Mauricio Campaña | 0984048645| mauriciocamp@gmail.com

# Arquitectura BP

Basado en la información proporcionada en el caso, se recomienda la siguiente arquitectura para la solución:

# Arquitectura General

El presente documento de arquitectura tiene como objetivo describir la estructura y diseño del sistema propuesto para una plataforma bancaria moderna basada en microservicios. La arquitectura está diseñada para garantizar la escalabilidad, alta disponibilidad, seguridad y eficiencia operativa, alineándose con los requisitos de un entorno bancario altamente regulado.

A lo largo del documento, se detallarán los componentes clave del sistema, incluyendo la autenticación segura mediante OAuth 2.0, el manejo de datos críticos y auditoría con bases de datos relacionales y no relacionales, y el uso de tecnologías avanzadas como Edge Computing y Content Delivery Networks (CDN) para minimizar la latencia y mejorar la experiencia del usuario final.

Además, se incluirán soluciones para garantizar la continuidad del servicio mediante estrategias de replicación de bases de datos y sistemas de failover, así como el uso de herramientas de monitoreo y auto-healing para optimizar la operación y mitigar fallos de infraestructura.

Este diseño está pensado para soportar la evolución futura de la plataforma, permitiendo su fácil mantenimiento y expansión, a la vez que asegura el cumplimiento de normativas y estándares de la industria financiera.

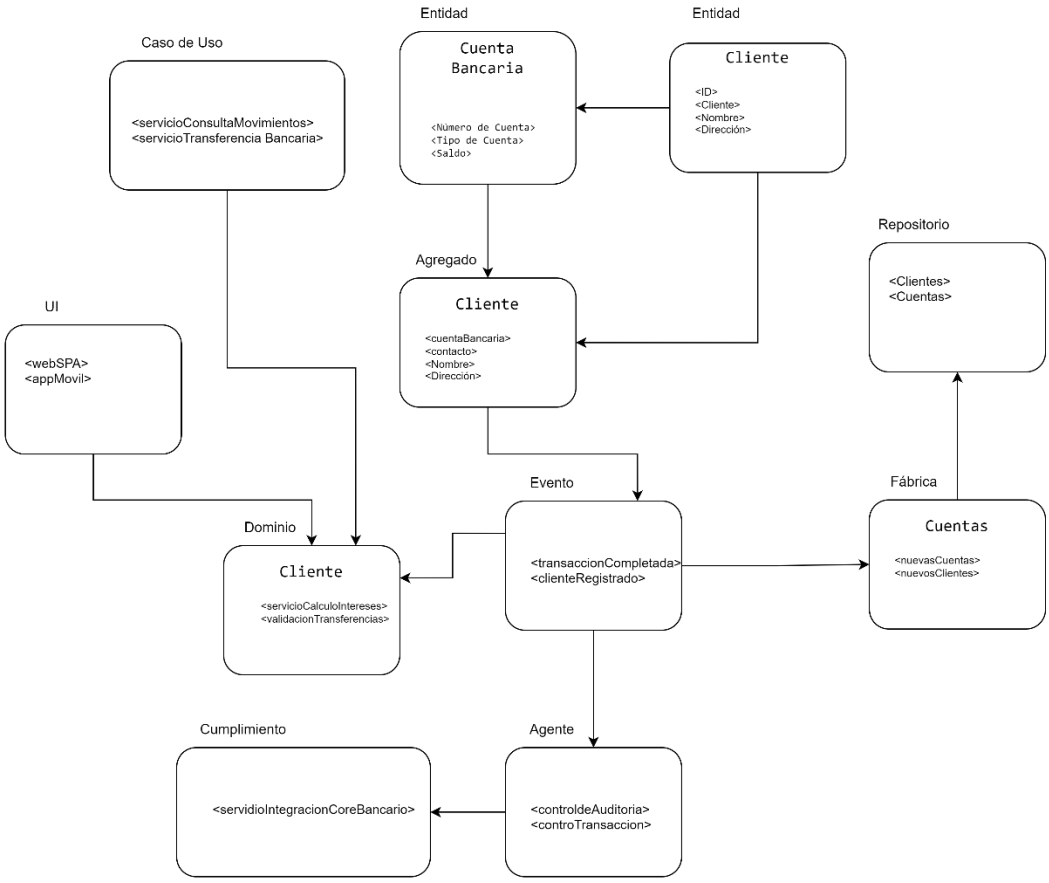
## 1. Segmentación de responsabilidades y desacoplamiento:

Para la implementación a gran escala de esta solución, se propone el uso de dos arquitecturas clave: Domain-Driven Design (DDD), que facilita la abstracción de los conceptos del negocio y los traduce a la lógica interna de la aplicación, permitiendo una mejor alineación entre los requisitos de negocio y el desarrollo técnico; y la Arquitectura Hexagonal, que asegura la independencia entre los servicios, promoviendo una interacción modular y desacoplada, lo que favorece la escalabilidad, mantenibilidad y flexibilidad del sistema frente a cambios en la infraestructura o las integraciones externas.

