Decisión de Arquitectura y Plan de Migración  
Monolito a Microservicios

**Leonardo Escobar Osorio - Arquitecto de Software**  
**29 de Agosto de 2025**  
**Versión:** 1.0

# Contexto y objetivos

La empresa cuenta con un sistema monolítico que soporta catálogo de productos, cuentas de usuario y procesamiento de órdenes. El crecimiento del negocio y la necesidad de desplegar cambios de manera independiente requieren una arquitectura más modular, resiliente y escalable.

El objetivo es migrar gradualmente a microservicios minimizando riesgo y manteniendo continuidad operativa.

En la solución propuesta se plantea desacoplar cada uno de los procesos principales como servicios independientes, contenerizados y orquestados

**Objetivos específicos:**

• Desacoplar (Catálogo, Usuarios, Órdenes) para desplegar y escalar de forma independiente.

• Mejorar disponibilidad y resiliencia frente a fallos parciales.

• Acelerar el despliegue de requerimientos con pipelines CI/CD por servicio.

• Asegurar observabilidad, seguridad y gobernanza desde el primer dia

# Alcance inicial y supuestos

* Modularizar extraer y publicar servicios de Catálogo, Usuarios y Órdenes.
* Base de datos por servicio (propiedad de datos por dominio).
* Tráfico externo detrás de un API Gateway con autenticación OAuth2/OIDC
* Despliegue en contenedores (Kubernetes) y mensajes asíncronos con un broker

# Consideraciones

### Identificación de Servicios

Para empezar la migración yo separaría tres microservicios principales:

* **Catálogo de Productos**: manejar todo lo relacionado con productos, precios y disponibilidad. Es el que expone la información que el cliente ve al navegar.
* **Usuarios**: encargarse de las cuentas, roles y permisos, además de conectarse con el sistema de autenticación.
* **Órdenes**: llevar el control de las compras, desde que se crea la orden hasta que pasa por el pago y la confirmación.

**Patrones de Comunicación**

Cuando un cliente crea una orden usaría un **flujo mixto**:

* **Síncrono (REST)** para que el usuario reciba de inmediato un número de orden y tenga respuesta rápida.
* **Asíncrono (mensajes/eventos)** entre los servicios internos (órdenes, catálogo e inventario, pagos) para procesar stock y pagos en segundo plano sin bloquear al usuario.

### Estrategia de Datos

Aplicaría el patrón de **base de datos por servicio**, donde cada microservicio maneja su propia base según sus necesidades.  
Así se evita el acoplamiento y cada equipo puede trabajar de forma independiente. Para mantener consistencia entre servicios usaría eventos y patrones como sagas .

### Componentes Transversales

La autenticación y autorización las manejaría con dos piezas clave:

* Un **API Gateway**, que centraliza el acceso, valida tokens y aplica reglas.
* Un **Servidor de Autorización** (por ejemplo Auth0), encargado de emitir y validar los JWT con OAuth2/OIDC.

# Arquitectura propuesta (alto nivel)

Componentes:

* API Gateway / Ingreso, ruteo, número de peticiones, autenticacion y autorización, canal seguro
* Servicio Catálogo (REST): productos, precios, stock leído.
* Servicio Usuarios (REST): cuentas, perfiles, autenticación/roles (integrado con IdP).
* Servicio Órdenes (REST + eventos): toma de órdenes sincrónica; publicación de eventos para procesos posteriores.
* Creación de Endpoints
* Broker de Mensajes: eventos de dominio (orden creada, orden pagada, inventario reservado, etc.).
* Observabilidad: logging centralizado, métricas y trazas distribuidas.
* Persistencia: base de datos por servicio; patrones
* ETL para consultas, analíticas para BI, dashboards

# Modelo de servicios y límites (DDD)

• Bounded Context Catálogo: Producto, Inventario, Precio. Responsables de mantener la verdad del stock e información comercial

• Bounded Context Usuarios: Usuario, Rol, Permiso, Sesión. Control de acceso, gestión de credenciales y roles

• Bounded Context Órdenes: Orden, detalle orden, Pago, Estado. Orquesta/coreografía del flujo de compra.

# Comunicación entre servicios

• Síncrona (HTTP/REST) para consultas tipo lectura inmediata: Usuarios, Catálogo.

• Asíncrona (eventos) para procesos de negocio con bajo acoplamiento y tolerancia a fallos Ej: Crear una orden emite un evento de orden creada y el servicio de inventario la procesa después

• Patrones de resiliencia: timeouts, reintentos, estado de servicio, aislar recursos.

