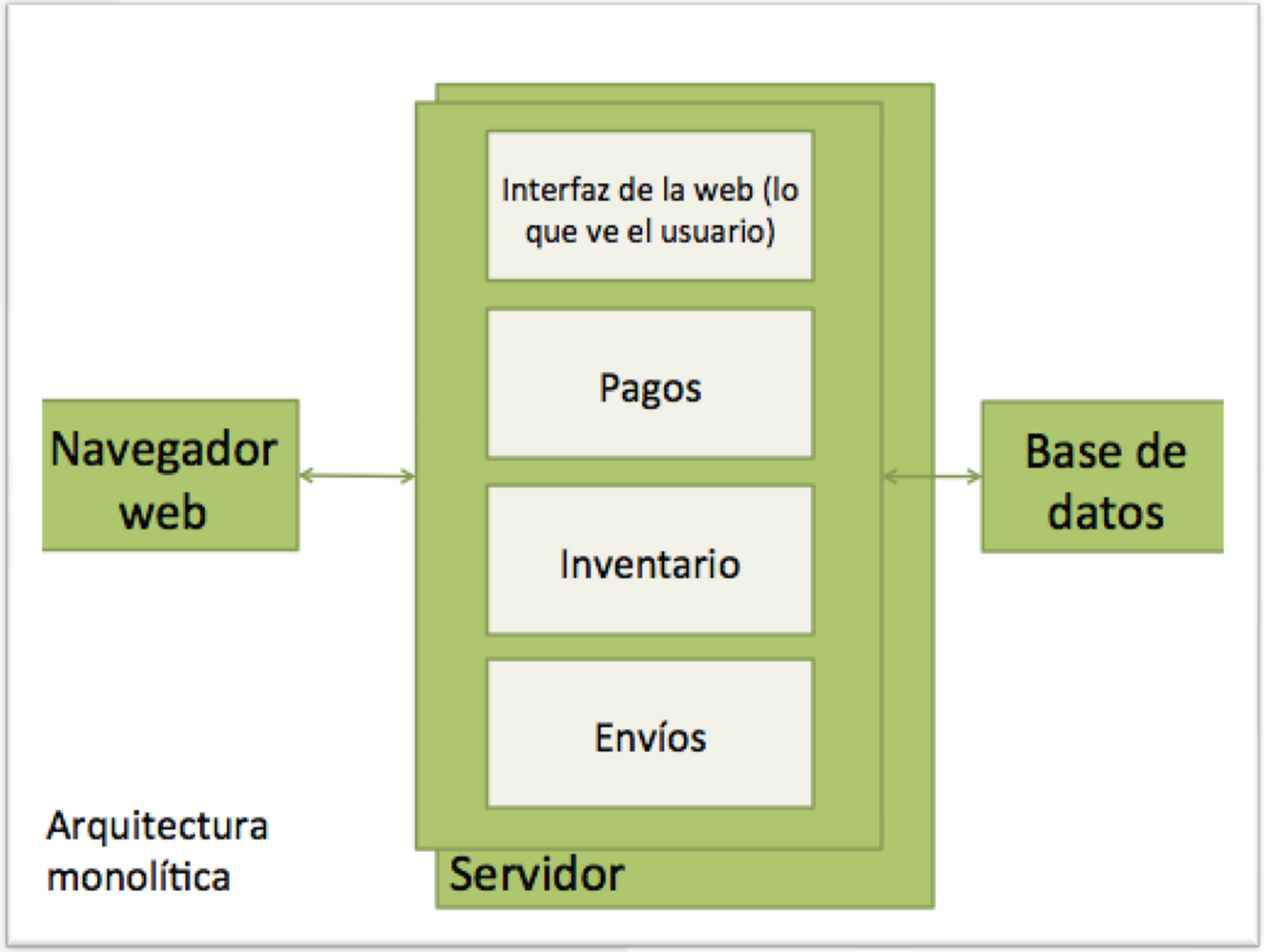
Tema 1 Introducción a los Micro Servicios

Por qué Microservicios

Respuesta al problema de mantenimiento y evolución de **sistemas monolíticos**

Qué es un monolito

Aplicación de software en la que todas sus **capas** (interfaz de usuario, lógica de negocio y acceso a datos) están **combinadas** en un mismo programa y sobre una misma plataforma.

