Arquitectura de Integración para la Modernización Bancaria

Nombre: Gabriel Salgado

Fecha: 14 de noviembre de 2025

Asunto: Propuesta de Arquitectura de Integración para la Modernización de Sistemas

---

1. Resumen Ejecutivo

Esta propuesta detalla una arquitectura de integración moderna, robusta y escalable diseñada para la modernización de los sistemas del banco. La solución se centra en un enfoque **híbrido (Multicore)**, habilitado por una **arquitectura orientada a eventos (EDA)** y un diseño **API-Led** basado en los dominios de servicio del estándar **BIAN**.

La arquitectura utiliza **Azure API Gateway** como puerta de enlace, microservicios desarrollados en **Spring Boot**, **Keycloak** como proveedor de identidad central (IAM) y **Grafana** para el monitoreo y la observabilidad. El objetivo es permitir la coexistencia del core bancario tradicional y el nuevo core digital, garantizando agilidad, baja latencia, alta disponibilidad (HA) y cumplimiento normativo.

2. Diseño de Arquitectura de Alto Nivel

La arquitectura propuesta se basa en una plataforma de integración híbrida que utiliza **Azure API Gateway (APIM)** como puerta de enlace única y un **Bus de Eventos Empresarial** (ej. Apache Kafka) como sistema nervioso central para la comunicación asíncrona.

Alineación con BIAN

El estándar BIAN (Banking Industry Architecture Network) se utiliza para definir el panorama de servicios. En lugar de crear APIs monolíticas, diseñamos microservicios (en **Spring Boot**) que se alinean con los **Dominios de Servicio de BIAN** (Service Domains).

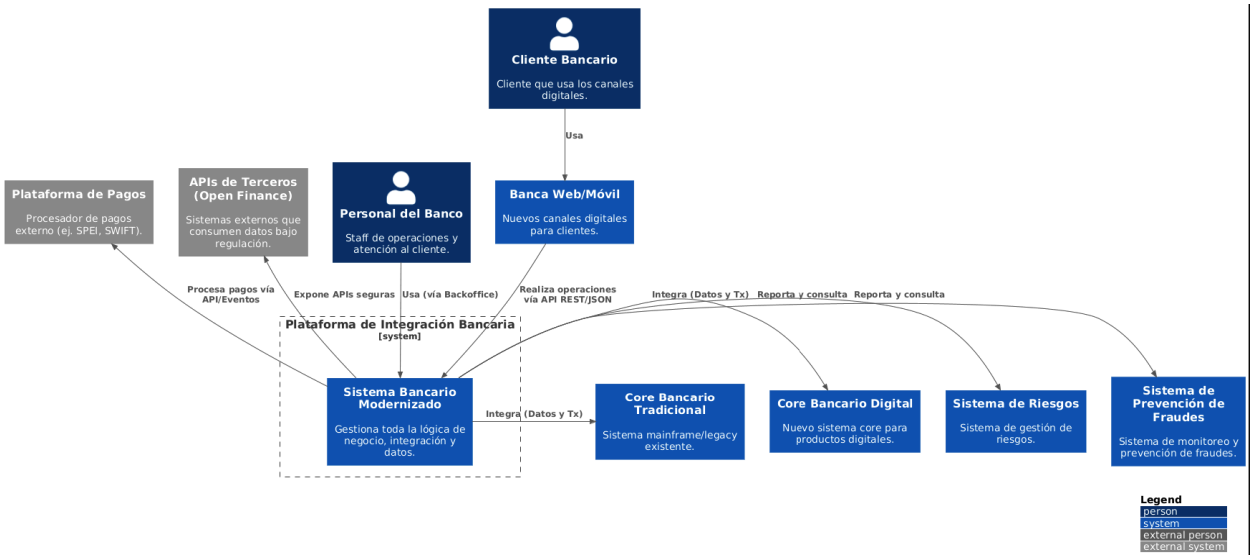
- **Ejemplo:** En lugar de una API de "Cliente", tendremos dominios BIAN como `Customer Offer`, `Customer Access`, y `Party Lifecycle Management`.

- **Beneficio:** Esto asegura un lenguaje común, reutilización y una clara separación de incumbencias.

