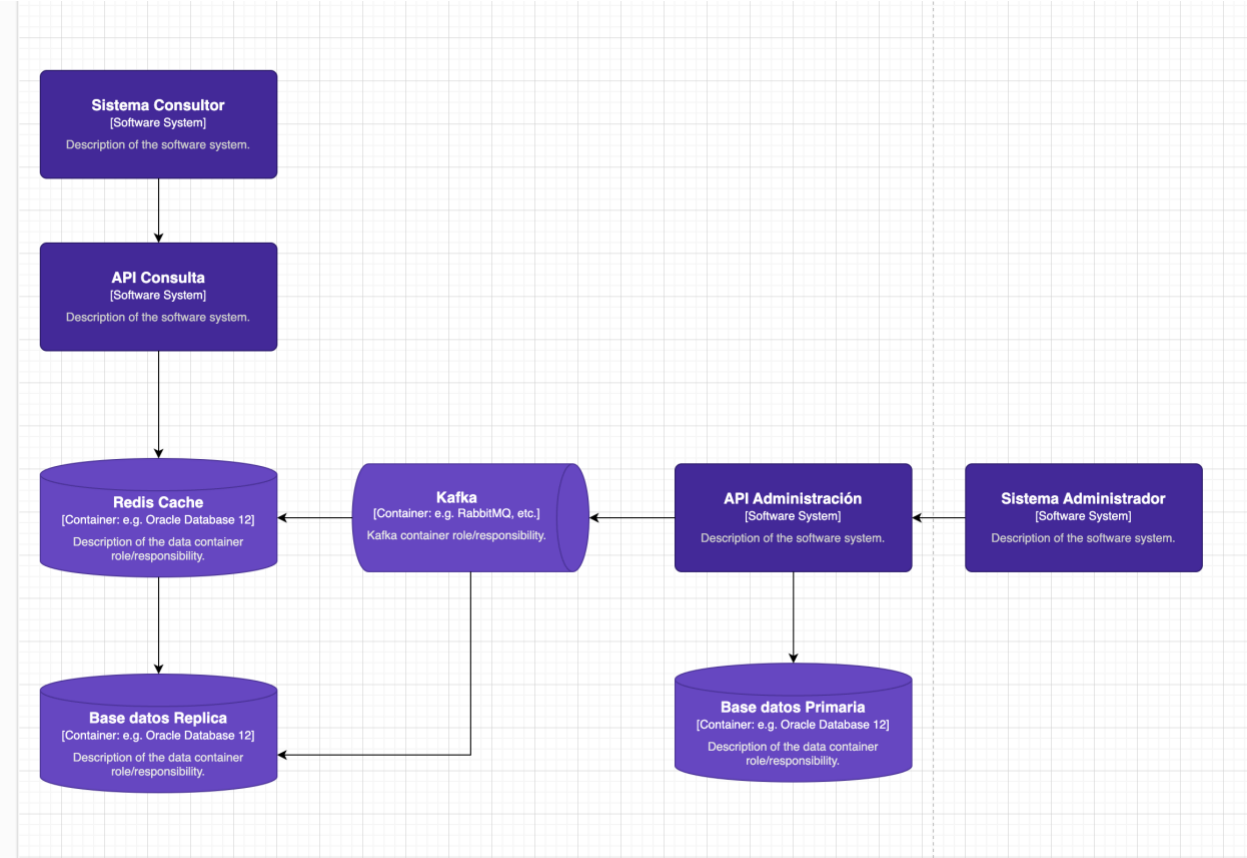
Reto 3

1. Administración de un Catálogo de Productos con Alta Concurrencia

Propuesta de Arquitectura



- **Base de Datos:** Se puede usar una base de datos relacional con replica para manejar las escrituras y lecturas.

- **Cache distribuido:** Implementación de un cache distribuido como Redis para consultas de productos frecuentes.

- **Servicios API:**
    - **Microservicio de consulta:** Este maneja las solicitudes de consulta de productos, verificando con el caché primero o si no existe cache con la base de datos secundaria.

    - **Microservicio de administración:** Responsable de actualizaciones, con un mecanismo de invalidación de caché.

- **Cola de Mensajes:** Uso de una cola como RabbitMQ o Apache Kafka para procesar actualizaciones de productos de manera asíncrona y sincronizar los cambios entre la base de datos principal y las réplicas.

- **Escalabilidad:**
  - Implementación de un balanceador de carga AWS ELB o NGINX.
  - Autoescalado para manejar picos de tráfico en servicios de consulta.
  - Kubernetes en conjunto con ISTIO.