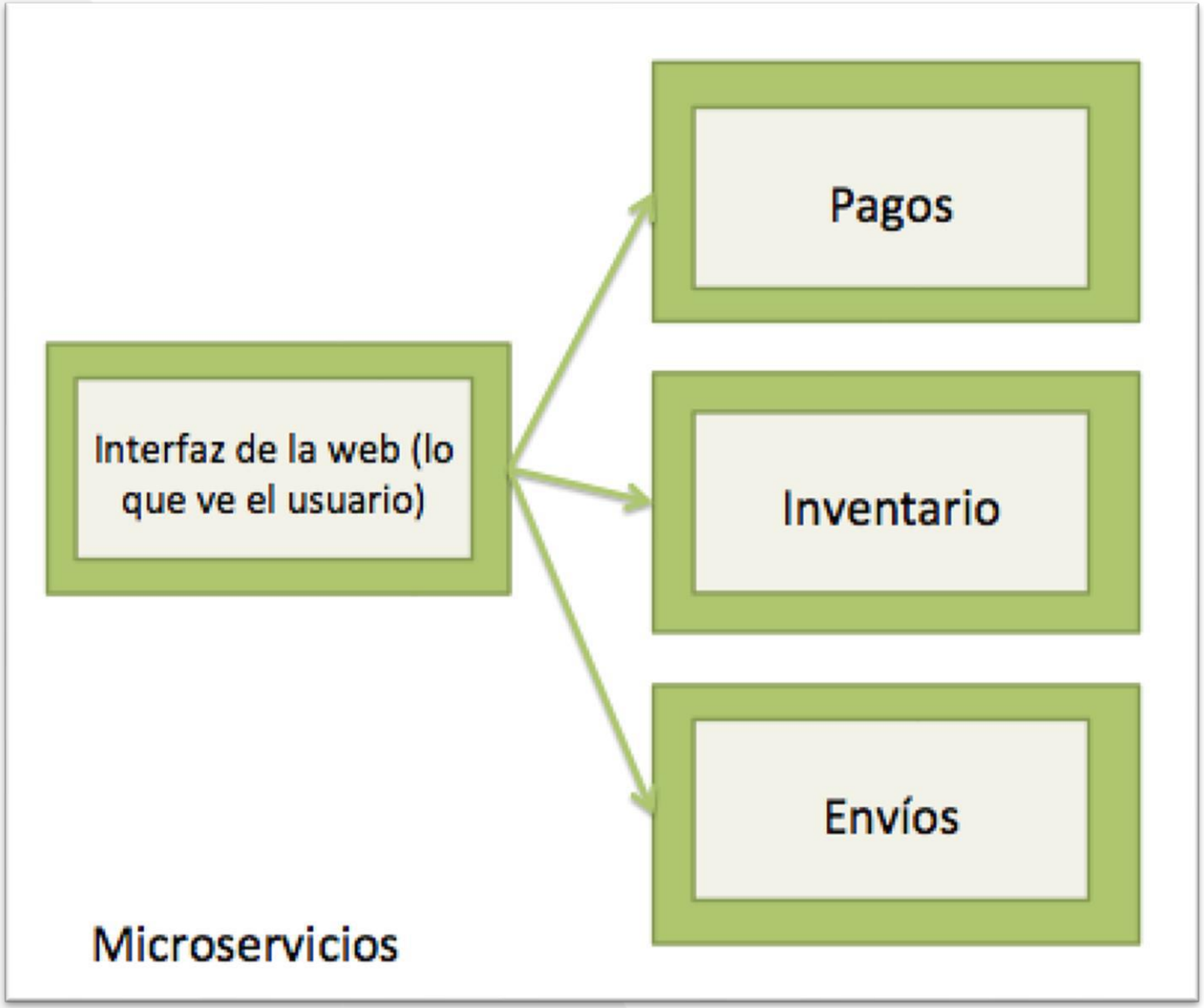


Problemas de los monolitos

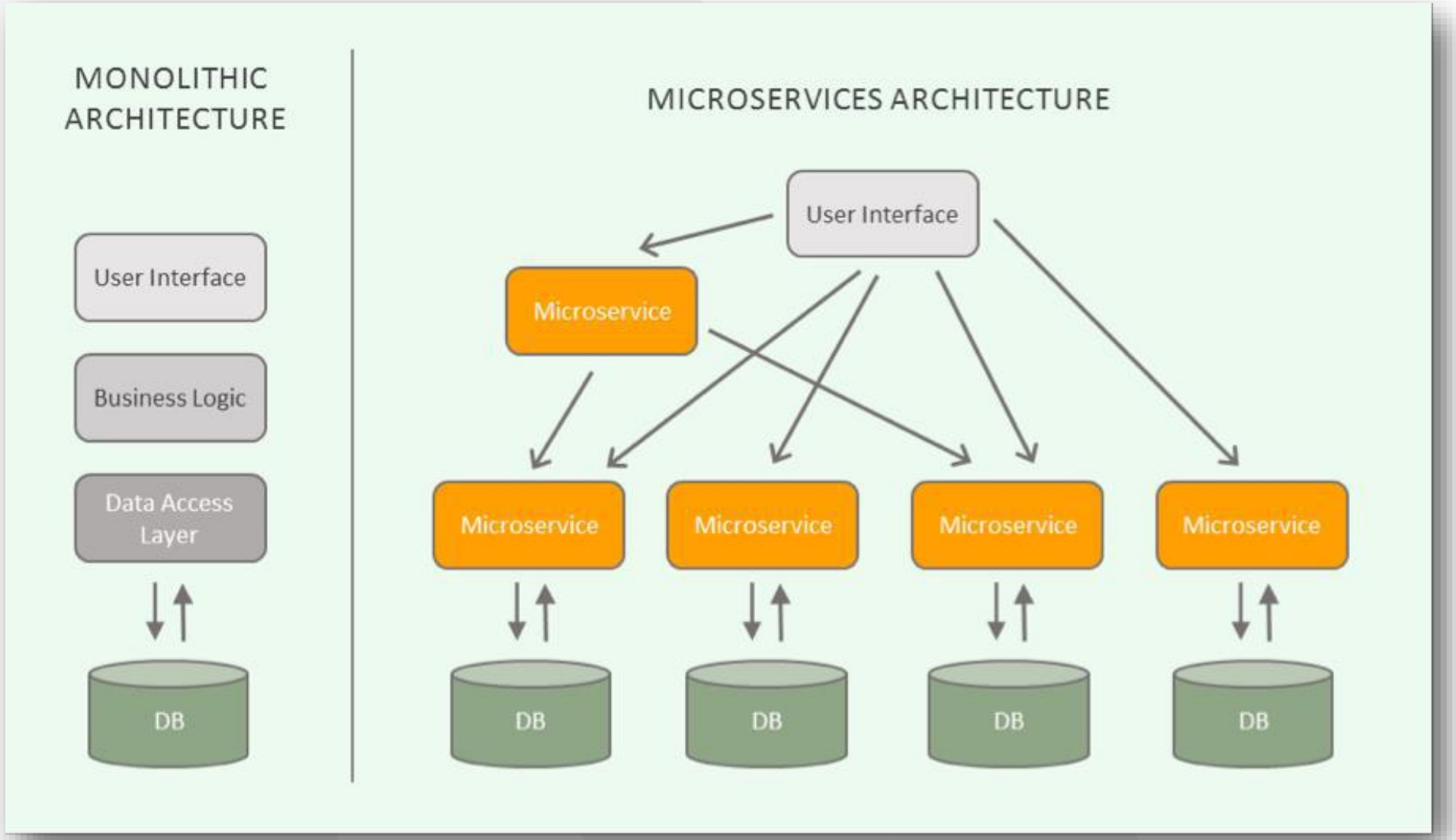
- Si se produce un fallo del sistema **se caen todos** los servicios
- Es difícil de escalar
- Imposibilidad de innovación tecnológica
- Despliegues o actualizaciones conflictivas
- **Complejo de gestionar** equipos de desarrollo
  - Alto número de desarrolladores
  - Nivel de conocimiento de todo el Sistema
  - Dificil desarrollar funcionalidades en paralelo

Qué son los microservicios

Pequeñas aplicaciones con una funcionalidad muy **concreta** y un alto nivel de **especialización** que trabajan en **conjunto**



Monolito vs Microservicio



Características de los microservicios

Son pequeños e independientes.

Cada servicio puede administrarse por un equipo de desarrollo pequeño.

Los servicios pueden actualizarse sin tener que volver a implementar toda la aplicación.

Los servicios son los responsables de conservar sus propios datos o estado externo.

Los servicios se comunican mediante API bien definidas.