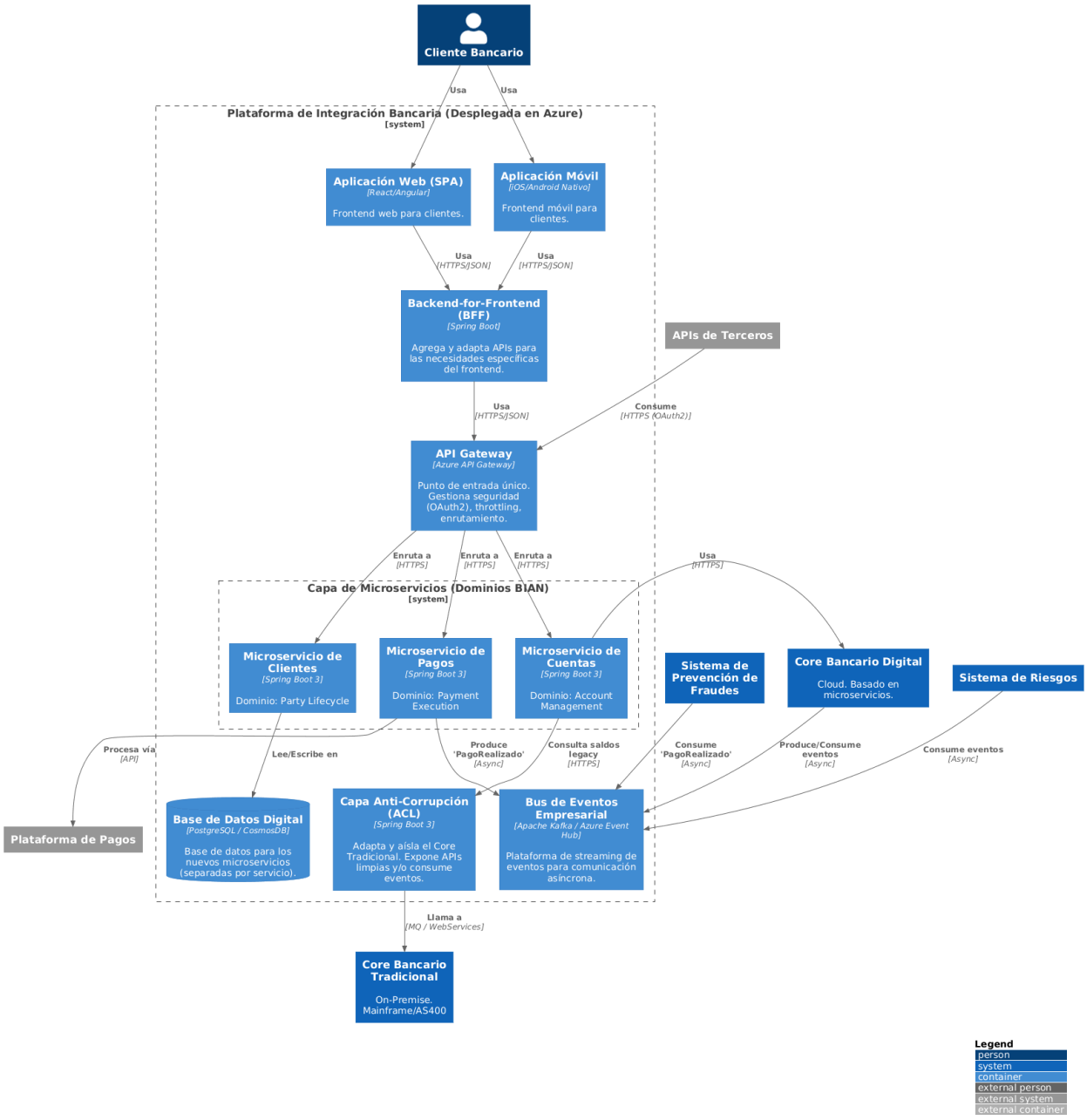
Diagramas C4

A continuación, se presentan los diagramas C4 que describen la arquitectura.