Justificación

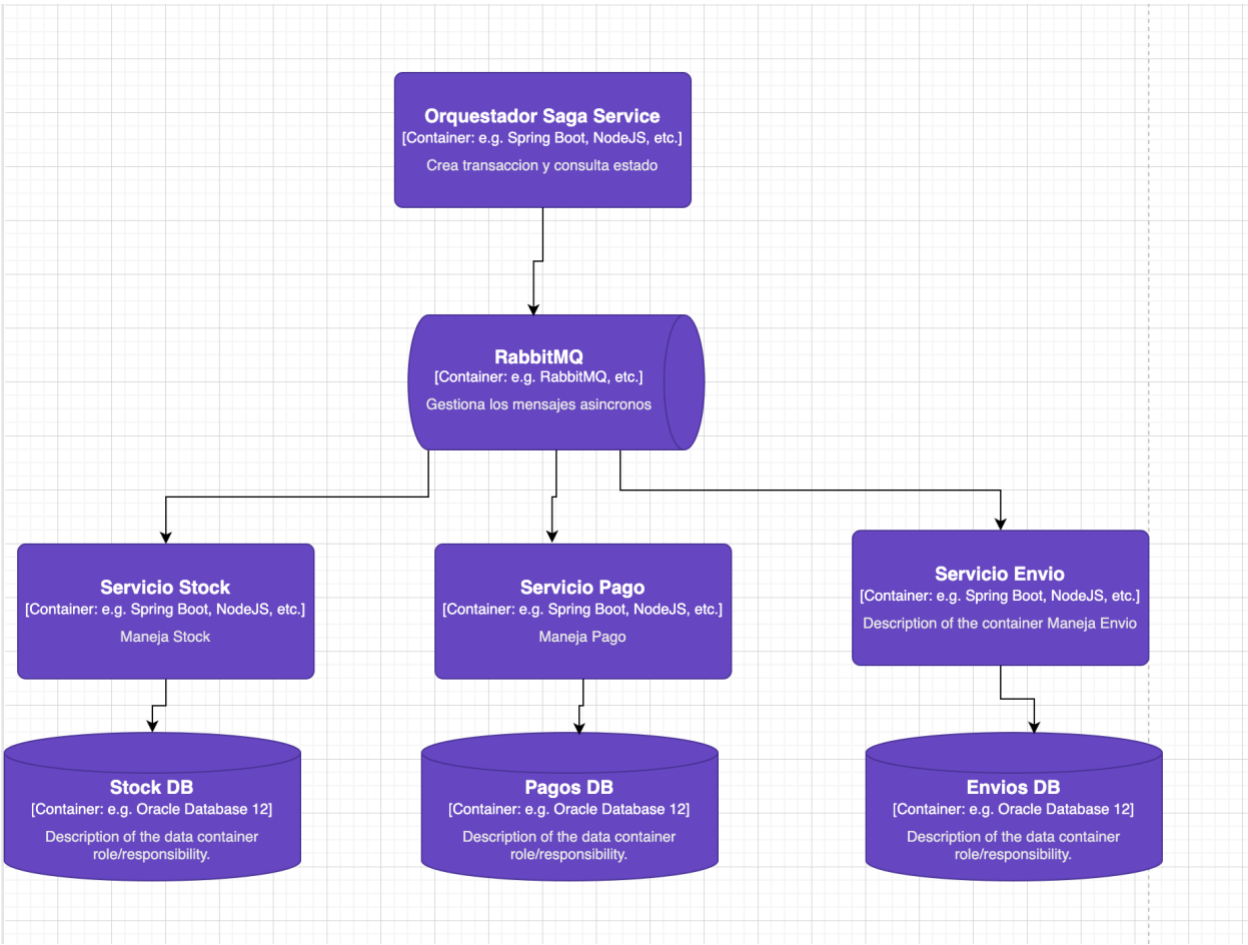
Si separamos en nuestros servicios la lectura/escritura posiblemente usando CQRS reduce la carga en la base de datos primaria.

El cache reduce significativamente la latencia y mejora la experiencia del usuario.

La arquitectura basada en microservicios permite escalar de manera independiente las operaciones de consulta y administración.

2. Transacciones Distribuidas en un Sistema de Microservicios

Propuesta de Arquitectura



- **Saga Pattern:** Implementariamos un servicio con patron coreografía para coordinar transacciones distribuidas.
    - Orquestador central para coordinar los pasos de la transacción.
    - En caso de fallo, ejecutar mecanismos de compensación (Rollback).

- **Consistencia Eventual**: Integramos una cola de mensajeria para manejar los eventos de dominio que se puedan tener y tener una comunicación asincrona.

- **Bases de Datos**: Base de datos transaccional SQLSERVER o ORACLE

- **Seguridad:**
    - Uso de TLS para comunicación entre microservicios.
    - Tokens JWT para autenticación y autorización.

