

Arquitectura de Microservicios

Una Explicación Detallada

Conceptos, Implementación y Mejores Prácticas

Mayo 2025



Contenido

Arquitectura de Microservicios: Una Explicación Detallada

- 01 🏠 Introducción
- 02 ☰ Contenido
- 03 ❓ ¿Qué son los Microservicios?
- 04 ⌚ Orígenes y Evolución
- 05 📋 Características Principales
- 06 ↔ Monolito vs. Microservicios
- 07 🔗 Componentes Clave
- 08 🎯 Principios de Diseño
- 09 👍 Ventajas
- 10 ⚠️ Desventajas y Desafíos
- 11 🧩 Patrones de Diseño
- 12 💬 Comunicación entre Servicios
- 13 🗄️ Bases de Datos
- 14 📦 Contenedores
- 15 🌐 Orquestación con Kubernetes
- 16 🚀 Implementación CI/CD
- 17 🏢 Casos de Uso Reales
- 18 🏆 Mejores Prácticas
- 19 ⚠️ Errores Comunes
- 20 🌱 El Futuro de los Microservicios

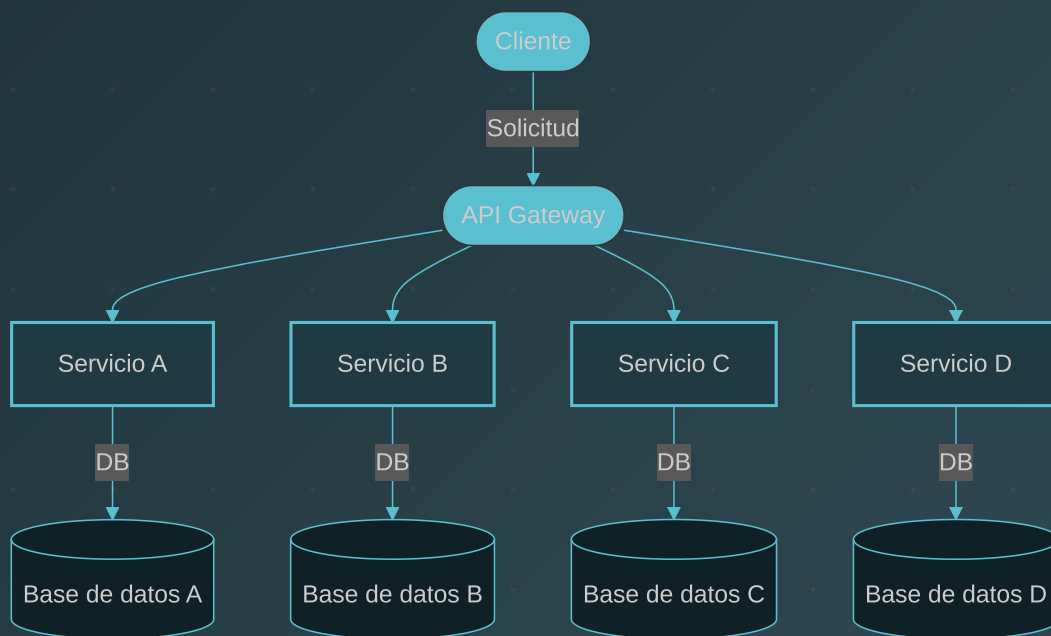
Progreso

2/20 diapositivas

¿Qué son los Microservicios?

Definición

Los microservicios son un enfoque arquitectónico en el que una aplicación se construye como un conjunto de **pequeños servicios independientes** que se comunican a través de APIs bien definidas.



Conceptos Fundamentales



Servicios Pequeños e Independientes

Cada servicio implementa una capacidad de negocio específica y acotada.



Despliegue Independiente

Cada microservicio puede ser desplegado, actualizado y escalado de forma independiente.



Base de Datos por Servicio

Cada microservicio gestiona su propia base de datos para garantizar el desacoplamiento.



Comunicación vía API

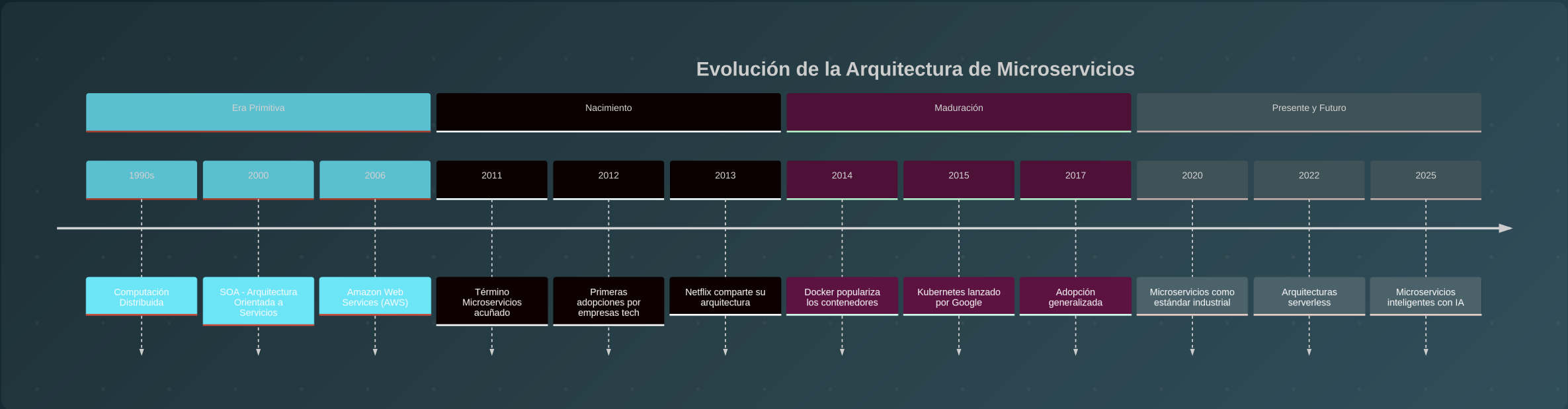
Los microservicios se comunican entre sí a través de APIs bien definidas, típicamente REST o mensajería.