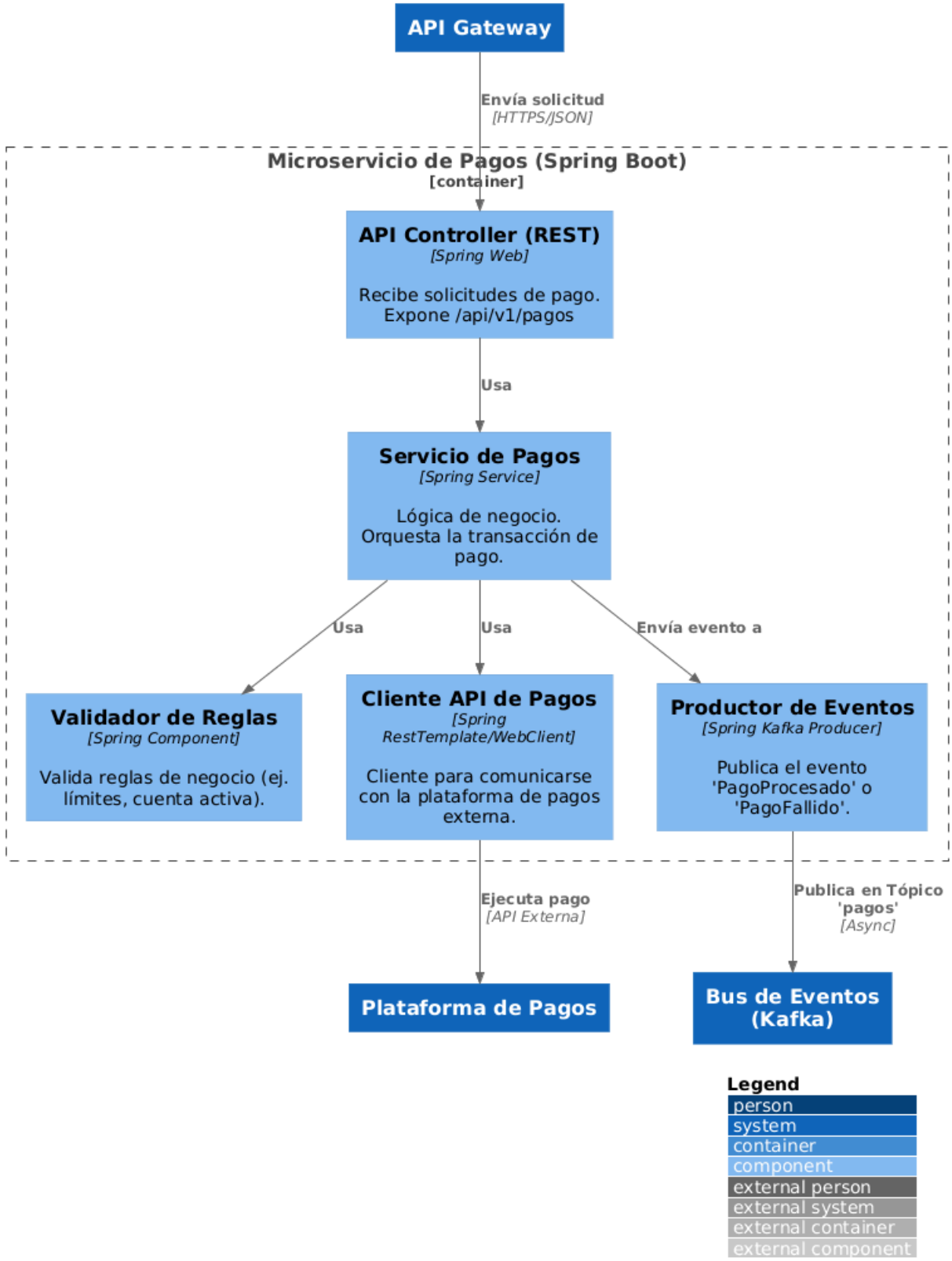
Nivel 1: Diagrama de Contexto del Sistema



Nivel 2: Diagrama de Contenedores



Nivel 3: Diagrama de Componentes



3. Patrones de Integración y Tecnologías

Se utilizará una combinación de patrones para garantizar la eficiencia, el desacoplamiento y la capacidad de evolución del sistema.