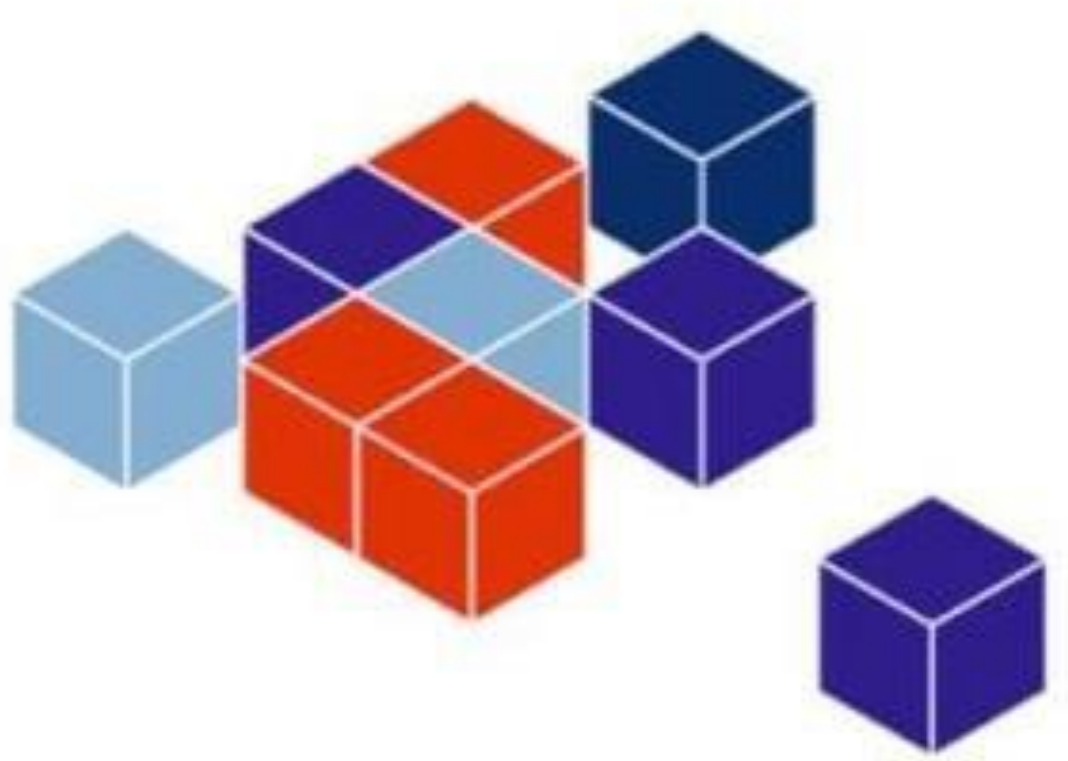
No es necesario que los servicios compartan la misma pila de tecnología, las bibliotecas o los marcos de trabajo.

Ventajas de los microservicios

- Alta **tolerancia a fallos**. Si una parte del sistema se cae lo demás sigue funcionando (Circuit breaker)
- Permite escalabilidad del sistema
- Implementación **sencilla**
- Código más **mantenible** (menos interdependencias)
- Agilidad de cambios
- Gestión de equipos más simple
- Despliegues y actualizaciones con riesgos controlados
- Permite uso de distintas tecnologías

Requisitos microservicios

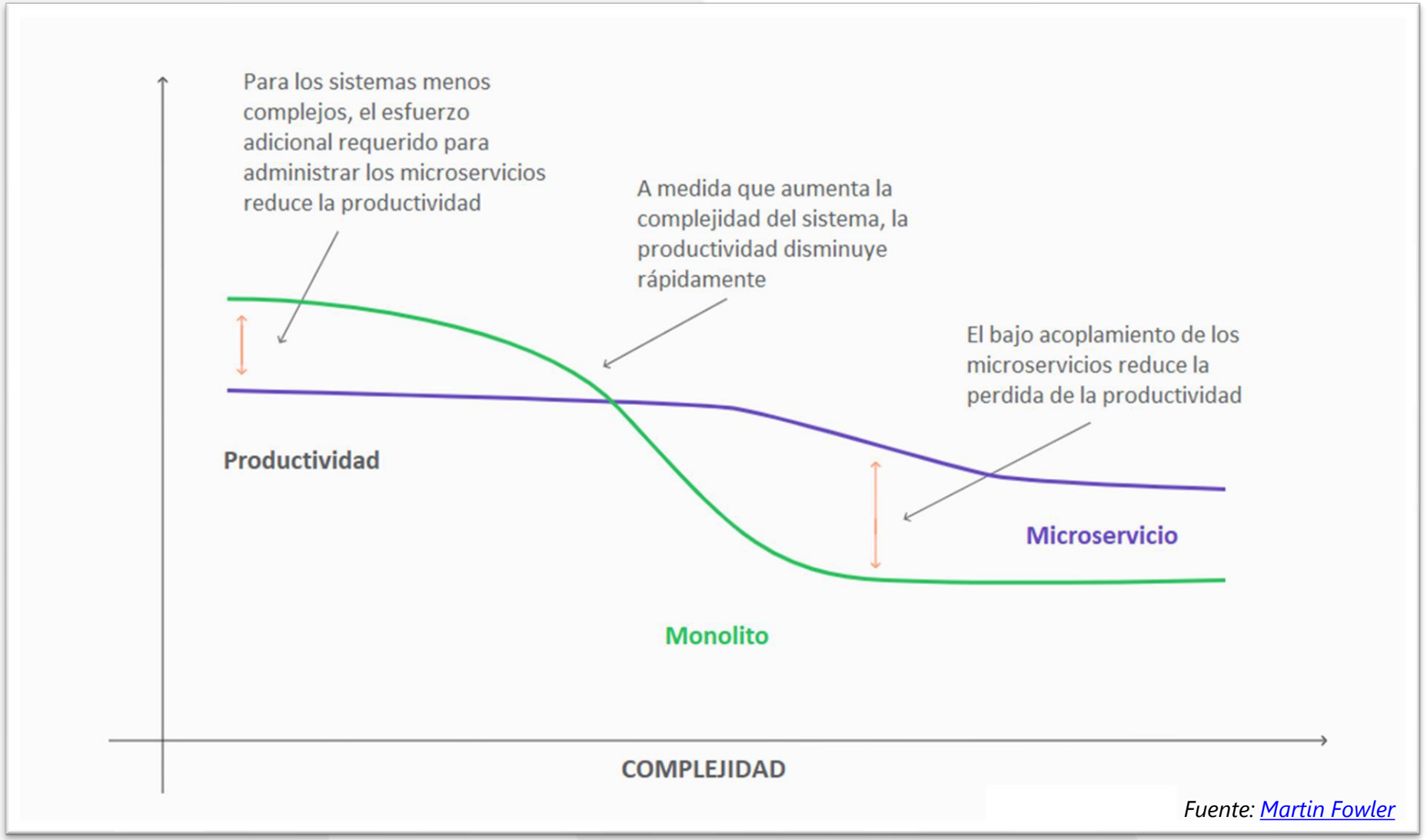
- Deben tener un solo dominio
- Alta especialización de sus funciones
- Exponer una API
- Tener una base de datos propia