Equipos Autónomos

Equipos pequeños desarrollan, prueban y despliegan sus servicios de forma independiente.

Orígenes y Evolución de los Microservicios



De Monolitos a SOA

Evolución desde aplicaciones monolíticas a servicios más especializados pero aún pesados.

Influencia de la Nube

La computación en la nube permitió el despliegue flexible necesario para los microservicios.

Revolución de Contenedores

Docker (2013) transformó la implementación de microservicios con contenedores ligeros y portables.

Era de Orquestación

Kubernetes permitió gestionar eficientemente ecosistemas complejos de microservicios.

Pioneros de los Microservicios



Características Principales de los Microservicios



Componentes Independientes

Servicios que pueden ser desplegados y actualizados de forma autónoma sin afectar al resto.



Organización por Capacidades

Estructurados alrededor de funcionalidades de negocio específicas y no por capas tecnológicas.



Descentralización de Datos

Cada servicio gestiona su propia base de datos, permitiendo usar diferentes tecnologías según necesidades.



Automatización

Infraestructura como código y CI/CD para pruebas y despliegue automatizados y confiables.



Resiliencia

Diseñados para tolerar fallos mediante patrones como Circuit Breaker y mecanismos de aislamiento.



Escalabilidad

Cada servicio puede escalarse independientemente según la demanda, optimizando recursos.



Diseño Evolutivo

Arquitectura que puede evolucionar y adaptarse gradualmente a los cambios del negocio.



Interfaces Definidas

APIs bien definidas que permiten la comunicación entre servicios sin exponer detalles internos.

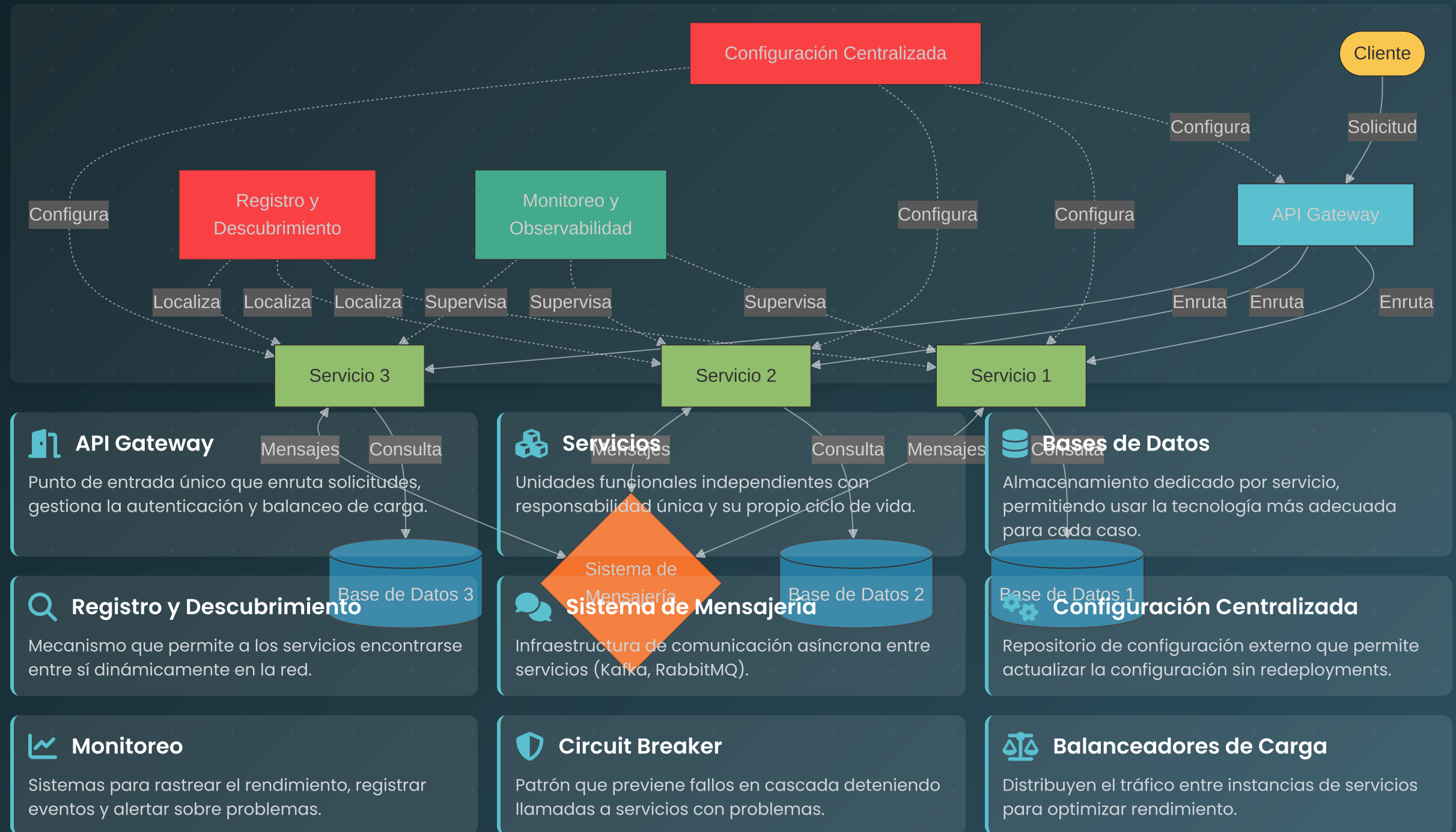
Los microservicios fomentan: **Autonomía** • **Flexibilidad** • **Escalabilidad** • **Resiliencia**

Monolito vs. Microservicios



La elección depende del tamaño del proyecto, complejidad y requerimientos de escalabilidad

Componentes Clave de los Microservicios



Principios de Diseño de Microservicios



Responsabilidad Única

Cada servicio debe enfocarse en una **capacidad de negocio** específica y tener una única responsabilidad.



Autonomía

Servicios **independientes** que pueden ser desplegados y actualizados sin afectar a otros servicios.