Patrón de Integración	Dónde se aplica	Por qué se aplica (Justificación)
API-Led Connectivity	En toda la arquitectura.	Se definen 3 capas de APIs: System APIs (conectan a los cores), Process APIs (orquestan lógica de negocio), y Experience APIs (exponen datos al BFF).
Strangler Fig (Estrangulador)	En la migración del Core Tradicional.	El API Gateway y el ACL actúan como una fachada. Las nuevas solicitudes se enrutan al Core Digital. Las antiguas se "estrangulan" gradualmente.
Anti-Corruption Layer (ACL)	Contenedor acl_legacy.	Aísla el Core Tradicional (protocolos SOAP/MQ) del resto de la arquitectura moderna (REST/JSON/Eventos).
Event-Driven (Publicar/Suscribir)	Kafka y Microservicios.	Para operaciones asíncronas. Ej: un pago genera un evento. Los sistemas de Fraude y Riesgos se suscriben a este evento y reaccionan en tiempo real.
Backend-for-Frontend (BFF)	Contenedor bff.	Optimiza la carga de datos para los frontales (Web/Móvil).
CQRS (Segregación de Comandos y Consultas)	En microservicios críticos (ej. Cuentas).	Para optimizar las lecturas. Las escrituras (comandos) van a los Cores, las lecturas (consultas) se sirven desde una réplica de lectura optimizada.

Tecnologías Clave:

- **Cloud:** *Microsoft Azure* (para escalabilidad y servicios gestionados).
- **Contenedores:** *Azure Kubernetes Service (AKS)* para orquestar los microservicios.
- **API Gateway:** *Azure API Gateway (APIM).*
- **Bus de Eventos:** Apache Kafka (gestionado, ej. Confluent Cloud
- **Microservicios:** *Spring Boot 3* (Java).
- **Bases de Datos:** Azure DB for PostgreSQL (Relacional) y Azure CosmosDB (NoSQL).
- **Identidad:** *Keycloak* (desplegado en AKS o como servicio).
- **Observabilidad:** *Grafana, Prometheus, Loki* (El "Stack GLP").

4. Seguridad, Cumplimiento Normativo y Ley de Datos

El modelo de seguridad **Zero Trust** se implementa utilizando **Keycloak** como pilar central.

- **Autenticación y Autorización (IAM):**
    - Se usará **Keycloak** como proveedor de identidad (IdP) centralizado, implementando **OIDC (OpenID Connect)** sobre **OAuth 2.0**.
    - Los clientes (Web/Móvil) obtienen un **JWT (JSON Web Token)** de Keycloak.
    - El **Azure API Gateway** valida el JWT en cada solicitud, integrándose con Keycloak (vía OIDC endpoint).
    - La autorización interna (entre microservicios) utilizará **mTLS (Mutual TLS)**.
- **Seguridad de APIs:**
    - **Azure API Gateway:** Aplica políticas de seguridad (Rate Limiting, Prevención de DDoS, Validación de Schemas JSON).
    - **Open Finance:** Se implementará el perfil FAPI (Financial-grade API) de OAuth 2.0, soportado por Keycloak.
- **Cumplimiento y Ley Orgánica de Protección de Datos (LOPD):**
    - **Cifrado:** Todos los datos personales (PII) estarán cifrados **en tránsito** (TLS 1.3) y **en reposo** (**Azure Key Vault** para gestionar claves).
    - **Minimización de Datos:** Las APIs y microservicios solo expondrán los datos estrictamente necesarios.
    - **Pseudonimización:** Los datos sensibles en logs y entornos no productivos serán enmascarados.
    - **Auditoría:** Todas las llamadas a APIs que accedan a datos personales serán registradas para auditoría.

5. Estrategia de Alta Disponibilidad (HA) y Recuperación (DR)

La estrategia de HA/DR se basa en la redundancia y automatización de Azure.

- **Alta Disponibilidad (HA):**
    - **Infraestructura (Azure):** Todos los componentes (Azure APIM, AKS, Event Hubs, BBDD) se desplegarán en una configuración **Activo-Activo** a través de múltiples **Zonas de Disponibilidad (AZs)** dentro de una misma región de Azure.
    - **Microservicios (Spring Boot):** Se desplegarán múltiples réplicas (Pods) de cada microservicio en el clúster de **AKS**. Si una réplica falla, Kubernetes la reinicia automáticamente en otro nodo.
    - **Core Tradicional (On-Prem):** Se asumirá una configuración Activo-Pasivo.
    - **Bases de Datos:** Se usarán servicios de BBDD gestionados (ej. Azure DB for PostgreSQL - Flexible Server con HA).
- **Recuperación ante Desastres (DR):**
    - **Estrategia Multi-Región:** Se define un RTO (Tiempo de Recuperación) y RPO (Punto de Recuperación) bajos.