## Arquitectura basada en Domain-Driven Design (DDD)

La arquitectura basada en Domain-Driven Design (DDD) se centra en modelar el sistema de software en torno a los conceptos y reglas del dominio del negocio. Esta arquitectura garantiza que la lógica de negocio sea el foco principal, permitiendo que el desarrollo refleje fielmente los procesos empresariales y facilitando la mantenibilidad y escalabilidad del sistema. A continuación, se describen los principales componentes de la arquitectura DDD y su implementación en un sistema bancario.

### DDD Diagrama Base




**1. Contextos Delimitados (Bounded Contexts)**

Se identifican dos contextos delimitados clave:

- Contexto de Gestión de Clientes:

Maneja todas las operaciones relacionadas con los clientes, como su creación, actualización y validación. Incluye las entidades y servicios relacionados con los clientes y su información.

- Entidades principales: Cliente
- Servicios: Cálculo de intereses, validación de transferencias

- Eventos de dominio: Cliente registrado

- Contexto de Gestión de Cuentas Bancarias:
  Gestiona todas las operaciones relacionadas con las cuentas bancarias, como las transferencias y consultas de movimientos. Incluye las reglas de negocio para la administración de cuentas.
    - Entidades principales: Cuenta Bancaria
    - Servicios: Transferencias bancarias, consulta de movimientos
    - Eventos de dominio: Transacción completada

**2. Entidades y Objetos de Valor**

Las entidades clave dentro de los contextos son:

- Cliente: Representa a los usuarios del sistema bancario, con atributos como ID, Nombre, Dirección, y Cuentas Bancarias asociadas.
- Cuenta Bancaria: Entidad que representa las cuentas de los clientes, con atributos como Número de Cuenta, Tipo de Cuenta y Saldo.
- Objeto de Valor: El Monto se considera un objeto de valor que define la cantidad a transferir y está asociado con reglas de negocio, como no permitir valores negativos.

**3. Servicios de Dominio**

Los servicios de dominio encapsulan la lógica del negocio que no pertenece directamente a una entidad. Se identifican los siguientes servicios:

- Servicio de Transferencia Bancaria: Realiza la transferencia de fondos entre cuentas. Valida el saldo y actualiza el estado de las cuentas.
- Servicio de Consulta de Movimientos: Proporciona un historial de movimientos para una cuenta bancaria.
- Servicio de Cálculo de Intereses: Realiza cálculos periódicos sobre los saldos de los clientes y sus cuentas.
- Validación de Transferencias: Asegura que las transacciones cumplen con las reglas del negocio antes de proceder.

**4. Repositorios**

Se deben implementar repositorios que manejen la persistencia de las entidades sin exponer detalles técnicos de las bases de datos. Los principales repositorios son:

- Repositorio de Clientes: Administra la persistencia y recuperación de los datos de clientes.
- Repositorio de Cuentas Bancarias: Administra las operaciones de almacenamiento y consulta de las cuentas bancarias.

**5. Eventos de Dominio**

Los Eventos de Dominio son cruciales para mantener la integridad del sistema y permitir la interacción con otros servicios sin acoplamiento directo. Algunos eventos relevantes son:

- Cliente Registrado: Disparado cuando se completa el registro de un cliente.
- Transacción Completada: Se genera cuando una transferencia de fondos es realizada exitosamente entre dos cuentas.

Estos eventos pueden ser consumidos por servicios externos como auditoría o notificaciones.

**6. Auditoría y Notificaciones**

El sistema de auditoría recopila eventos clave del sistema como Transacción Completada y Cliente Registrado, garantizando que las actividades sean monitoreadas y cumplan con las normativas financieras. Los eventos también pueden ser utilizados para notificar a los usuarios sobre el estado de sus transacciones.

**7. Adaptadores de Entrada y Salida**

El sistema interactúa con el mundo exterior a través de Adaptadores de Entrada (como Web SPA y Aplicaciones Móviles) y Adaptadores de Salida (como servicios externos de auditoría y bases de datos).

- Adaptadores de Entrada: Proveen interfaces para que los usuarios interactúen con el sistema.
- Adaptadores de Salida: Permiten la comunicación con servicios externos, como la base de datos o el core bancario.