Domain-Driven Design

Servicios delimitados por **contextos de dominio** bien definidos dentro del negocio.



Aislamiento de Fallos

Los errores en un servicio no deben **propagar fallos** a otros servicios o al sistema completo.



Despliegue Independiente

Cada servicio debe poder **desplegarse de forma aislada** sin requerir coordinación con otros servicios.



Descentralización

Evitar servicios centralizados para prevenir **puntos únicos de fallo** y cuellos de botella.



API Bien Definidas

Interfaces **claras y estables** que ocultan los detalles internos de implementación de cada servicio.



Observabilidad

Capacidad de **monitorizar y diagnosticar** el comportamiento y el estado de cada servicio en el sistema.



Escalabilidad Horizontal

Diseño para **escalar** servicios individualmente añadiendo más instancias según la demanda.



Evolución Independiente

Servicios que evolucionan a **su propio ritmo** sin depender de ciclos de desarrollo de otros componentes.



Resiliencia

Diseño para **recuperarse automáticamente** de fallos y continuar operando en condiciones adversas.



Orquestación Mínima

Preferir **coreografía** sobre orquestación centralizada para la comunicación entre servicios.

Estos principios guían la creación de sistemas de microservicios **flexibles, escalables y mantenibles**

Ventajas de los Microservicios

Ventajas Tecnológicas



Escalabilidad Selectiva

Escalado independiente de servicios según demanda, **optimizando recursos** y reduciendo costos.



Flexibilidad Tecnológica

Libertad para usar diferentes tecnologías y lenguajes según las **necesidades específicas** de cada servicio.



Testabilidad Mejorada

Unidades más pequeñas facilitan **pruebas automatizadas** y aislamiento de componentes para testing.

Ventajas Organizacionales



Equipos Autónomos

Equipos pequeños con **responsabilidad completa** sobre servicios específicos, mejorando productividad y propiedad.



Ciclos de Desarrollo Rápidos

Desarrollo y **despliegue acelerado** de funcionalidades sin necesidad de coordinar con todo el sistema.



Paralelización del Trabajo

Múltiples equipos trabajando **simultáneamente** en diferentes servicios sin bloqueos ni dependencias.

Ventajas Empresariales



Resistencia a Fallos

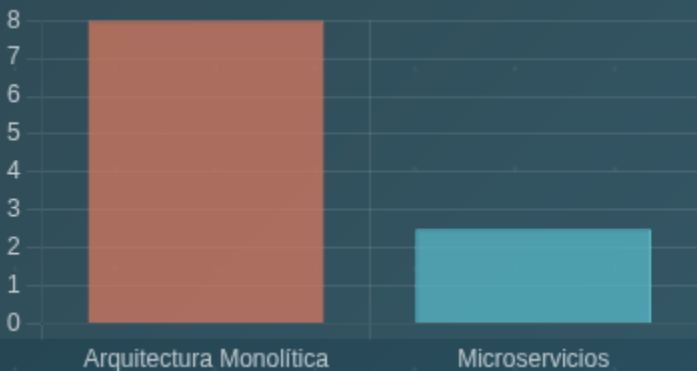
Fallos aislados que no afectan al sistema completo, mejorando la **disponibilidad global**.



Adaptabilidad al Cambio

Mayor **agilidad** para adaptarse a cambios del mercado y evolucionar partes específicas del sistema.

Tiempos de Entrega Mejorados



Los microservicios permiten crear sistemas más **adaptables** • **resilientes** • **escalables** • **mantenibles**

Desventajas y Desafíos de los Microservicios



Complejidad Distribuida

Gestionar un sistema distribuido añade **dificultad significativa** en la comunicación, monitoreo y depuración.



Consistencia de Datos

Mantener la **integridad de datos** entre múltiples bases de datos requiere patrones complejos como SAGA.



Sobrecarga Operativa

Cada servicio requiere **infraestructura propia** y mecanismos de despliegue, aumentando costos y complejidad.



Latencia en Red

La comunicación entre servicios introduce **retrasos de red** que pueden degradar la experiencia del usuario.



Duplicación de Código

Posible **repetición de lógica** común entre servicios cuando no se establece una estrategia de compartición adecuada.



Seguridad Distribuida

Mayor **superficie de ataque** y complejidad para implementar autenticación y autorización entre servicios.



Curva de Aprendizaje

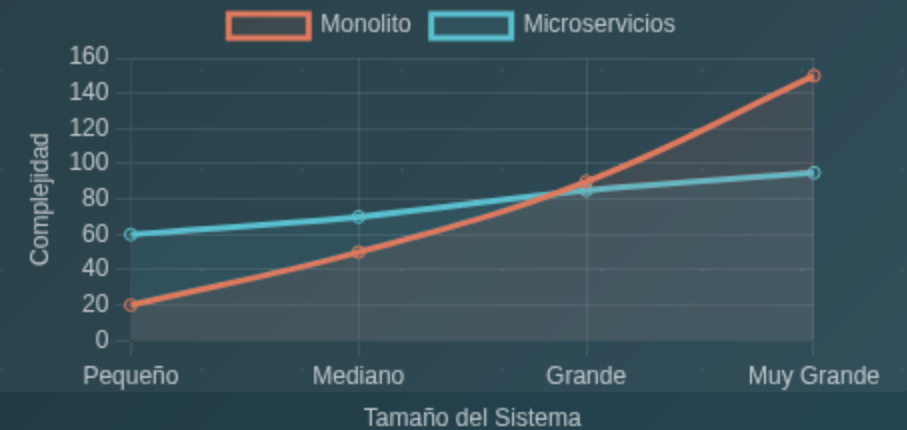
Requiere **nuevas habilidades** y conocimientos en el equipo para gestionar sistemas distribuidos.



Costos Iniciales

Inversión inicial significativa en infraestructura, herramientas y capacitación del equipo.