    - **Componentes Cloud:** Se mantendrá una infraestructura pasiva (Warm Standby) en una **región de Azure secundaria**. **Terraform** e **Azure DevOps** permitirán levantar los servicios rápidamente en la región de DR.
    - **Datos:** Las bases de datos y los tópicos de Kafka se replicarán de forma asíncrona a la región de DR.
    - **Tráfico:** En caso de un desastre regional, el tráfico DNS (**Azure Traffic Manager**) se conmutará a la región de DR.

Profundización: Estrategia de Observabilidad con Grafana

La HA y DR son imposibles sin una observabilidad robusta. Implementaremos el "Stack GLP" (Grafana, Loki, Prometheus) como solución centralizada.

1. **Métricas (Prometheus):** Cada microservicio **Spring Boot** expondrá métricas (tasas de error, latencia, uso de JVM) a través de `Actuator` y `micrometer-registry-prometheus`. Prometheus recolectará estas métricas.
2. **Logs (Loki):** Los logs de todas las aplicaciones (AKS) y componentes de infraestructura serán enviados a Loki. Esto permite correlacionar logs de diferentes servicios sin indexación costosa.
3. **Visualización (Grafana): Grafana** será el "único panel de vidrio". Se crearán dashboards para:
    - **Salud del Sistema (HA):** Visualizará las métricas de Prometheus (HTTP 5xx, latencia, reinicios de pods). Las alertas se configurarán vía Grafana/Alertmanager para notificar al equipo de guardia *antes* de una falla catastrófica.
    - **Flujo de Negocio:** Dashboards que rastreen el flujo de una transacción de principio a fin.
    - **Estado de DR:** Monitoreo del lag de replicación de bases de datos y Kafka hacia la región de DR.
4. **Trazabilidad (OpenTelemetry):** Para depurar flujos complejos (ej. un pago que toca 5 microservicios), se usará **OpenTelemetry**. Los servicios Spring Boot se instrumentarán para propagar un *trace ID*. Estos traces se enviarán a un backend (como Jaeger o Grafana Tempo) y se visualizarán en Grafana, permitiendo ver exactamente en qué microservicio falló una transacción.

6. Estrategia de Integración Multicore

El desafío es integrar un core *legacy* (confiable pero lento) y un core *digital* (ágil pero nuevo). El objetivo no es una "migración big-bang", sino una **coexistencia evolutiva**.

- **Fachada de API Unificada:** El cliente (BFF, Web/Móvil) NUNCA sabe a qué core está llamando. Solo interactúa con las APIs de Proceso en **Azure API Gateway**.

- **Enrutamiento Inteligente:** Los microservicios de **Spring Boot** (ej. `ms_cuentas`) contienen la lógica de enrutamiento.
    - **Ejemplo:** Un cliente "Digital" (creado en la nueva plataforma) tendrá sus datos en el **Core Digital**. Un cliente "Tradicional" (migrado) tendrá sus datos en el **Core Tradicional**. El `ms_clientes` sabe dónde buscar.
- **Patrón Strangler (Tarea 2):** Se utiliza para la migración. El ACL (`acl_legacy`) expone el Core Tradicional. Gradualmente, la lógica se migra a los nuevos microservicios y al Core Digital, "estrangulando" al Core Tradicional.

Profundización: Estrategia de Consistencia de Datos Multicore

El mayor reto del multicore es la consistencia de datos. No podemos usar transacciones distribuidas (XA) por su fragilidad. Usaremos un modelo de **Consistencia Eventual** basado en eventos y el **Patrón Saga**.

- **Fuente de Verdad (Source of Truth):** Se define por dominio.
    - Ej. El Core Tradicional es la fuente de verdad para "Saldos Contables".
    - Ej. El Core Digital es la fuente de verdad para "Productos Digitales" (ej. billetera).
- **Sincronización (Change Data Capture - CDC):**
    - Para datos que *nacen* en el Legacy (ej. un depósito en sucursal), se usará una herramienta de **CDC** (ej. Debezium) sobre la base de datos del legacy (o el ACL publicará un evento).
    - *Flujo:*
        1. Depósito en sucursal -> `UPDATE` en BBDD Legacy.
        2. CDC/ACL captura el cambio y publica un evento en Kafka: `{"evento": "SaldoActualizado", "clienteId": "123", "nuevoSaldo": 500}`.
        3. Un microservicio de sincronización consume este evento y actualiza la *vista de lectura* (CQRS) en la base de datos digital.
        4. La Banca Móvil (que lee de la BBDD digital) refleja el saldo actualizado en segundos.
- **Transacciones Distribuidas (Patrón Saga):**
    - Para operaciones que abarcan ambos cores (ej. una transferencia de una cuenta *legacy* a una *digital*).
    - Se usará el **Patrón Saga de Orquestación**, donde el `ms_pagos` actúa como orquestador.
    - *Flujo Exitoso:*
        1. `ms_pagos` -> Llama al ACL para debitar del Core Tradicional.
        2. `ms_pagos` -> Llama al `ms_cuentas` para acreditar en el Core Digital.
        3. `ms_pagos` -> Marca la saga como "Completada".
    - *Flujo de Falla (Compensación):*
        1. `ms_pagos` -> Llama al ACL para debitar del Core Tradicional (Éxito).