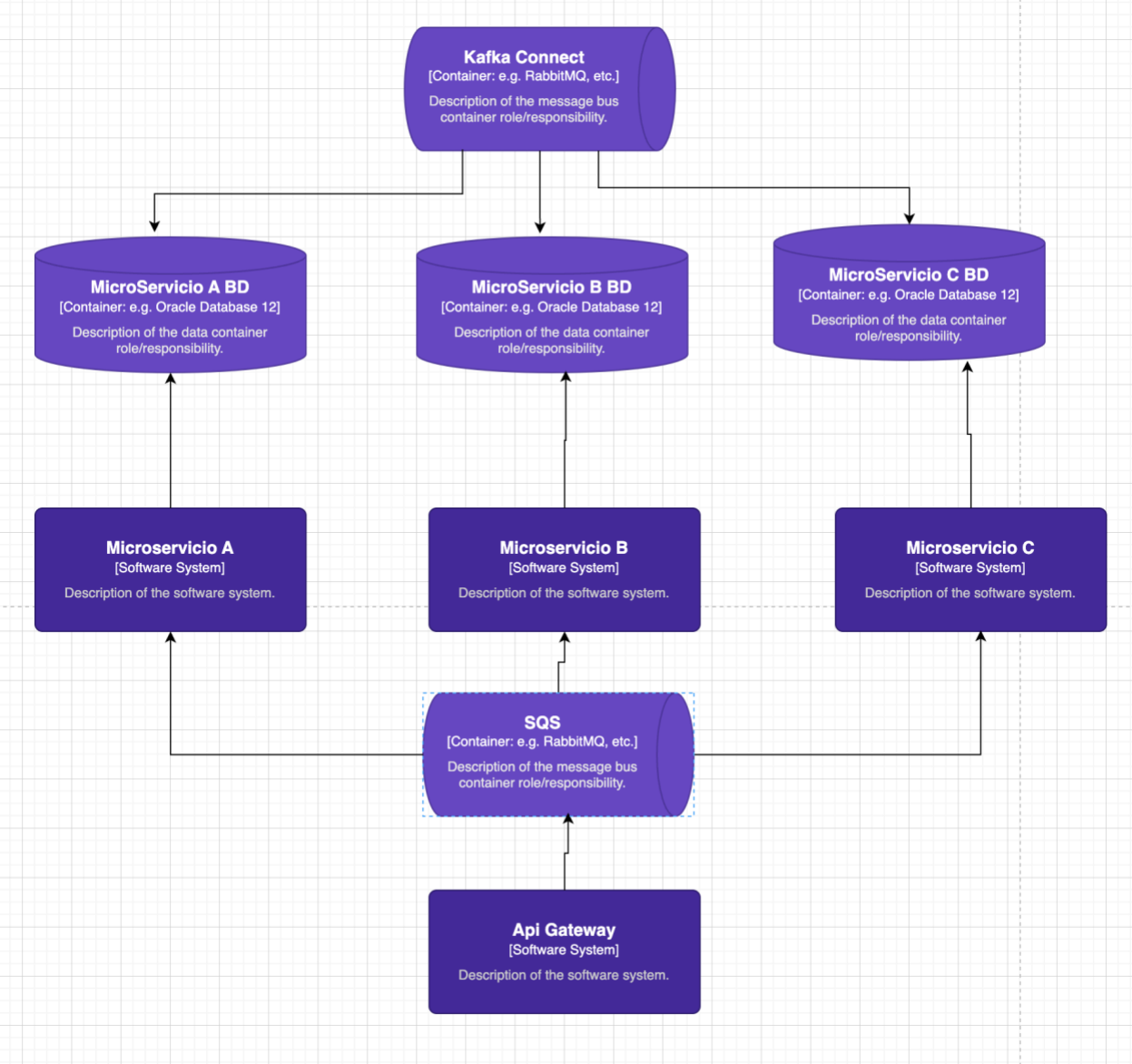
- **Monitoreo:** Implementación de herramientas de monitoreo y tracing distribuidos como Jaeger o Datadog, cloudwatch.

Justificación

- El patrón Saga asegura que las operaciones distribuidas sean consistentes incluso si fallan parcial o totalmente.
- Al tener log de trazabilidad podemos monitorear por posibles fallas para realizar una mejora continua.
- La arquitectura basada en microservicios con mensajes asincrónicos minimiza el acoplamiento entre servicios.

3. Sincronización de Datos entre Servicios con Arquitectura Escalable

Propuesta de Arquitectura



- **Eventos de Dominio:** Uso de eventos de dominio para comunicar cambios de datos relevantes a otros servicios.
    - Ejemplo: Publicar eventos a través de un bus de eventos SQS

- **CQRS :**
    - Separación de los modelos de lectura y escritura.
    - Cada servicio mantiene su propia copia de los datos necesarios sincronizada mediante eventos.

- **Base de Datos Distribuida:** Usar bases de datos como DynamoDB que soporten replicación y escalabilidad horizontal.

- **Kafka Connect:**
  - Para sincronización en tiempo real, además del manejo de datos en caché local.

- **Despliegue Escalable:**
  - Implementación en contenedores con Kubernetes.
  - Autoescalado basado en uso de CPU/memoria o volumen de eventos.

Justificación

- Los eventos de dominio permiten que cada servicio mantenga su propia copia de datos relevantes, asegurando independencia y consistencia eventual.

- CQRS mejora el rendimiento al optimizar lecturas y escrituras.

- El uso de bases de datos distribuidas y herramientas de autoescalado asegura que el sistema pueda manejar grandes volúmenes de datos sin comprometer la sincronización.