Complejidad vs. Tamaño del Sistema



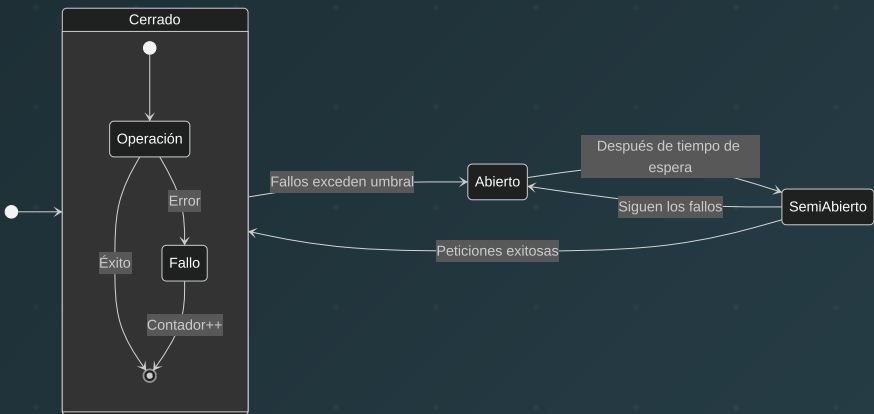
Nivel de Preparación Requerido



Los microservicios no son una **solución universal** - Su adopción debe ser **gradual** y **justificada** por necesidades específicas.

Patrones de Diseño en Microservicios

Patrón Circuit Breaker



Previene fallos en cascada al detectar y aislar servicios con problemas

Patrones de Comunicación

API Gateway
Punto de entrada centralizado que enruta peticiones y actúa como fachada para microservicios.

BFF (Backend for Frontend)
API Gateways específicos para diferentes tipos de clientes (móvil, web, IoT).

Patrones de Resiliencia

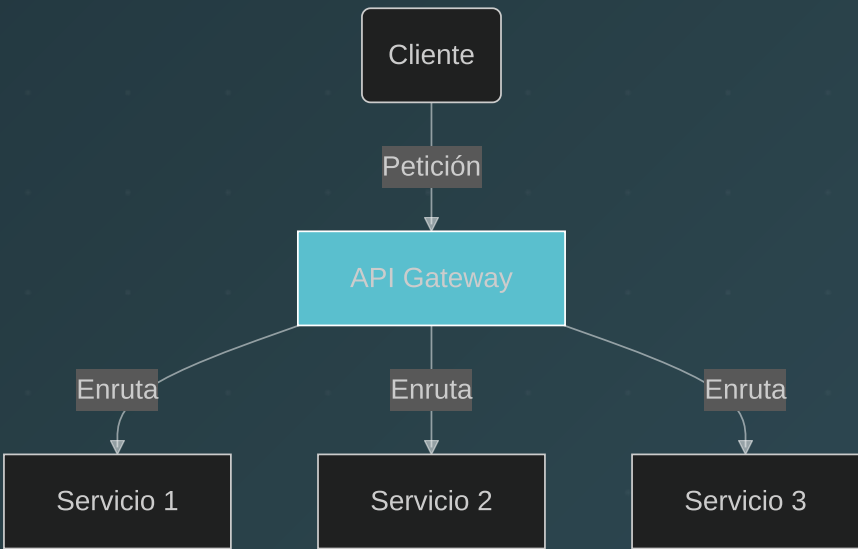
Circuit Breaker
Previene llamadas a servicios con fallos, implementando estados (cerrado, abierto, semi-abierto).

Health Check
Monitoreo constante del estado de cada servicio mediante endpoints específicos.

Retry Pattern
Reintento automático de operaciones fallidas con backoff exponencial.

Timeout Pattern
Establece límites de tiempo para evitar esperas infinitas en comunicaciones.

Patrón API Gateway



Punto de entrada único que gestiona peticiones y oculta la complejidad interna

Patrones de Datos

Database per Service
Cada microservicio tiene su propia base de datos, asegurando independencia y desacoplamiento.

Event Sourcing
Almacena cambios de estado como secuencia de eventos, en lugar del estado actual.

CQRS
Command Query Responsibility Segregation: separa operaciones de lectura y escritura.

Saga Pattern
Gestiona transacciones distribuidas con compensaciones para mantener consistencia.

Patrones de Despliegue

Sidecar Pattern
Componente auxiliar conectado al servicio principal para extender funcionalidades.

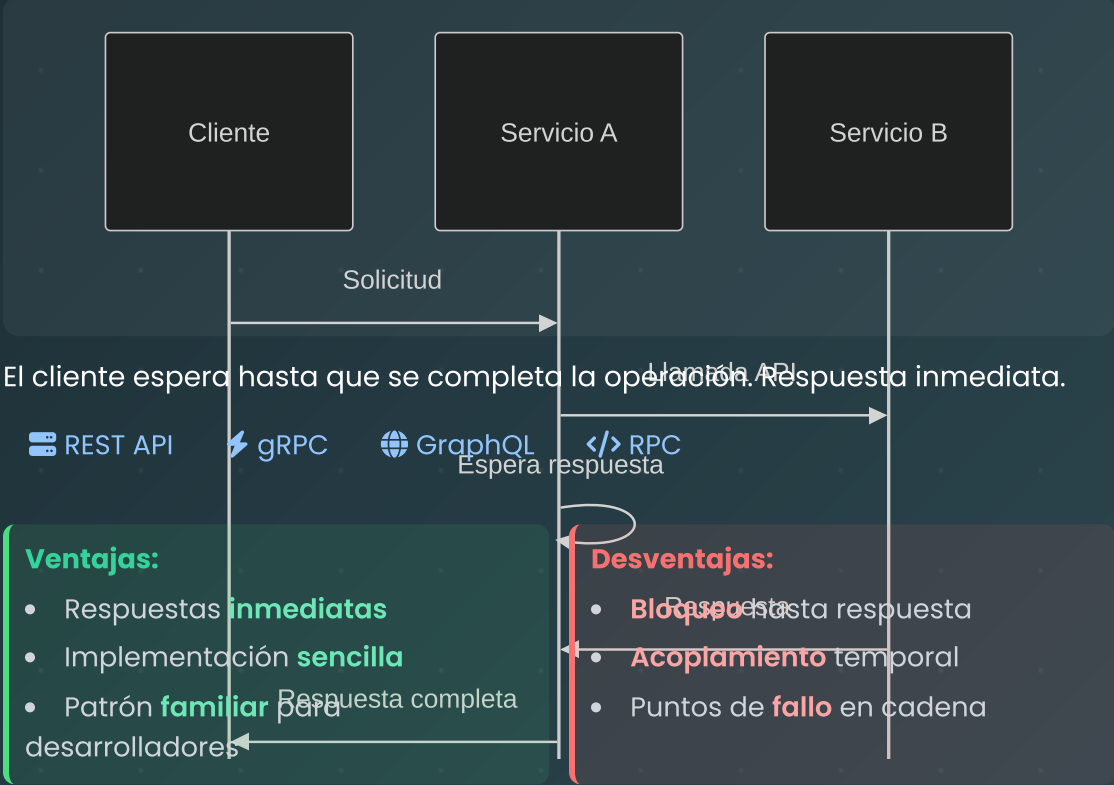
Service Mesh
Infraestructura que gestiona comunicación entre servicios con proxies dedicados.