Problemas microservicios

- **Orquestación compleja** de los sistemas
- Aumento de la complejidad del entorno de desarrollo
- Los **contratos** de los microservicios (API) no son fáciles de cambiar
- Los equipos de desarrollo necesitan formación y/o experiencia
- Las uniones de datos son complejas
- Sistema responsable de la seguridad

Comparativa de la arquitectura de microservicios y la monolítica



Fuente: Martin Fowler

Comunicaciones entre microservicios

Se puede utilizar el tipo de comunicaciones que se quiera, pero lo recomendable es que sea algún **estándar**. De esta forma ya se tienen librerías para el desarrollo rápido y eficiente

Tipos de comunicaciones:

- REST
- gRPC
- SOAP
- Sockets
- Otras

Patrones de software utilizados en microservicios

Patrones para la descomposición	Patrones de infraestructura	Patrones de integración	Patrones de acceso a datos	Patrones de observabilidad	Patrones para servicios externos
Por capacidades de negocio	Configuración centralizada	**API Gateway**	Tabla de índices	**Agregación de logs**	Un backend por frontend
Por subdominios (enfoque DDD)	**Descubrimiento y registro**	Publicador/ suscriptor	Sharding	Monitoreo de punto final	Capa anticorrupción
Estrangulador	**Balanceo de carga**	Coreografía	**Una base de datos por servicio**	**Métricas de rendimiento**	
Bulkhead	Reintentos	Orquestación	CQRS	Traceo distribuido	
	Circuit breaker	Saga			
	Azul/verde				

Circuit Breaker (1/2)

Objetivo: identificar cuando hay un fallo de un sistema y contener el problema.

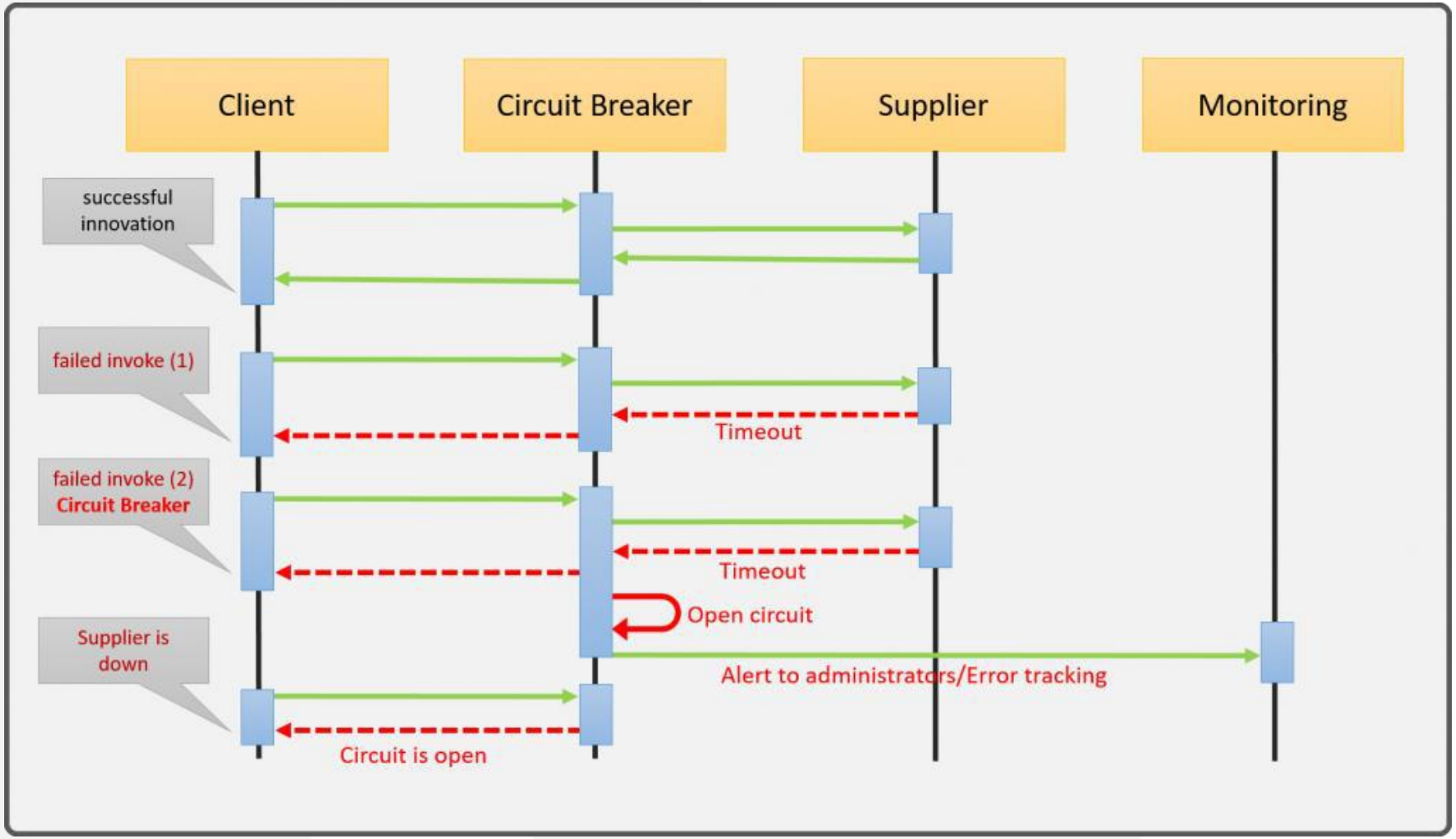
Acciones:

- Informar del error
- Gestionar el problema (se abre el circuito)

Ventajas:

- **Monitorización**: el sistema está controlado en tiempo real. Ante un fallo se disparan las alarmas
- **Sobrecarga**: al abrir el circuito se corta un posible efecto de bola de nieve en todo el sistema, el problema está controlado
- **Tolerancia a fallos:** el Circuit Breaker puede redireccionar la petición al siguiente proveedor en caso de que alguno falle, evitando tener que enviarle el error al cliente.