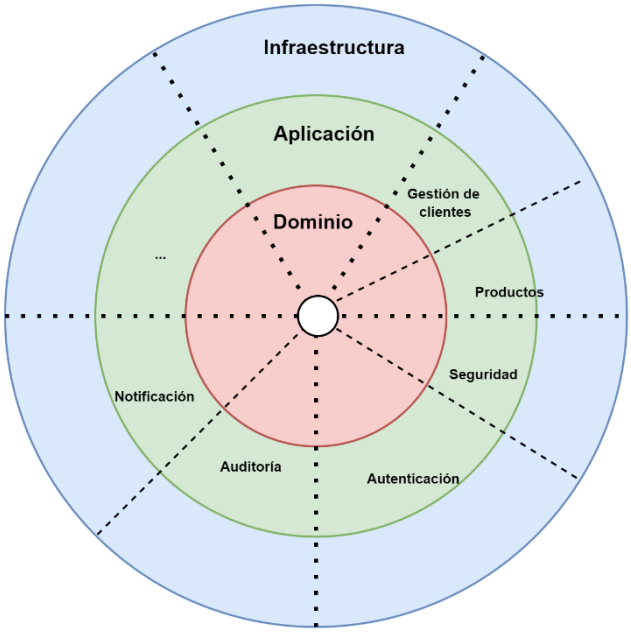
Se recomienda utilizar los siguientes patrones de diseño:

- CQRS (Command Query Responsibility Segregation) para separar las operaciones de lectura y escritura a la Base de Datos, lo que permite escalar los servicios de manera independiente.

- Event-Driven Architecture (EDA): para desacoplar componentes mediante la publicación y suscripción de eventos por medio de un Broker de Eventos.

## Arquitectura Hexagonal

Este enfoque permite desacoplar el núcleo del dominio de las interfaces externas, lo que facilita el mantenimiento, la flexibilidad y la adaptabilidad del sistema frente a cambios tecnológicos.




1. **Núcleo del Dominio (Core)**

   **Entidades:**

   - Cliente y Cuenta Bancaria. Estas entidades representan el núcleo de las operaciones bancarias.

   **Servicios de Dominio:**

   - Servicio de Transferencia Bancaria: Realiza las transferencias entre cuentas, validando el saldo disponible y aplicando las reglas de negocio.
   - Consulta de Movimientos: Ofrece un historial de transacciones.
   - Cálculo de Intereses: Realiza cálculos sobre saldos de cuentas de clientes.

   **Eventos de Dominio:**

- Transacción Completada y Cliente Registrado, que pueden ser utilizados para disparar acciones en otros servicios (auditoría, notificaciones).

2. **Puertos (Ports)**
   - **Puertos de Entrada:** Son interfaces que exponen las operaciones del dominio hacia los adaptadores de entrada (interfaces de usuario o servicios externos).

     ITransferService, IAccountQueryService, ICustomerService.

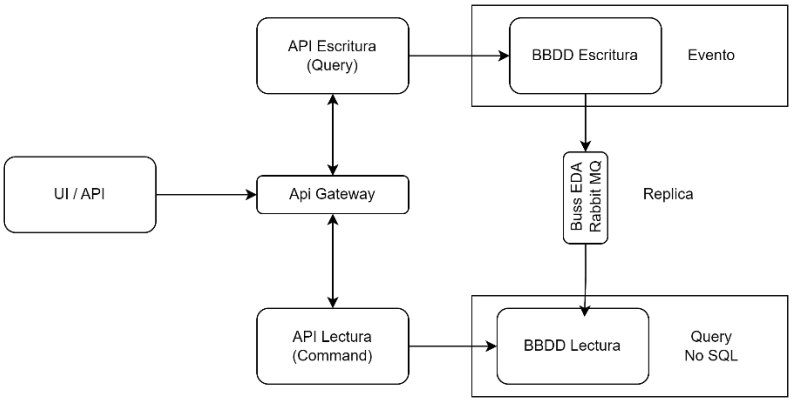
   - **Puertos de Salida: Son** interfaces que permiten que el núcleo interactúe con las dependencias externas, como bases de datos, servicios de auditoría o sistemas de mensajería.

     IAuditRepository, INotificationService, ITransactionRepository.

3. **Adaptadores (Adapters)**
   - **Adaptadores de Entrada (Interfaces de Usuario):**
     - Web SPA y Aplicaciones Móviles: Adaptadores que manejan las solicitudes del usuario a través de interfaces gráficas. Se conectan a los puertos de entrada para realizar operaciones de negocio, como consultar saldos, realizar transferencias, etc.
     - API REST: Un adaptador que expone los servicios del dominio mediante HTTP, permitiendo a sistemas externos o clientes consumir las operaciones de la aplicación.
   - **Adaptadores de Salida (Infraestructura):**
     - Base de Datos (PostgreSQL/MSSQlServer/Mongo): Implementa los repositorios que almacenan y recuperan entidades como Cliente y Cuenta Bancaria.
     - Auditoría: Implementa el puerto de salida *IAuditRepository* para registrar eventos importantes como transacciones completadas.
     - Sistema de Notificaciones: Implementa el puerto *INotificationService* para enviar alertas o mensajes a los usuarios sobre transacciones y cambios en su cuenta.
     - Core Bancario Externo: Un servicio externo que se integra mediante un adaptador para sincronizar operaciones bancarias.