• control de eventos y consolidación de estados que eviten duplicidad

# Datos y consistencia

• Estrategia: Base de datos por servicio. Cada servicio expone su modelo vía API

• Consistencia: eventual mediante Sagas ejemplo reservar inventario y cobrar pago, si el pago falla se deshace la reserva

• Vistas materializadas en endpoints

• Migración de datos: Manejar doble escritura lectura entre el sistema actual y el nuevo hasta poder apagar el sistema legado, crear querys de consistencias de datos

# Seguridad

• Autenticación y autorización con OAuth2/OIDC

• API Gateway aplica validación de tokens, límites de tasa, WAF.

• Secrets en gestor seguro (KMS/Secrets Manager).

• Cifrado en tránsito (https) y en reposo; políticas de mínimo privilegio.

• Auditoría de accesos, trazas y cambios de configuración.

# Despliegue, CI/CD e Infraestructura

• Contenedores por servicio (Docker) orquestados por Kubernetes.

• IaC (Terraform) + Helm para releases.

• Pipelines CI/CD por servicio: ambiente de pruebas y ambiente de producción , automatización de pruebas

# Observabilidad

* Logs centralizados, manejo de auditoria creación, modificación, transacciones.
* Definicion de métricas, trazas o KPIs para identificar anomalías
* Creacion de Dashboards por dominio y tableros ejecutivos.

# Ventajas

• Microservicios vs monolito modular: +autonomía y escalado; −complejidad operativa.

• Base por servicio: +independencia; −compleja la consistencia → mitigada con Sagas y eventos.

• Asíncrono en órdenes: +resiliencia; −inconsistencias y mayor observabilidad necesaria.

• API Gateway: +seguridad y gobernanza y alta disponibilidad).

# Pasos para de migración

**Fase 0 – Preparación:** Telemetría, feature flags, gateway y autenticación listos.

**Fase 1 – Catálogo:** Extraer API de productos/stock; crear DB propia; sincronizar con monolito (ETL/doble escritura).

**Fase 2 – Usuarios:** Externalizar identidad (IdP) y perfiles; actualizar clientes a nuevo endpoint.

**Fase 3 – Órdenes:** Implementar flujo de órdenes con Saga; publicar eventos; desactivar rutas del monolito gradualmente.

**Fase 4 – Retiro del monolito:** Apagar módulos migrados; decommission de infraestructura heredada.

# Riesgos y mitigaciones

• Inestabilidad por latencia en red → aplicar timeouts/circuit breakers y pruebas de resiliencia (chaos).

• Inconsistencias entre dominios → idempotencia, reintentos y reconciliación periódica.

• Falta de gobierno de esquemas de eventos → versionado, contratos (AsyncAPI), revisión de diseño.

• Déficit de observabilidad → presupuestar plataformas y KPIs desde el inicio.

# Métricas de éxito

• Reducción >30% del lead time de cambios por dominio.

• Disponibilidad ≥ 99.9% por servicio.

• Tiempo de recuperación de fallas < 30 min con procedimientos documentados y alertas.

• Escalado independiente medible (CPU/memoria por servicio).

Preguntas de contexto:

1. **Atributos de Calidad: ¿Qué son los "Atributos de Calidad" (o Requisitos No Funcionales)? Da un ejemplo de dos atributos que puedan entrar en conflicto y explica por qué.**

Son requisitos no funcionales que determinan cómo se comporta el sistema más allá de sus funcionalidades core. Ejemplos: seguridad, disponibilidad, rendimiento, usabilidad, mantenibilidad.

• Conflicto típico: Seguridad vs Rendimiento.

Aumentar la seguridad (cifrado, validaciones) puede generar latencia y bajar el rendimiento. El arquitecto debe balancear prioridades según el negocio.

1. **Teorema CAP: Explica brevemente el Teorema CAP (Consistencia, Disponibilidad, Tolerancia a Particiones). ¿Cómo influye este teorema en la elección de una base de datos para un sistema distribuido?**

Indica que en un sistema distribuido solo se pueden garantizar 2 de los 3 atributos a la vez:

• Consistencia (C): todos los nodos ven la misma información al mismo tiempo.

• Disponibilidad (A): siempre hay respuesta, aunque no sea la más actualizada.

• Tolerancia a particiones (P): el sistema sigue funcionando aunque haya fallos en la red.