Circuit Breaker (2/2)



Referencia:

- https://resilience4j.readme.io/docs/circuitbreaker
- https://camel.apache.org/components/3.15.x/eips/resilience4jConfiguration-eip.html

Ciclo de vida de un proyecto Agile

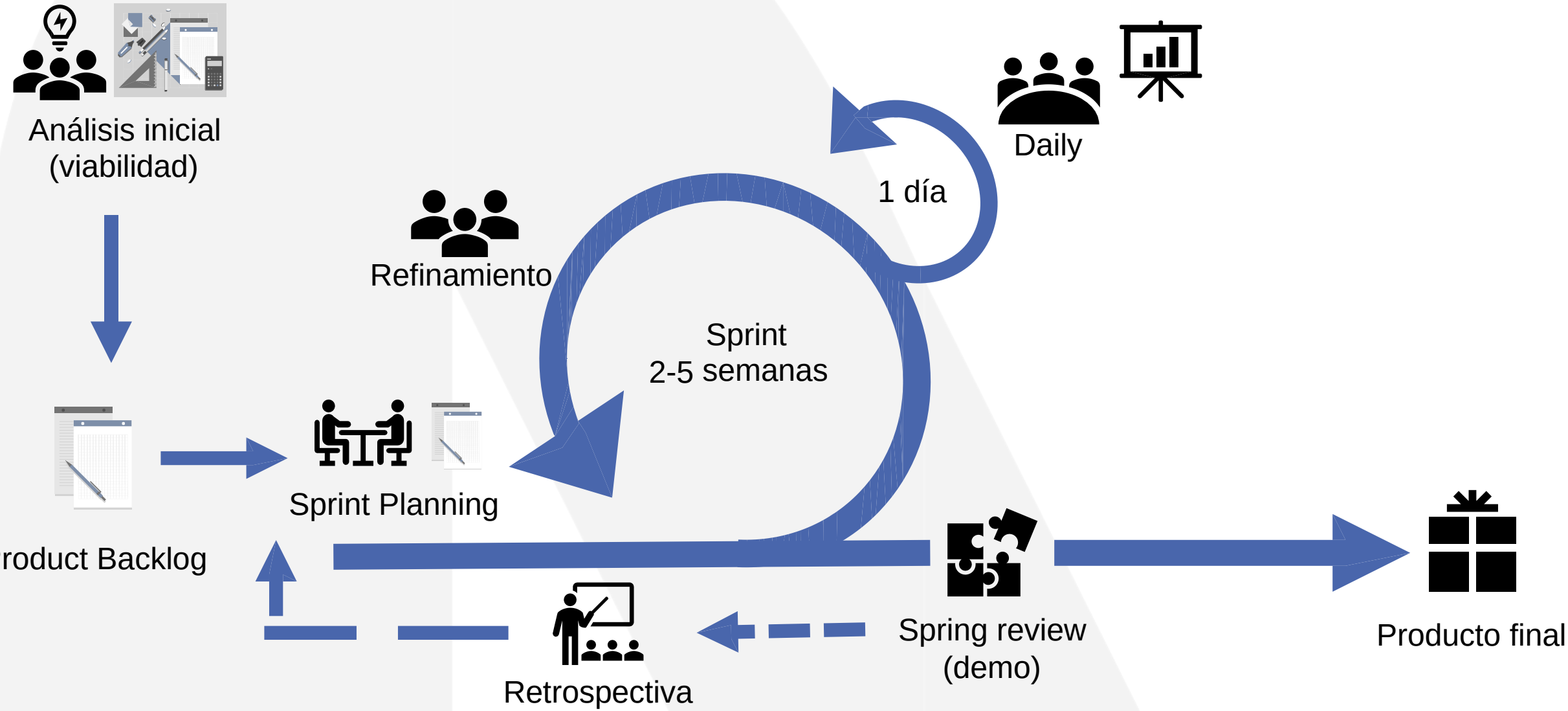
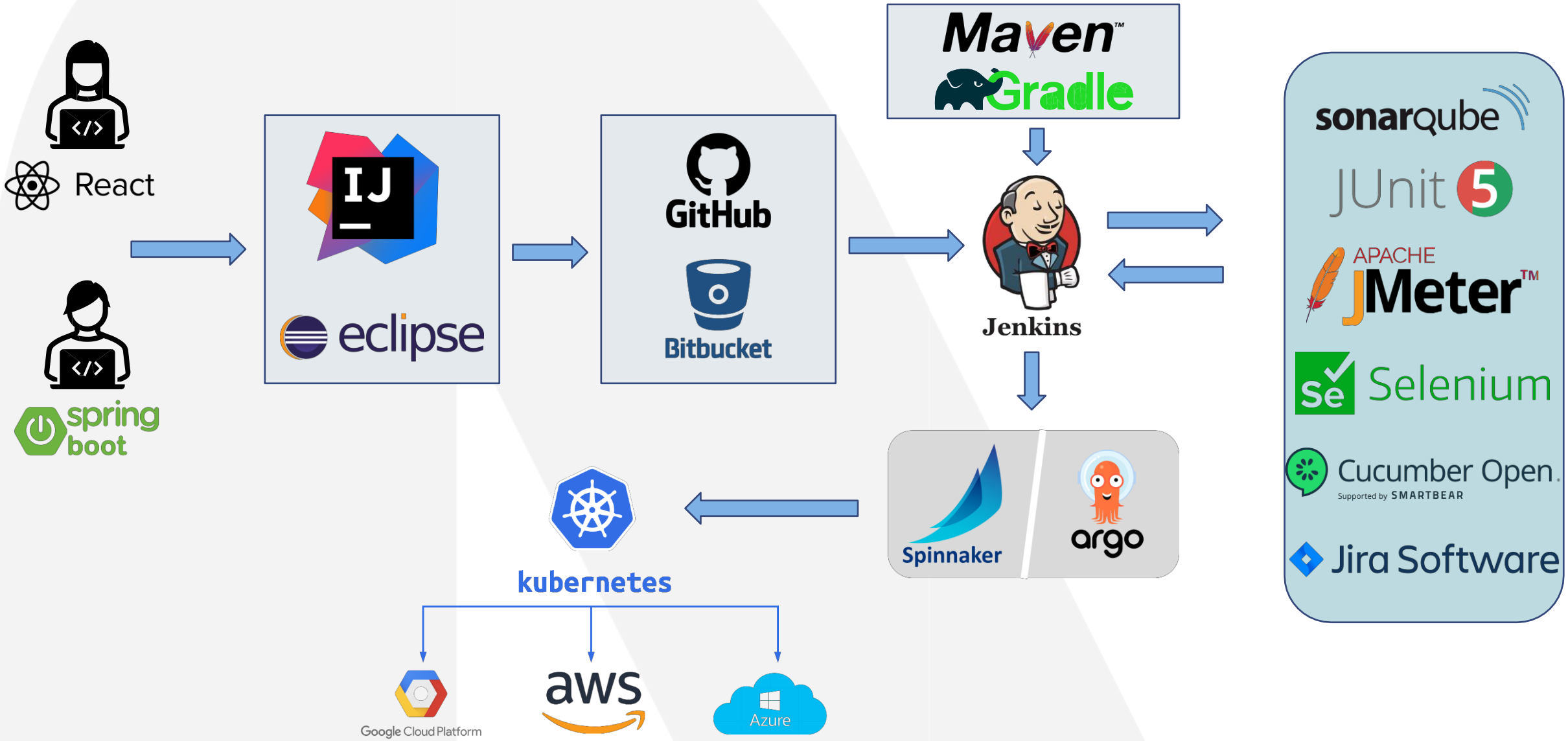
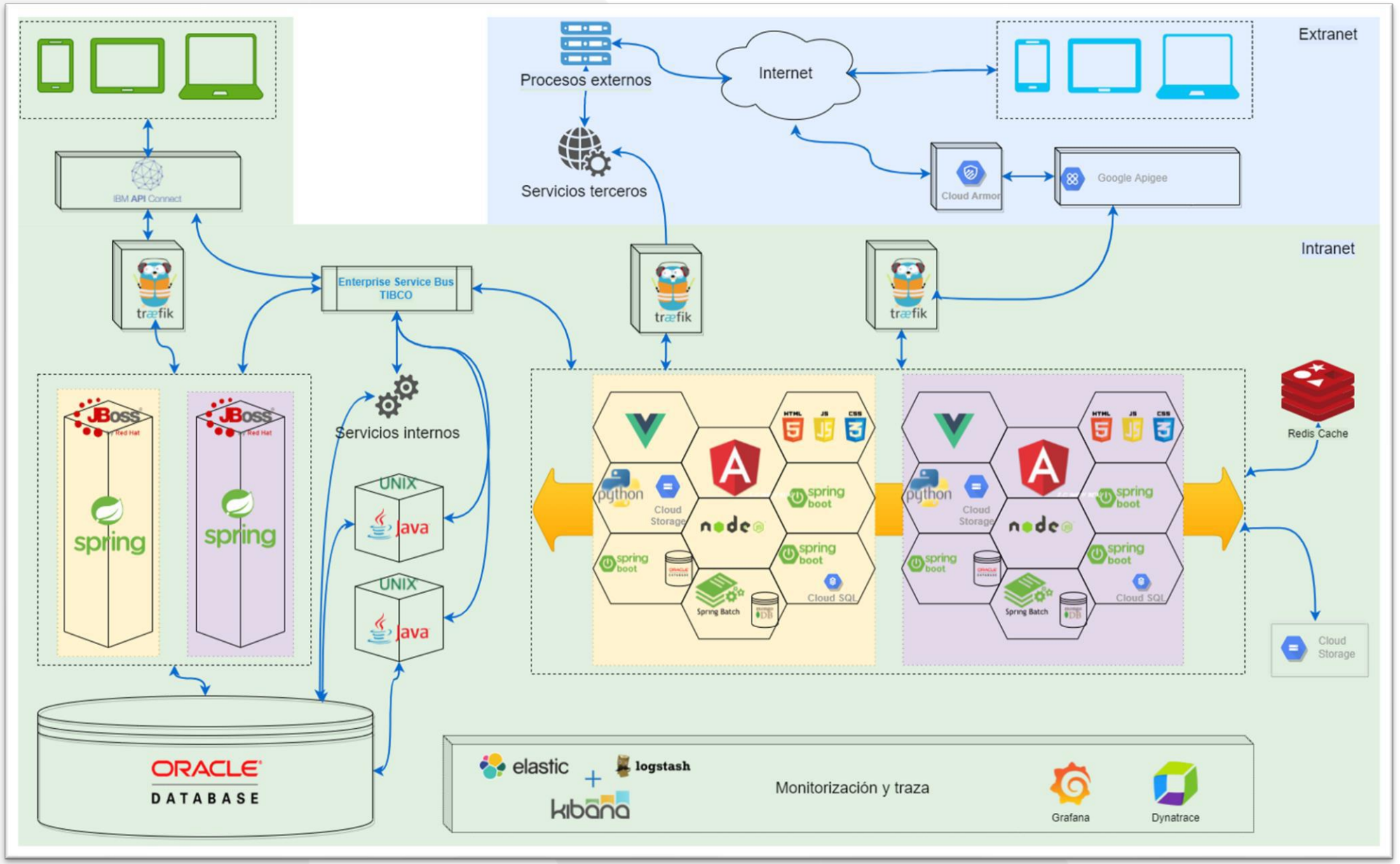