2. `ms_pagos` -> Llama al `ms_cuentas` para acreditar en el Core Digital (Falla).
3. `ms_pagos` detecta la falla e inicia la compensación: Llama al ACL para *re-acreditar* (revertir) el débito en el Core Tradicional.
4. La transacción queda autómicamente revertida para el usuario.

7. Gestión de Identidad y Acceso (IAM)

Como se mencionó, **Keycloak** será el IdP central.

- **Mecanismos Detallados:**
  1. **Autenticación de Clientes (Web/Móvil):**
     - El cliente es redirigido a la página de login de **Keycloak**.
     - El cliente se autentica (usuario/pass, MFA configurado en Keycloak).
     - Keycloak emite un **JWT (Access Token + ID Token)**.
  2. **Autenticación de APIs de Terceros (Open Finance):**
     - Utilizarán el flujo `Client Credentials` o `Authorization Code` de OAuth 2.0 en Keycloak.
  3. **Autorización en Azure API Gateway:**
     - El cliente envía el JWT en el *header* `Authorization: Bearer <token>`.
     - El API Gateway (usando su política `validate-jwt`) valida la firma del token contra las llaves públicas (JWKS endpoint) de **Keycloak**.
     - El API Gateway verifica los **scopes** (permisos) del token. Si la API requiere el scope `pagos:escribir` y el token no lo tiene, la solicitud es rechazada (403 Forbidden).
  4. **Acceso entre Microservicios (mTLS):**
     - Dentro del *service mesh* de AKS, los servicios se autentican entre sí usando certificados (mTLS).
  5. **Control de Acceso a Sistemas Legacy:**
     - El ACL (`acl_legacy`) es responsable de traducir el JWT (o la identidad del usuario) a lo que el Core Tradicional entienda (ej. Kerberos, usuario/pass de servicio).

8. Estrategia de API Internas y Externas (Mensajería)

Se define un catálogo claro de APIs basado en el estándar OpenAPI 3.0.

- **APIs Externas (Experience APIs):**
    - **Diseño:** Orientadas al consumidor (BFF, Open Finance).
    - **Protocolo: REST/JSON** sobre HTTPS. Síncronas.
    - **Estándar:** Documentadas con OpenAPI 3.0. Alojadas en el portal de desarrolladores de **Azure API Gateway**.
    - **Seguridad:** OAuth 2.0 (gestionado por Keycloak).
- **APIs Internas (Process y System APIs):**
    - **Diseño:** Orientadas al dominio BIAN.
    - **Protocolo Síncrono: REST/JSON** (implementado con **Spring Web**).
    - **Seguridad:** mTLS.
- **Mensajería (Eventos Asíncronos):**
    - **Protocolo: Apache Kafka** (para *streaming*).
    - **Estándar:** El formato del mensaje será estandarizado usando **CloudEvents** (para la envoltura) y **Apache Avro** (para el *payload*), gobernado por un *Schema Registry*.
    - **Uso:** Notificación de Eventos (ej. `PagoProcesado`) y Sincronización Multicore.

9. Modelo de Gobierno de APIs y Microservicios

Proponemos un modelo de **"Gobierno Federado"** (o "Centro de Habilitación" - C4E), que balancea la agilidad del equipo con la estabilidad de la plataforma.

Equipo Central de Plataforma (C4E)

El C4E no construye las APIs, sino que **habilita** a los equipos de dominio para que las construyan de forma eficiente y segura.

- **Responsabilidades:** Gestiona la infraestructura (AKS, Kafka, Azure APIM, Keycloak), define estándares, y provee "pavimentos dorados" (ej. pipelines de CI/CD, plantillas de Spring Boot).