Strangler Pattern
Migración incremental desde un monolito hacia microservicios.

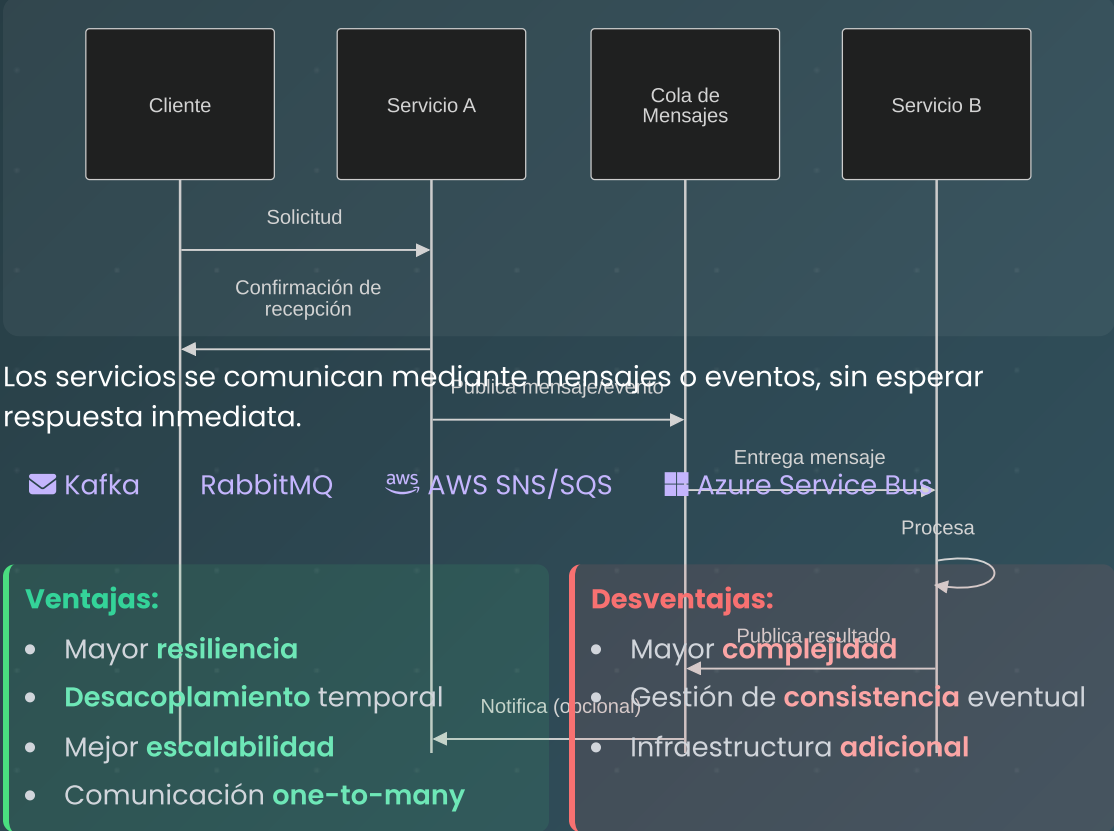
Blue-Green Deployment
Dos entornos idénticos para actualizaciones sin tiempo de inactividad.

Comunicación entre Microservicios

Comunicación Sincrónica



Comunicación Asincrónica



Estilos de Comunicación

Request/Response Llamadas directas esperando respuesta (REST, RPC)	Publicar/Suscribir Emisión de eventos a múltiples suscriptores
Stream Processing Procesamiento de flujos continuos de datos	Cola de Mensajes Envío de mensajes a consumidores específicos

Formatos de Datos

JSON	Protocol Buffers	Avro	XML
MessagePack	BSON		

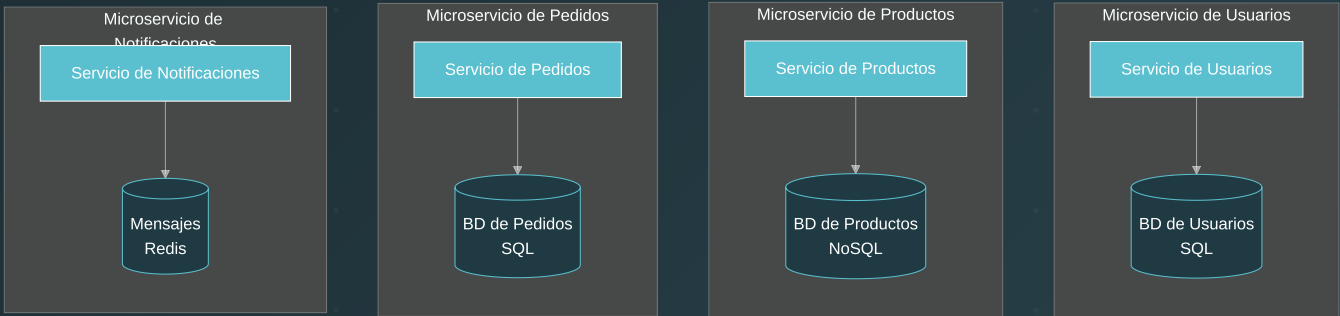
El formato de serialización afecta el rendimiento, tamaño y compatibilidad de la comunicación.