Diagrama despliegue aplicación



Ejemplo de infraestructura



Ejemplo Supermercado: Migración Pedido Tiempo Real (1/4)

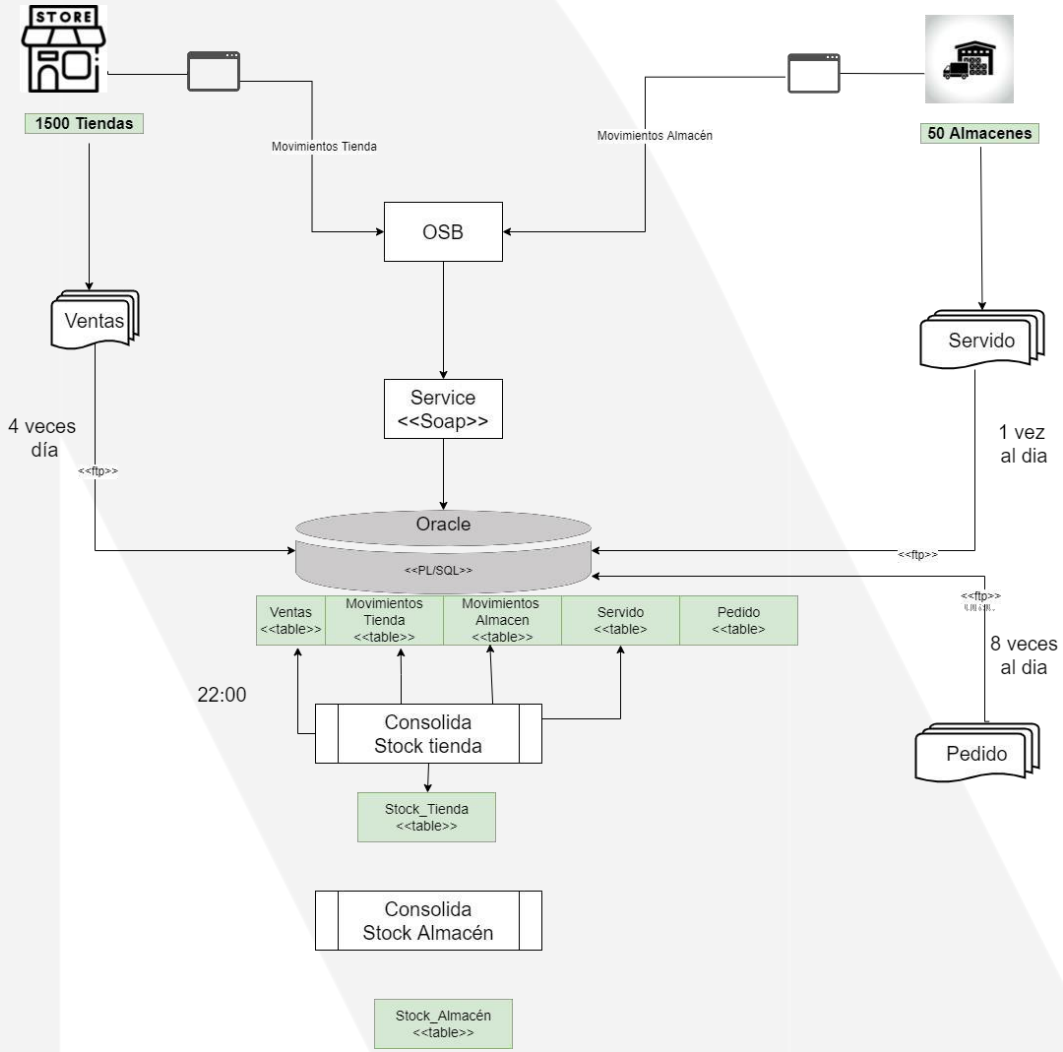
Objetivo: Una cadena de supermercados, con mas de 1500 tiendas quiere migrar su forma de gestionar el stocks actual, (se actualiza una vez al día) a poder consultarlo en tiempo real.

Situación actual:

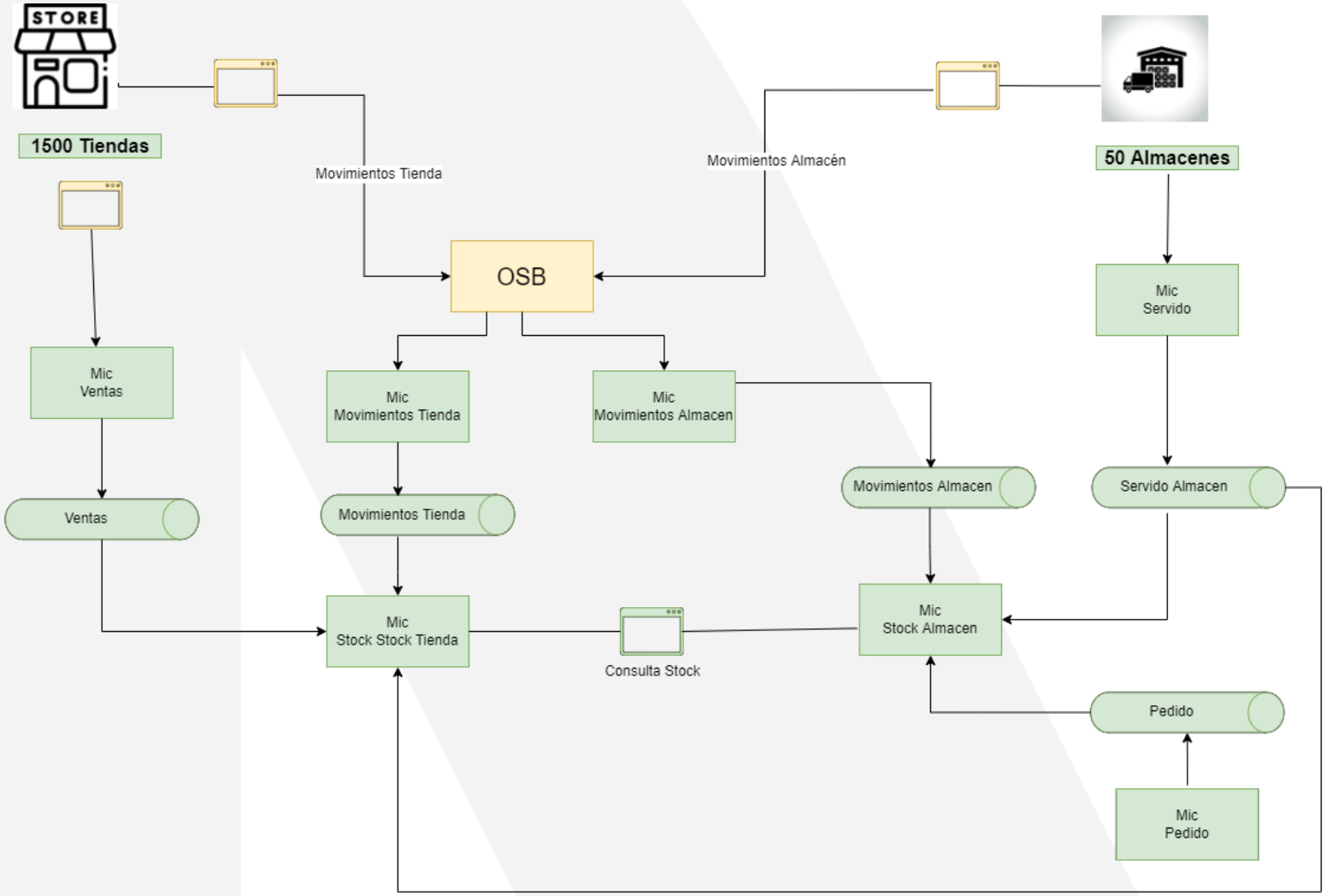
- Tienda:
  - Movimientos Tienda: Se envían a través de un terminal/tablet a través de un servicio SOAP al sistema de gestión centralizada según se van produciendo
    - Autoconsumos
    - Robos
    - Recuentos
    - ….
  - Ventas: Cada tienda envía por fichero plano 4 veces al día al sistema de gestión centralizada

- Almacén:
  - Movimientos Almacén: Cada almacén envía a través de una aplicación web, los diferentes movimientos que se van produciendo
    - Autoconsumos
    - Roturas
    - Decomisos
    - …
  - Servido: Se envía el servido a tienda una vez al día, vía fichero plano
  - Pedido: Se envía el pedido (destino almacén 8 veces al día) vía fichero plano