## 2. Desacoplamiento y comunicación entre servicios:

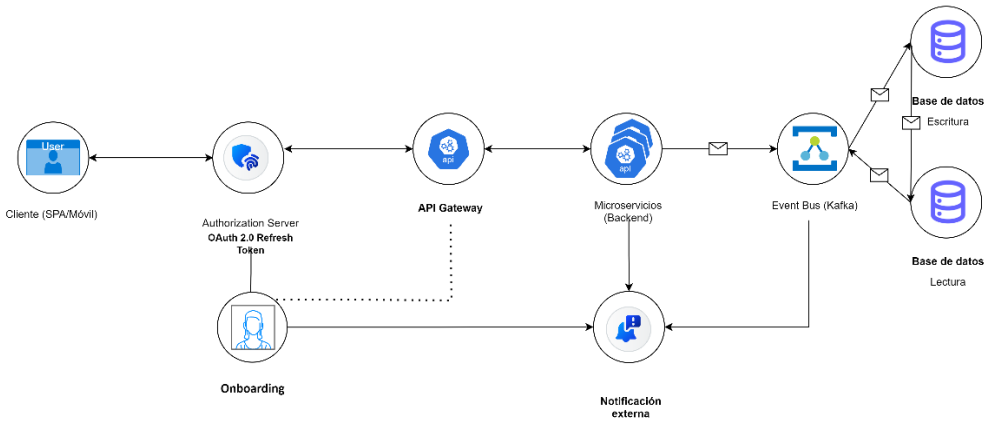
Utiliza API REST para la comunicación sincrónica entre servicios y eventos con Pub/Sub o Message Queues (como RabbitMQ o Kafka) para la comunicación asíncrona. El Objetivo es bajar la dependencia entre servicios y permite mayor flexibilidad.



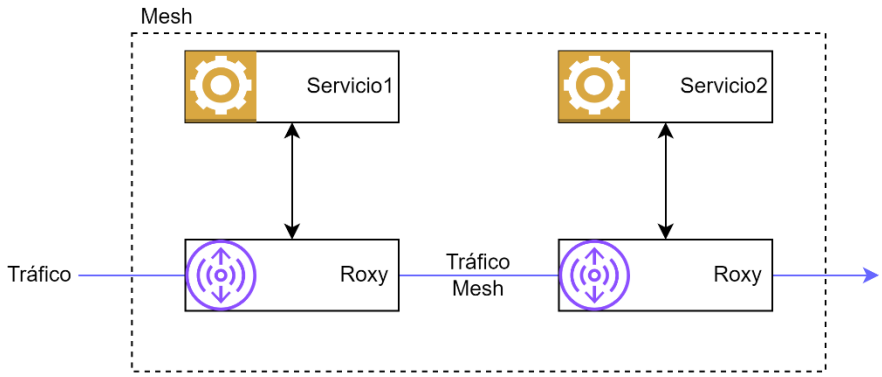

# Integración con Sistemas Externos

## 1. Conexión con Core Bancario y sistemas externos:

- Utiliza un API Gateway que permita la transformación y enrutamiento de mensajes entre el sistema Core y el sistema independiente de cliente. Esto desacopla los sistemas y facilita la integración con otros sistemas futuros.




- Arquitectura basada en un Service Mesh para gestionar las comunicaciones, resiliencia y seguridad entre microservicios.




## 2. Métodos de integración bancarios:

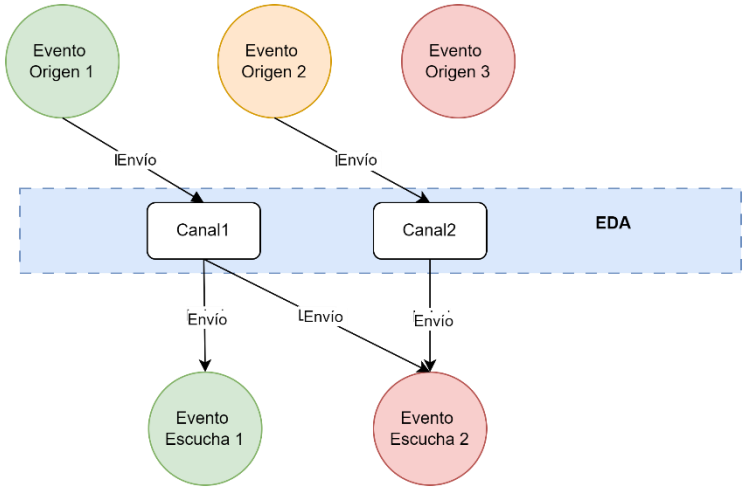
- Utilizar API REST para la exposición de servicios con datos del cliente, transacciones y movimientos, y gRPC para aquellas operaciones que requieran bajo tiempo de respuesta y alta eficiencia, como los pagos interbancarios.