Estrategias Híbridas

Los sistemas más robustos utilizan combinaciones de comunicación:
Sincrónica para consultas REST/GraphQL para operaciones de lectura que requieren respuesta inmediata
Asincrónica para cambios de estado Eventos/mensajes para actualizaciones y operaciones complejas

Bases de Datos en Microservicios

Principio: Base de Datos por Servicio



Database per Service

Cada microservicio gestiona su propia base de datos, con acceso exclusivo a sus datos.

- ✓ Alto desacoplamiento
- ⚠ Transacciones distribuidas

Shared Database

Múltiples servicios comparten la misma base de datos. Evitar en lo posible.

- ✓ Transacciones ACID
- ⚠ Alto acoplamiento

CQRS

Command Query Responsibility Segregation: separar operaciones de lectura y escritura.

- ✓ Optimización de consultas
- ⚠ Mayor complejidad

Saga Pattern

Secuencia de transacciones locales con acciones compensatorias para fallos.

- ✓ Consistencia eventual
- ⚠ Lógica compleja

Transacciones Distribuidas con Saga Pattern



Syntax error in text
mermaid version 11.6.0

Tipos de Bases de Datos

Relacionales (SQL)

Ideal para datos estructurados y transaccionales PostgreSQL MySQL

Documentos (NoSQL)

Para datos semi-estructurados y esquemas flexibles MongoDB CouchDB

Key-Value

Alta velocidad, cachés y datos sencillos Redis DynamoDB

Columnar

Análisis y grandes volúmenes de datos Cassandra HBase

Gráficos

Relaciones complejas y datos interconectados Neo4j JanusGraph

Selección de Base de Datos



Mejores Prácticas

- Usar **base de datos por servicio** como enfoque predeterminado
- Elegir el tipo de BD según **necesidades específicas** del servicio
- Implementar **transacciones compensatorias** para operaciones distribuidas
- Aceptar la **consistencia eventual** cuando sea posible



Contenedores en Microservicios



¿Qué son los contenedores?

Entornos ligeros y portables que empaquetan una aplicación con todas sus dependencias en una unidad estandarizada de software, permitiendo que se ejecute de manera consistente en cualquier entorno.



Docker



Containerd



Podman



LXC

Beneficios para Microservicios



Aislamiento

Cada microservicio opera independientemente sin interferir con otros.



Portabilidad

Funcionamiento consistente en cualquier entorno: local, pruebas o producción.



Eficiencia de Recursos

Menor sobrecarga que las VMs, mayor densidad de aplicaciones por servidor.



Escalabilidad

Fácil de escalar horizontalmente creando más instancias de contenedores.

Máquinas Virtuales vs. Contenedores

Máquina Virtual



- Cada VM tiene su propio SO completo
- Mayor aislamiento
- Más pesado (GB)

Contenedores



- Comparten el SO del host
 - Ligeros (MB)
 - Inicio rápido

Ejemplo de Dockerfile

```
# Imagen base
FROM node:14-alpine

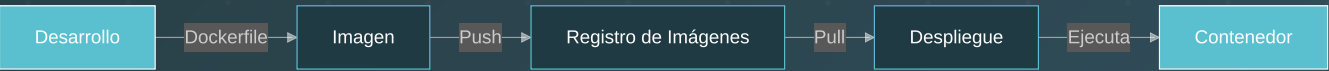
# Directorio de trabajo
WORKDIR /usr/src/app

# Copiar archivos de dependencias
COPY package*.json ./

# Instalar dependencias
```

Este Dockerfile crea una imagen para un microservicio Node.js

Flujo de Trabajo con Contenedores



Los contenedores proporcionan un flujo de trabajo consistente desde desarrollo hasta producción

> Comandos Básicos

`docker build`
Construir imagen

`docker run`
Ejecutar contenedor

`docker push/pull`
Publicar/descargar imágenes

`docker-compose up`
Iniciar múltiples contenedores

Los contenedores son el **bloque fundamental** para la **orquestración** con Kubernetes y otras plataformas

Orquestación con Kubernetes

¿Qué es Kubernetes?

Plataforma de **orquestación de contenedores** de código abierto que automatiza el despliegue, escalado y gestión de aplicaciones en contenedores, ideal para ecosistemas de microservicios.

- Cloud Native
- Alta Disponibilidad
- Declarativo
- Auto-reparación

Arquitectura de Kubernetes



Pods
Unidad básica, contiene uno o más contenedores que comparten recursos.

Services
Proporciona dirección estable y balanceo de carga para los pods.

ConfigMaps y Secrets
Gestiona configuraciones y datos sensibles separados del código.

Deployments
Gestiona despliegues y actualizaciones declarativas de aplicaciones.

Ingress
Gestiona el acceso externo a servicios, típicamente HTTP.

Namespaces
Dividen los recursos en grupos aislados (equipos/ambientes).

Beneficios para Microservicios

Despliegue Automatizado
Permite actualizar servicios sin tiempo de inactividad (rolling updates).

Escalado Horizontal
Escala servicios individuales según demanda, manual o automáticamente.

Auto-recuperación
Reemplaza automáticamente contenedores fallidos o nodos no saludables.

Service Discovery
Localización automática de servicios mediante DNS integrado.

Balanceo de Carga
Distribuye tráfico entre múltiples instancias de un servicio.