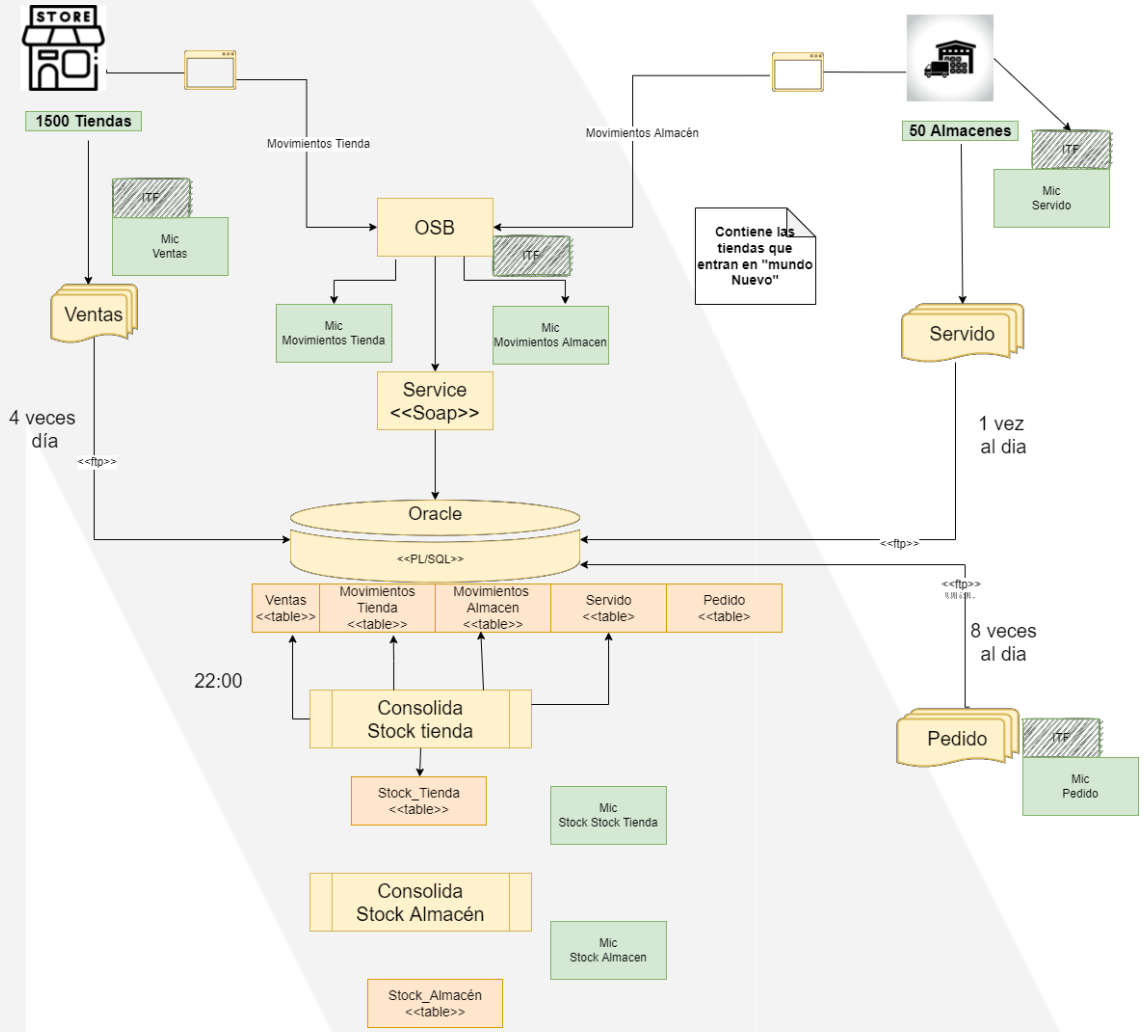
Ejemplo Supermercado: Migración Pedido Tiempo Real Situación Inicial (2/4)



Ejemplo Supermercado: Migración Pedido Tiempo Real Situación Final (3/4)

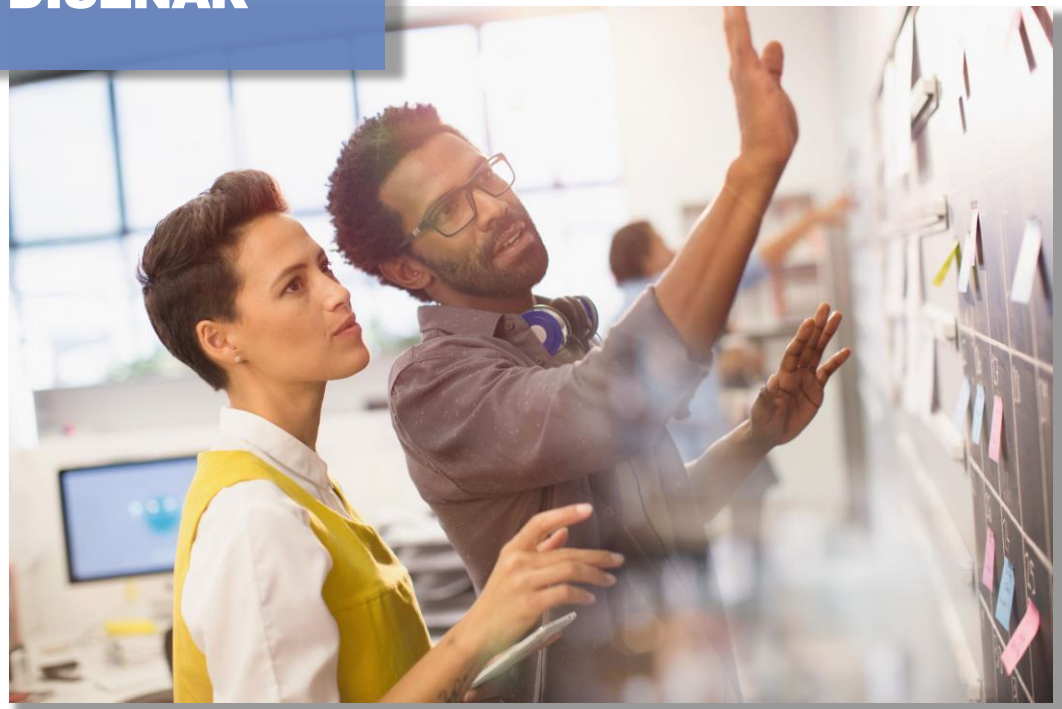


Ejemplo Supermercado: Migración Pedido Tiempo Real Convivencia (4/4)



Demo (Api-First)

1- DISEÑAR



2- CONSTRUIR