- Dependiendo de los casos de uso donde REST es más eficiente frente a gRPC.

  **REST** es ideal para clientes como aplicaciones móviles y aplicaciones web (SPA) porque está basado en HTTP/1.1, el cual es más compatible con los navegadores web. Por otro lado, **gRPC** es mucho más eficiente que REST para la comunicación entre microservicios en un entorno controlado (como en una red privada o dentro de un clúster Kubernetes). Al usar HTTP/2, soporta multiplexación, compresión de mensajes y una comunicación más rápida y eficiente, esto puede ser aplicables en servicios de gestión de cuentas y el servicio de procesamiento de transacciones.

# 3. Comunicación y sincronización con sistemas de notificaciones:

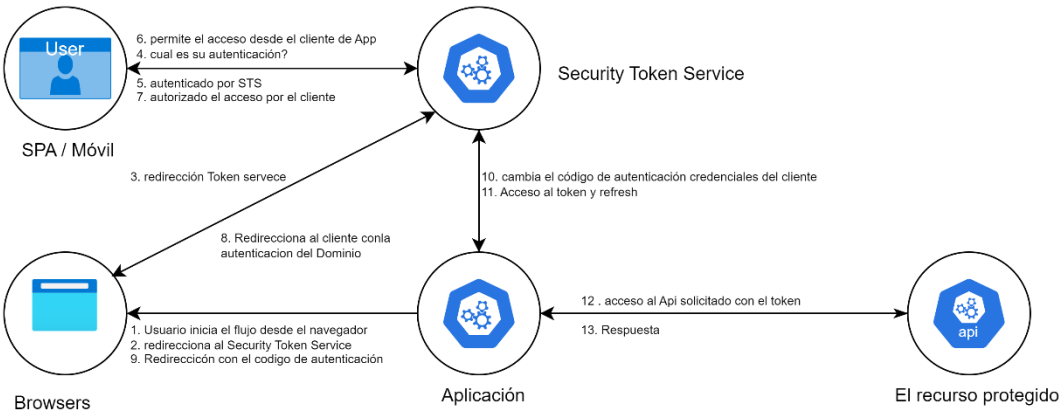
- Considero que se debe implementar una arquitectura de eventos con Pub/Sub. Cada vez que una transacción o movimiento ocurra, se publica un evento que puede ser consumido por diferentes sistemas de notificación.
- Usando Kafka o RabbitMQ se puede implementar un Event Bus upara garantizar que los mensajes de notificación sean entregados y en su defecto tomados desde consumidor.




# Autenticación y Seguridad

## 1. Flujo de autenticación Oauth2.0 recomendado:

- Se recomienda Authorization Code Flow ya que es el más seguro para aplicaciones que necesitan acceso a un recurso protegido desde un cliente web o móvil. Este flujo minimiza el riesgo de exposición de credenciales.




Este flujo consta de los siguientes pasos:

**Solicitud de autorización:**

El cliente (SPA o aplicación móvil) redirige al usuario al servidor de autorización. El usuario inicia sesión y otorga el permiso a la aplicación para acceder a los recursos protegidos.

**Código de autorización:**

Una vez que el usuario otorga el permiso, el servidor de autorización redirige de nuevo al cliente con un código de autorización temporal.

**Intercambio del código de autorización por tokens:**

El cliente envía el código de autorización al servidor de autorización (junto con las credenciales del cliente, como el client_id y el client_secret en aplicaciones seguras) y solicita un token de acceso. En este paso, el servidor de autorización verifica la validez del código.

**Token de acceso:**

Si el código es válido, el servidor de autorización devuelve un token de acceso (y opcionalmente un token de actualización) al cliente.

**Acceso al recurso protegido:**

El cliente utiliza el token de acceso para realizar solicitudes autenticadas a la API o recurso protegido en nombre del usuario.

### 2. Onboarding con reconocimiento facial:

- Utilizar un servicio de validación biométrica como Amazon Rekognition o Microsoft Azure Face API para realizar el reconocimiento facial. Estos servicios proporcionan APIs robustas y seguras para manejar datos biométricos.
- Documentación: Integrar estos servicios en el flujo de autenticación mediante APIs que garantizan el cumplimiento de normativas como GDPR y PSD2.