Ejemplo de Despliegue

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: payment-service
  labels:
    app: payment-service
spec:
  replicas: 3
  selector:
    matchLabels:
      app: payment-service
```

Kubernetes en la Nube

- Amazon EKS
- Azure AKS
- Google GKE

Kubernetes proporciona la **plataforma ideal** para operar **microservicios a escala** con alta disponibilidad y resiliencia

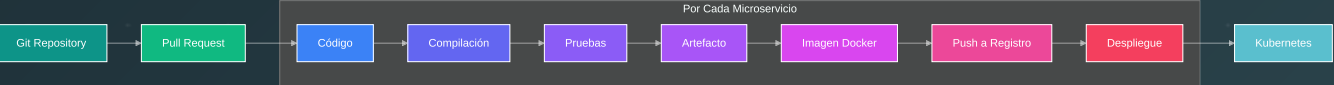
Implementación CI/CD para Microservicios

CI/CD en Microservicios

Integración Continua & Despliegue Continuo

Automatización del ciclo completo de desarrollo que permite entregar **cambios frecuentes y confiables** a cada microservicio de forma independiente, sin afectar al resto del sistema.

Pipeline de CI/CD para Microservicios



Cada microservicio tiene su propio pipeline independiente, permitiendo ciclos de desarrollo paralelos

Fases Clave

- Control de Versiones**
Repositorios independientes o monorepo con gestión eficiente.
- Pruebas Automatizadas**
Unitarias, integración, contratos y pruebas de componentes.
- Containerización**
Empaquetado en imágenes Docker con versiones inmutables.
- Despliegue Progresivo**
Canary, blue-green o rolling updates para minimizar impacto.
- Observabilidad**
Monitoreo en tiempo real para detectar problemas rápidamente.

Herramientas Populares


GitHub Actions


Jenkins


GitLab CI


CircleCI


Docker


ArgoCD

Mejores Prácticas

- Automatizar **todo el proceso** desde commit hasta despliegue
- Implementar **feature flags** para activar/desactivar funcionalidades
- Utilizar **pruebas de contrato** entre servicios
- Adoptar **GitOps** para gestionar configuración de infraestructura
- Considerar **versionado semántico** para APIs
- Implementar **estrategias de rollback** automáticas

Monolito: Un Pipeline

- Ciclos largos de integración
- Coordinación entre equipos
- Despliegue completo de la aplicación
- Pruebas extensas end-to-end
- Mayor riesgo por cambios

Microservicios: Múltiples Pipelines

- Ciclos cortos e independientes
- Equipos autónomos
- Despliegue granular y frecuente
- Pruebas específicas por servicio
- Riesgo aislado a un solo servicio

CI/CD es **esencial** para entregar microservicios con **frecuencia, confiabilidad y seguridad**

Casos de Uso Reales de Microservicios

Netflix

Pionero en microservicios. Migró desde monolito a más de **800 servicios** con enfoque en streaming y recomendaciones.

- ✓ 99.99% de disponibilidad
- ✓ +50 deploys diarios

Amazon

Adoptó SOA y microservicios para escalar su mercado online. Miles de servicios autónomos con **equipos de "two-pizza"**.

- ✓ Despliegues cada 11,6 seg
- ✓ Escalado masivo en picos

Uber

Plataforma basada en microservicios para gestionar **millones de viajes** diarios. Usa sistema de mensajería avanzado para coordinación.

- ✓ 2300+ microservicios
- ✓ Procesamiento en tiempo real

PayPal

Migró de monolito Java a microservicios con Node.js para transacciones financieras. Mejoró **rendimiento y seguridad**.

- ✓ -50% tiempo de desarrollo
- ✓ +200M transacciones/día

Spotify

Famoso por su modelo de **"Squads"**. Cada equipo gestiona sus microservicios con alta autonomía para streaming musical y recomendaciones.

- ✓ Autonomía de equipos
- ✓ Innovación acelerada

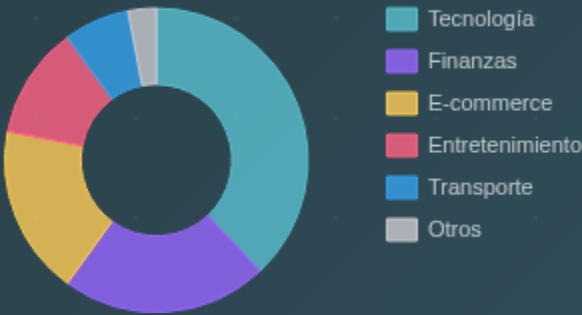
Airbnb

Transformó su plataforma a microservicios para soportar **escala global**. Usa GraphQL para unificar APIs de servicios.

- ✓ Implementación gradual
- ✓ Escalabilidad estacional

Adopción por Sector

Distribución por Industria (%)



Patrones de Éxito

- Equipos pequeños y autónomos**
Enfoque en capacidades de negocio
- Automatización intensiva**
Pipeline CI/CD para cada servicio
- Migración gradual**
Adopción por fases, no todo de golpe
- Monitoreo extensivo**
Observabilidad y detección temprana
- Diseño para fallos**
Resiliencia como característica clave

"No construimos servicios para encapsular código o crear abstracciones. Construimos servicios para encapsular equipos."

— Netflix

Mejores Prácticas en Arquitectura de Microservicios

Diseño de Servicios



CRÍTICO

Un Propósito, Un Servicio

Enfoca cada servicio en una **única responsabilidad** y contexto de negocio bien definido.



Modelado por Dominio

Usa **Domain-Driven Design** para identificar límites de contexto y servicios.



CRÍTICO

Observabilidad Integral

Implementa **logging, métricas y trazas** distribuidas para diagnóstico rápido de problemas.



Diseño para Fallos

Usa **Circuit Breaker, Retries** y otros patrones de resiliencia para manejar fallos.



APIs Estables

Define interfaces **claras y duraderas**. Evolucionas con versiones en lugar de cambios disruptivos.



Tamaño Adecuado

Ni demasiado grandes ni demasiado pequeños. Busca el **equilibrio** para evitar complejidad innecesaria.



Documentación Efectiva

Mantén **documentación actualizada** de APIs, dependencias y procesos operativos.



Health Checks

Implementa endpoints de **salud y disponibilidad** para permitir auto-recuperación de servicios.