Influencia en elección de DB:

• Si el negocio requiere datos siempre actualizados → elegir CP (ej. MongoDB, HBase).

• Si el negocio prioriza disponibilidad frente a red inestable → elegir AP (ej. Cassandra, DynamoDB).

1. **PatronesdeResiliencia:¿Quéeselpatrón"CircuitBreaker"(Interruptor)? ¿Qué tipo de fallo en cascada ayuda a prevenir en una arquitectura de microservicios?**

Es un componente de resiliencia que abre el circuito cuando detecta repetidos fallos en un servicio remoto.

• Evita que un microservicio saturado siga recibiendo peticiones.

• Previene fallos en cascada, protegiendo al resto del sistema y permitiendo recuperación gradual.

1. **Coreografía vs. Orquestación: Compara y contrasta los enfoques de coreografía y orquestación para la comunicación entre servicios. ¿Cuáles son los trade-offs de cada uno?**

Coreografía: cada servicio reacciona a eventos y actúa en consecuencia.

* Menos acoplamiento, mayor independencia.
* Complejidad aumenta, difícil de seguir el flujo.

Orquestación: un servicio central (orquestador) controla el flujo de la transacción.

* Claridad, trazabilidad, un punto de control.
* Riesgo de cuello de botella y acoplamiento al orquestador.

Trade-off: simplicidad de seguimiento (orquestación) vs independencia y escalabilidad (coreografía).

1. **Patrón Strangler Fig: ¿Qué es el "Patrón del Higo Estrangulador" (Strangler Fig Pattern) y en qué escenario de modernización de sistemas es una estrategia recomendada?**

Estrategia de modernización que migra un monolito poco a poco: se extraen funcionalidades a nuevos servicios, mientras el monolito sigue funcionando hasta que es “estrangulado” por completo.

Escenario recomendado: modernizar aplicaciones legadas sin detener el negocio.

1. **Gestión de Datos en Microservicios: Explica el principio de "Base de Datos por Servicio". ¿Qué desafíos introduce este patrón y cómo se pueden mitigar? (Ej: consistencia de datos).**

Cada microservicio es dueño de su propia base de datos.

* Independencia de despliegue y escalado.
* Desafíos: duplicación de datos, consistencia entre servicios.

Mitigación:

* Consistencia eventual vía eventos.
* Patrones Saga para coordinar transacciones distribuidas.
* CQRS y materialized views para consultas complejas.

1. **Observabilidad: ¿Cuáles son los "Tres Pilares de la Observabilidad"? ¿Por qué es insuficiente tener solo logs en un sistema distribuido complejo?**

Los Tres Pilares:

1. Logs → eventos de texto.

2. Métricas → valores numéricos (CPU, latencia, throughput).

3. Trazas distribuidas → seguimiento de una petición a través de múltiples servicios.

Por qué no basta solo con logs: en sistemas distribuidos se necesita correlación y métricas de desempeño; sin métricas ni trazas no puedes identificar latencia ni cuellos de botella.

1. **Infraestructura como Código (IaC): ¿Qué es IaC y por qué es una práctica fundamental para un arquitecto de software en la era de la nube y DevOps?**

Definir y gestionar infraestructura mediante código declarativo (Terraform, Ansible, CloudFormation).

• Reproducible, auditable, versionable.

• Escalabilidad y consistencia en entornos.

• Es esencial en cloud y DevOps porque permite CI/CD también para la infraestructura.

1. **Service Discovery: ¿Qué es el "Descubrimiento de Servicios" (Service Discovery)? Describe la diferencia entre el patrón del lado del cliente y el del lado del servidor.**

Mecanismo para que los microservicios encuentren dinámicamente a otros sin necesidad de configuraciones fijas.

• Client-side discovery: el cliente consulta el registro (ej. Netflix Eureka) y decide a qué instancia conectarse.

• Server-side discovery: el cliente llama a un balanceador (ej. Kubernetes Service) y este enruta la petición.

1. **Bounded Context (Contexto Delimitado): ¿Qué es un Bounded Context en el contexto de Domain-Driven Design (DDD) y cómo se relaciona con la definición de los límites de un microservicio?**

Bounded Context (DDD)

Un Bounded Context define un límite claro de un dominio de negocio y su modelo, evitando ambigüedades.

• Ejemplo: “Órdenes” y “Usuarios” son contextos distintos.

• Relación con microservicios: cada Bounded Context suele corresponder a un microservicio autónomo, dueño de su propio lenguaje y datos.