### 3. Protección de datos biométricos:

- Tomar en cuenta que: En el proceso de registro de cuentas y clientes se debe mantener una entidad por cada usuario del sistema, en donde se registra la autorización del cliente para el uso consensuado de su información, esta entidad debe ser mantenida por el perfil del gestor de datos personales que se designa por cada organización según la normativa vigente. La entidad debe gestionar reportes, notificaciones y solicitudes de eliminación de información según el cliente lo solicite.

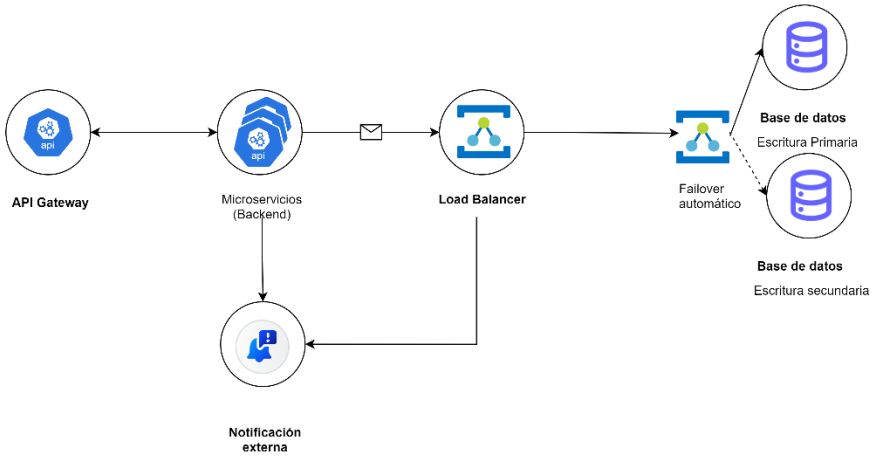
- Cifrar los datos biométricos en tránsito usando TLS y en reposo usando algoritmos como AES-256. Almacenar los datos en sistemas conformes con normativas de protección de datos.

# Infraestructura y Alta Disponibilidad

## Alta disponibilidad y tolerancia a fallos:

- Arquitectura en la nube utilizando AWS o Azure, con servicios como Auto Scaling, Load Balancing y Multi-AZ deployments para garantizar alta disponibilidad.

**Descripción del flujo de la arquitectura:**




**Cliente -> API Gateway -> Microservicios:**

Los clientes (SPA o móvil) envían solicitudes a través de un API Gateway, que enruta estas solicitudes a los microservicios adecuados. El API Gateway actúa como punto de control para la autenticación y balanceo de carga.

**Microservicios -> Base de Datos (Master):**

Los Microservicios se encargan de la lógica de negocio y las operaciones de lectura/escritura. Para las escrituras (por ejemplo, nuevas transacciones), las solicitudes se envían a la Base de Datos Principal (Master).
Para operaciones de lectura (como consultar el historial de transacciones), las solicitudes pueden ser redirigidas a las Read Replicas, aliviando la carga de la base de datos principal.

**Replicación de Base de Datos:**

La Base de Datos Principal replica automáticamente los datos hacia varias Read Replicas. Estas réplicas pueden estar ubicadas en diferentes Zonas de Disponibilidad (AZs) para garantizar la disponibilidad de los datos.
Los datos se replican de manera asincrónica (para leer réplicas) o sincrónica (en entornos críticos donde la consistencia es fundamental).

**Failover Automatizado:**

En caso de que la Base de Datos Principal falle, un sistema de failover automático detecta el fallo y promueve una de las Read Replicas para convertirse en la nueva base de datos principal (master). Esto garantiza la continuidad del servicio sin interrupciones mayores.
El Load Balancer redirige automáticamente las solicitudes de escritura hacia la nueva base de datos principal.

**Multi-AZ Cluster y Almacenamiento Distribuido:**

Todos los componentes críticos, incluidas las bases de datos, se replican en múltiples zonas de disponibilidad (AZs). Esto asegura que si una AZ completa falla (por ejemplo, un centro de datos), el sistema sigue funcionando en otras AZs.
El almacenamiento distribuido permite que los backups y restauraciones se realicen en múltiples ubicaciones geográficas, permitiendo una rápida recuperación ante desastres globales.

**Sistemas de Backup y Restauración:**

Se realizan backups automáticos y programados de la base de datos. Los respaldos se almacenan en un almacenamiento distribuido (por ejemplo, S3 en AWS o Azure Blob Storage), asegurando que los datos estén disponibles para una recuperación completa si es necesario.