Datos y Estado



CRÍTICO

Base de Datos por Servicio

Cada servicio debe tener **su propia base de datos** para mantener independencia y evitar acoplamientos.



Consistencia Eventual

Acepta la **consistencia eventual** entre servicios y diseña para manejarla correctamente.



CRÍTICO

CI/CD Automatizado

Implementa **pipelines completos** para cada servicio con despliegue automatizado y testeo.



Equipos Autónomos

Forma equipos **multifuncionales** con propiedad completa de sus servicios (DevOps).



Evitar Datos Compartidos

Evita **dependencias ocultas** o acceso directo a bases de datos de otros servicios.



Patrones SAGA

Implementa **transacciones distribuidas** con compensaciones para mantener consistencia.



Compartir Conocimiento

Crea **comunidades de práctica** y mecanismos para compartir aprendizajes entre equipos.



Migración Incremental

Adopta **estrategias graduales** como el patrón Strangler para migrar sistemas existentes.

✓ Hacer:

✓ Automatizar despliegues e infraestructura

✓ Empezar con monolito modular y evolucionar


✗ Evitar:

✗ Crear microservicios demasiado pequeños ("nanoservicios")

✗ Compartir bases de datos entre servicios

Errores Comunes en Microservicios


Errores Arquitectónicos



Diseño incorrecto de fronteras **CRÍTICO**

Crear microservicios basados en capas técnicas (UI, lógica, datos) en lugar de capacidades de negocio.


💡 Solución: Diseñar en torno a dominios y contextos acotados (DDD).



Microservicios demasiado pequeños

Crear "nanoservicios" con excesiva granularidad, aumentando la complejidad operativa y de comunicación.


💡 Solución: Enfocarse en la cohesión funcional, no en el tamaño.



Ignorar la complejidad distribuida

Subestimar los desafíos de latencia, consistencia de datos y fallas parciales en sistemas distribuidos.

💡 Solución: Diseñar considerando las 8 falacias de la computación distribuida.




Acoplamiento excesivo

Crear interdependencias entre servicios que deberían funcionar de manera autónoma.

💡 Solución: Aplicar comunicación asíncrona y eventos cuando sea posible.


Errores de Datos e Implementación



Compartir bases de datos **CRÍTICO**

Múltiples servicios compartiendo la misma base de datos, creando un acoplamiento fuerte a nivel de datos.


💡 Solución: Base de datos independiente por servicio o al menos esquemas separados.



Despliegue monolítico

Implementar despliegues acoplados o coordinados que invalidan la independencia de servicios.


💡 Solución: Pipelines CI/CD independientes y despliegues automatizados por servicio.



Bibliotecas compartidas excesivas

Sobrecarga de bibliotecas comunes que generan dependencias entre servicios y fuerzan actualizaciones sincronizadas.

💡 Solución: Limitar dependencias compartidas a lo esencial y versionar adecuadamente.




Ignorar el factor organizacional

Adoptar microservicios sin reorganizar los equipos, manteniendo fronteras organizacionales que no coinciden con los servicios.

💡 Solución: Aplicar la Ley de Conway; alinear equipos con servicios.




Errores Operacionales



Monitoreo insuficiente


Falta de observabilidad adecuada que dificulta diagnosticar problemas en sistemas distribuidos complejos.




Ignorar la resiliencia

No implementar patrones de resiliencia como Circuit Breaker, retries o fallbacks para manejar fallos.

Señales de Alerta


Llamadas síncronas en cascada


Transacciones distribuidas complejas


Despliegues que toman más de 15 minutos

El Futuro de los Microservicios

Tendencias emergentes y evolución de la arquitectura



Serverless Microservices

Servicios sin servidor con **ejecución bajo demanda** y escalado automático. Menor gestión de infraestructura.

📈 Adopción Alta



Service Mesh Avanzado

Evolución de mallas de servicio con **seguridad y observabilidad** integradas a nivel de infraestructura.

📈 Adopción Alta



IA en Microservicios

Incorporación de **capacidades de IA** para autogestión, auto-recuperación y optimización de servicios.

📈 Adopción Media

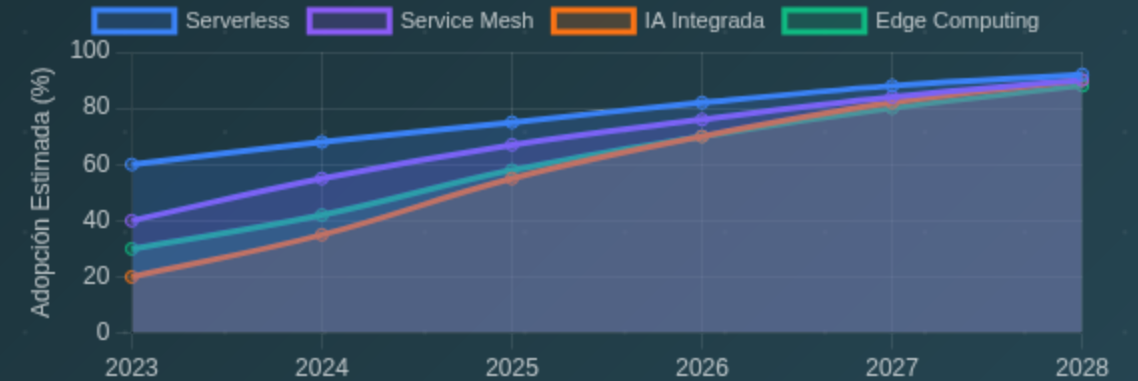


Edge Microservices

Despliegue de servicios en el **borde de la red**, cerca de usuarios y dispositivos IoT para menor latencia.

📈 Adopción Media

📊 Evolución Esperada



"Los microservicios evolucionarán hacia ecosistemas más inteligentes, autogestionados y distribuidos"

👥 Tendencias Organizacionales

Plataforma como Producto

Equipos de plataforma que tratan su infraestructura como un producto que "venden" a equipos de desarrollo.

Competencias Híbridas

Equipos con conocimientos multidisciplinarios en desarrollo, operaciones, seguridad y negocios.

GitOps y Desarrollo Declarativo

Evolución hacia infraestructura y configuraciones gestionadas completamente como código.

Conclusión

Los microservicios continuarán siendo un paradigma dominante, evolucionando hacia arquitecturas más **inteligentes, automatizadas y descentralizadas**, aumentando la eficiencia y reduciendo la complejidad operativa.