Equipos de Dominio (Squads)

Equipos multifuncionales (Dev, QA, Ops, Negocio) alineados a dominios BIAN (ej. "Squad de Pagos").

- **Responsabilidades:** Diseñan, construyen (en **Spring Boot**), prueban, despliegan y **operan** (You Build It, You Run It) sus propios microservicios y APIs.

Pilares del Gobierno

1. **Gobierno de Diseño (Design-Time):**

    - **API-First:** El contrato **OpenAPI 3.0** es la primera entrega. Se diseña y se maqueta en Azure APIM antes de escribir código.
    - **Estándares:** El C4E define y publica estándares claros:
        - Nomenclatura (ej. `kebab-case` para URLs).
        - Versionamiento (ej. `/api/v1/...`).
        - Manejo de errores (ej. RFC 7807 Problem Details).
    - **Linters:** Se integra un linter de OpenAPI (ej. Spectral) en el pipeline de CI para *forzar* los estándares automáticamente.
2. **Gobierno de Ejecución (Run-Time):**
    - **Políticas Centralizadas:** El C4E crea y gestiona políticas reutilizables en **Azure API Gateway** (ej. "Política de Seguridad JWT-Keycloak", "Política de Rate-Limit Nivel Bronce").
    - **Consumo Forzado:** Los equipos de dominio *aplican* estas políticas a sus APIs, no las reinventan.
    - **Ciclo de Vida de API:** El C4E gestiona la promoción de APIs (de `desarrollo` a `producción`) y su eventual deprecación y retiro.
3. **Gobierno de Observabilidad (El Stack Grafana):**
    - **Plantillas (Templates):** El C4E provee plantillas de **Spring Boot** pre-configuradas con **OpenTelemetry**, `Actuator` y `Loki-appender`.
    - **Dashboards "Golden":** El C4E crea dashboards base en **Grafana** (Salud de JVM, Métricas de Kafka, Errores HTTP) que todos los servicios heredan.
    - **Alertas Base:** El C4E define alertas críticas (ej. "Servicio Caído", "Latencia > 2s") que aplican a todos los microservicios. Los Squads pueden añadir alertas específicas de negocio.

10. Estrategia de Migración Gradual (Minimizar Riesgo)

Se rechaza un "Big Bang". La migración se basará en el **Patrón Strangler Fig**, habilitado por el ACL y el API Gateway.

- **Fase 1: Exponer (Duración: 3-6 meses)**
    - **Objetivo:** Crear una fachada de APIs sobre el *legacy*.
    - **Acciones:**
        1. Implementar el **Azure API Gateway**.
        2. Construir el ACL (`acl_legacy`) en **Spring Boot** para "envolver" las funciones críticas del Core Tradicional.
        3. Implementar **Keycloak** para la gestión de identidad.
- **Fase 2: Construir (Duración: 6-12 meses)**
    - **Objetivo:** Lanzar la nueva Banca Web/Móvil.
    - **Acciones:**

        1. Desarrollar el BFF y las primeras Process APIs (ej. `ms_clientes`) en **Spring Boot**.
        2. Implementar el Core Digital (v1) para *nuevos productos* (ej. una nueva "cuenta digital" que no existe en el *legacy*).
- **Fase 3: Estrangular y Migrar (Duración: 12-24+ meses)**
    - **Objetivo:** Mover la lógica de negocio del *legacy* al digital.
    - **Acciones (Por Dominio - Ej. "Pagos"):**
        1. El API Gateway enruta `POST /api/pagos` al nuevo `ms_pagos` (Spring Boot).
        2. El `ms_pagos` orquesta la lógica, llamando al Core Digital o al ACL (Core Tradicional) según corresponda (ver Tarea 5 - Patrón Saga).
        3. Se inicia la migración de datos en segundo plano, cliente por cliente.

Consideraciones Finales (Cumplimiento de Recomendaciones):

- **Profundidad Técnica:** La solución se basa en una pila tecnológica específica y moderna (Azure, Spring Boot, Keycloak, Grafana) y patrones justificados (EDA, API-Led, Strangler, Saga, CDC).
- **Enfoque Práctico y Justificado:** Cada decisión tecnológica y de arquitectura está justificada para cumplir con los requisitos de HA, baja latencia, seguridad y migración de bajo riesgo.