**Monitoreo y Alertas:**

Prometheus y otras herramientas de monitoreo rastrean continuamente el rendimiento del sistema, incluyendo la latencia de las bases de datos, errores en microservicios y uso de recursos.
En caso de anomalías, se activan alertas que informan al equipo de administración o incluso disparan procesos de auto-escalado o auto-healing.

## 2. Baja latencia:

Se recomienda usar una combinación de CDN para distribuir el contenido estático de la SPA, y Edge Computing (por ejemplo, AWS Lambda@Edge) para acercar la lógica de la aplicación a los usuarios y reducir la latencia.

**Entrega de contenido estático con CDN:**

La SPA (o aplicación móvil) utiliza un CDN como Amazon CloudFront o Azure CDN para distribuir contenido estático (HTML, CSS, JS, imágenes, etc.) desde el servidor más cercano al usuario. Esto reduce drásticamente el tiempo de carga inicial de la aplicación, ya que las peticiones no tienen que viajar hasta el servidor central.

**Procesamiento de lógica de aplicación con Edge Computing:**

Se utiliza AWS Lambda@Edge o Azure Functions en Edge para mover parte de la lógica de la aplicación (como autenticación, autorización, o manejo de caché dinámico) más cerca de los usuarios. Estos puntos de presencia (PoPs) están distribuidos globalmente y permiten realizar cálculos o procesamiento sin necesidad de llegar al servidor central.

Cuando el cliente solicita acceso a un recurso protegido, Lambda@Edge verifica la autenticación en el PoP más cercano, mejorando el tiempo de respuesta antes de redirigir la solicitud a la API central.

**Distribución de microservicios en múltiples regiones:**

Los microservicios backend se despliegan en varias regiones geográficas utilizando servicios de nube como AWS EC2, ECS, Azure App Service o Kubernetes. Cada región puede tener una réplica completa de los servicios necesarios para el funcionamiento de la aplicación. Esto permite que las solicitudes dinámicas del cliente sean procesadas en la región más cercana.

Un usuario en Europa puede conectarse a microservicios desplegados en una región cercana (como EU-West) en lugar de que su solicitud tenga que viajar a una región lejana como US-East.

**Base de Datos Distribuida Globalmente:**

Una base de datos distribuida como Amazon Aurora Global Database o Azure Cosmos DB se replica en múltiples regiones. Esto asegura que las operaciones de lectura/escritura de datos tengan baja latencia, ya que las solicitudes se procesan en la región más cercana.
Las operaciones de escritura se pueden sincronizar entre regiones, y en caso de un fallo en una región, otra región puede asumir la escritura.
Las transacciones bancarias o consultas de historial se gestionan en la base de datos replicada más cercana al usuario para garantizar que las operaciones sean rápidas.

**Global Accelerator o Traffic Manager:**

AWS Global Accelerator o Azure Traffic Manager enrutan las solicitudes del cliente al endpoint más cercano utilizando la ubicación geográfica del usuario y la latencia mínima. Este servicio optimiza el recorrido de las solicitudes dinámicas para que lleguen a la región de microservicios más cercana.
Ejemplo de uso: Un usuario en Asia puede ser dirigido a los microservicios y bases de datos desplegados en una región de Asia-Pacífico, mientras que un usuario en América será dirigido a una región de US-East o US-West.

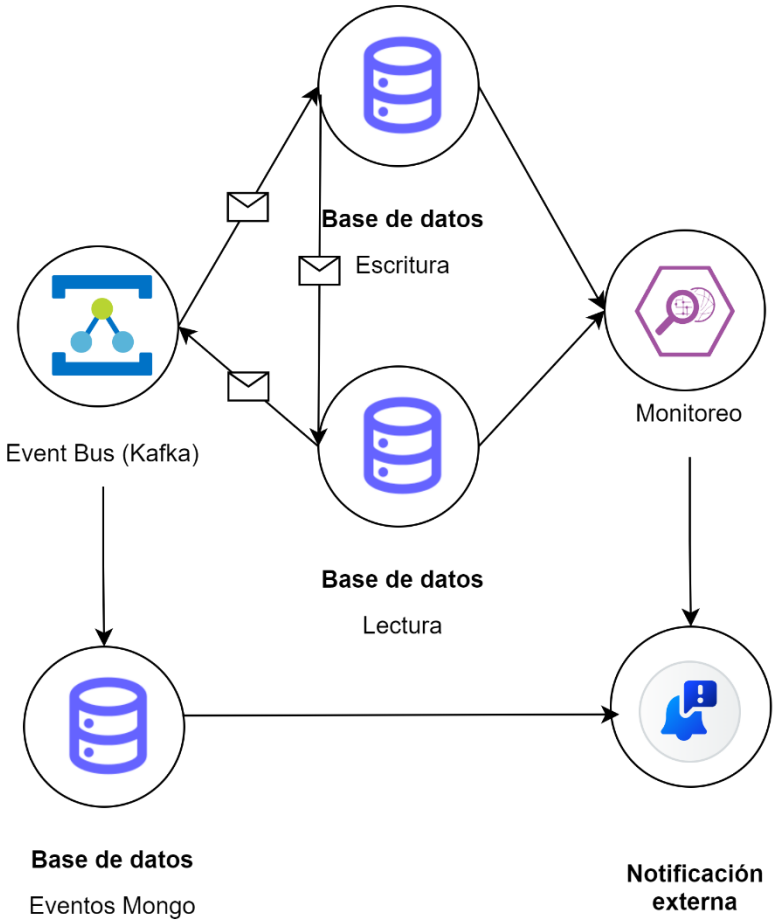
**DNS Geolocalizado:**

Un DNS geolocalizado asegura que los usuarios sean dirigidos a las versiones regionales de la aplicación o microservicios de forma automática. Esto garantiza que incluso en el nivel de red, las peticiones viajen la menor distancia posible.
Ejemplo de uso: Cuando un usuario accede a la aplicación desde su navegador o móvil, el DNS resuelve la dirección IP del servidor más cercano a su ubicación geográfica.

# Arquitectura de Datos y Auditoría

## 1. Arquitectura de acceso a datos:

Utilizar servicios de bases relacionales como PostgreSQL o MSSQLServer en kubernetes, para la gestión de auditorías, junto con bases de datos no relacionales como MongoDB para logs de alta frecuencia y almacenamiento masivo de datos no estructurados.




**Cliente (SPA/Móvil):**

El cliente genera eventos que son almacenados tanto en la base de datos relacional (para auditoría) como en la no relacional (para logs en tiempo real).

**Microservicios de Auditoría:**

Estos servicios capturan eventos críticos (como transacciones, accesos no autorizados) y los escriben en la base de datos relacional.

**Logs de Alta Frecuencia:**

Los eventos del sistema y registros de aplicaciones son enviados directamente a MongoDB para un almacenamiento rápido y no estructurado.
Bases de Datos:

PostgreSQL/MySQL (Relacional): Almacena los registros de auditoría estructurados.
MongoDB (No Relacional): Almacena logs de alta frecuencia y datos no estructurados.

**Control de Acceso Basado en Roles (RBAC):**

Gestiona el acceso seguro a las bases de datos, asegurando que solo los usuarios autorizados puedan consultar o modificar los registros de auditoría y logs.

### 2. Escalabilidad y consultas eficientes:

Para la solución de debe implementar sharding y replicación en la base de datos para manejar grandes volúmenes de datos. Utilizar indices y particionamiento para optimizar las consultas.

# Monitoreo y Excelencia Operativa

### 1. Monitoreo y auto-healing:

- Para el monitoreo se puede usar herramientas como Prometheus para recolectar métricas del sistema y Grafana para visualizarlas. Utilizar auto-scaling y políticas de auto-healing basadas en umbrales de recursos por cada servicio.

# Frontend y Aplicación Móvil

### 1. Frameworks recomendados:

- Considero que **React** para la SPA es una buena opción debido a su flexibilidad, rendimiento y ecosistema maduro.
- Para la **aplicación móvil**, usar Flutter debido a su capacidad para crear aplicaciones nativas de alto rendimiento desde un solo código base.

### 2. Autenticación biométrica y manejo de errores:

- Implementar autenticación biométrica en la aplicación móvil utilizando Face ID o Fingerprint API nativa de cada plataforma, la premisa es manejar estos tres enfoques:

  **Experiencia de usuario fluida:** Permite que el usuario continúe accediendo a la aplicación, aunque falle el componente de onboarding o autenticación biométrica.
  **Seguridad robusta:** Se mantienen las medidas de seguridad adecuadas con opciones de verificación secundaria en caso de pérdida o fallos en los métodos de autenticación primaria.

  **Adaptación a múltiples escenarios:** El usuario puede elegir entre diferentes métodos de autenticación, según sus preferencias o las circunstancias del fallo biométrico.

### Flujo General de Autenticación Biométrica:

Usuario abre la aplicación móvil y no puede iniciar autenticación por medio del componente de onboarding, la aplicación realiza el siguiente flujo desde el Front.

i. La aplicación solicita la autenticación biométrica utilizando Face ID o Fingerprint API.
ii. El sistema operativo (iOS o Android) verifica los datos biométricos.
iii. Si la autenticación biométrica es exitosa:
iv. El usuario es autenticado y accede a la aplicación.
v. Si la autenticación falla:
vi. Se activa el flujo de fallback para manejo de errores.

En caso de que el reconocimiento biométrico falle, se ofrece desde el front alternativas como usuario y contraseña o PIN solicitado en el registro del cliente.