Despliegue de aplicaciones web

UT 2.1 – Implantación de arquitecturas web

2 – Servidores – Escalabilidad – Ejemplos de arquitecturas

IES Clara del Rey – Madrid

Arquitectura cliente – servidor

Modelo de diseño de sistemas en el que existen dos roles:

- Cliente: dispositivo, aplicación o proceso que solicita servicios o recursos. Estos servicios o recursos pueden ser páginas web, datos, la ejecución de una tarea, como envío de correo, etc.
- Servidor: el sistema que almacena / gestiona / genera los recursos, y que responde a las solicitudes del cliente.

En este modelo múltiples clientes pueden interactuar con un único servidor. Esto permite centralizar tareas y lógica de negocio, mejora la seguridad, y facilita el despliegue y actualización de sistemas.

Es la arquitectura natural de los desarrollos web.

Servidor

Sistema que proporciona servicios, datos o recursos a otros equipos, aplicaciones o procesos (clientes) en una red.

Acepta peticiones y devuelve respuestas a estas peticiones.

En el desarrollo web se pueden ver involucrados muchos tipos de servidores:

- Servidores web y servidores de aplicaciones
- Servidores de bases de datos
- Otros servidores: correo, archivos, caché, etc.

Antes, la infraestructura se basaba en servidores físicos, ahora lo más habitual es que esté formada por servidores virtualizados, sobre soluciones como AWS, Azure o Google Cloud.

Servidores web y servidores de aplicaciones

Servidores web:

- Procesan las peticiones HTTP, y devuelven contenido, normalmente estático (HTML y CSS).
- Ejemplos: Apache, Nginx, IIS.

Servidores de aplicaciones:

- Facilita la ejecución de aplicaciones dinámicas, gestionando la lógica de negocio, y manejando las interacciones entre cliente y servidor.
- Pueden proporcionar servicios a las aplicaciones que ejecutan, como componentes, autenticación y autorización, sesión, etc.
- Suelen interactuar con otros servidores (BBDD, correo, etc.)
- Ejemplos: Tomcat, GlassFish, Node.js, IIS, Apache (con módulos)

Servidores on-premise y servidores cloud

Servidores on-premise

- Se encuentran en las instalaciones de una organización.
- La organización es responsable de la compra, instalación y mantenimiento del hardware y software.

Servidores cloud (en la nube)

- Virtualizados, en infraestructura proporcionada por proveedores de servicios en la nube, a través de Internet.
- Los recursos se alquilan y escalan según las necesidades.

La elección de un modelo u otro dependerá de factores como coste, escalabilidad o seguridad. Las dos soluciones tienen ventajas y desventajas.

Servidores on-premise y servidores cloud

Característica	Servidores On-Premise	Servidores en la Nube
Coste inicial	Alto, con compra de hardware y software	Bajo, pago por uso, sin grandes inversiones iniciales
Mantenimiento	Gestión interna, requiere personal especializado	Mantenimiento a cargo del proveedor de la nube
Escalabilidad	Limitada, requiere adquisición de nuevo hardware	Alta, se pueden añadir o quitar recursos de forma dinámica
Acceso y disponibilidad	Acceso local, puede ser menos flexible	Acceso global desde cualquier lugar con conexión a Internet
Control y seguridad	Mayor control sobre la infraestructura y datos	Menor control, pero con medidas implementadas por el proveedor
Despliegue	Lento, requiere instalación física y configuración	Rápido, recursos disponibles en minutos

Escalabilidad

Capacidad de un sistema, red o infraestructura para adaptarse a un aumento de la demanda y de la carga de trabajo.

Hay dos tipos de escalabilidad:

- Vertical (scale up): aumentar los recursos de un servidor (CPU, RAM, disco, ...), para mejorar su rendimiento.
- Horizontal (scale out): aumentar el número de servidores, para distribuir la carga entre ellos.

Escalado vertical – Scale up

Aumentar los recursos de un servidor (CPU, RAM, disco, ...), para mejorar su rendimiento.

Ventajas:

- Fácil de implementar, menos cambios.
- Las aplicaciones no necesitan estar preparadas para este escenario.

Desventajas:

- Hay un límite físico en la capacidad.
- Baja tolerancia a fallos. Si el único servidor que da servicio se se estropea.

Escalado vertical – Scale out

Aumentar el número de servidores, para distribuir la carga entre ellos.

Ventajas:

- Mejora la tolerancia a fallos. Si cae un servidor, los demás siguen atendiendo peticiones.
- Con los sistemas actuales, es fácil adaptarse (crecer o decrecer) automáticamente a la demanda.

Inconvenientes

- Las aplicaciones tienen que estar preparadas.
- Puede ser difícil de gestionar al añadir múltiples instancias.

Escalabilidad – Algunas claves

Balanceo de carga: Distribuye las peticiones de los clientes entre un conjunto de servidores. Mejora disponibilidad y el tiempo de respuesta.

Replicación: Replica sistemas (normalmente SGBD) para mejorar la disponibilidad de la información.

Caché: Almacenamiento de muy rápido acceso, para guardar los datos frecuentemente usados. Mejora el rendimiento y el tiempo de respuesta.

Microservicios: una aplicación se divide en múltiples servicios más pequeños, y quasi-independientes. Los microservicios se pueden a su vez escalar en función de la demanda.

Contenedores y orquestación: contenedores (Docker) para empaquetar aplicaciones y dependencias. Orquestación (Kubernetes) para escalar los contenedores. Facilita despliegue y gestión en entornos distribuidos.

Escalabilidad – Balanceo de carga

Distribuye automáticamente las peticiones entre varios servidores para evitar la sobrecarga de uno solo y garantizar un rendimiento óptimo.

- Mejora la disponibilidad: Si un servidor falla, el balanceador redirige las solicitudes a los servidores activos.
- Aumenta la escalabilidad: Se pueden añadir más servidores para manejar un mayor volumen de peticiones.
- Optimiza el rendimiento: Distribuye de manera eficiente las solicitudes para reducir los tiempos de respuesta.

Escalabilidad – Replicación

Se usa fundamentalmente en bases de datos.

Mantiene varias copias idénticas de una base de datos en diferentes servidores, asegurando la sincronización entre nodos. Pueden estar en distintas ubicaciones geográficas.

Busca garantizar la disponibilidad de los datos y ofrecer redundancia, mejorando así la tolerancia a fallos.

Si un servidor falla, las réplicas en otros servidores aseguran que los datos sigan estando disponibles.

Escalabilidad – Replicación

Los datos deben estar sincronizados entre las réplicas. Puede ser:

- En tiempo real (síncrona)
- Con algún retraso (asíncrona)

Hay dos tipos de replicación:

- Replicación maestra-esclavo: Solo el maestro acepta escrituras, y las réplicas (esclavos) manejan las lecturas. Las actualizaciones en el maestro se propagan a los esclavos.
- Replicación maestro-maestro: Todos los nodos pueden recibir lecturas y escrituras, y las actualizaciones se sincronizan entre ellos.

Escalabilidad – Balanceo vs replicación

Característica	Balanceo de Carga	Replicación
Objetivo principal.	Distribuir la carga de trabajo entre varios servidores.	Mantener múltiples copias de datos en diferentes nodos.
Tolerancia a fallos.	Redirige el tráfico a otras instancias si una falla.	Garantiza datos disponibles, aunque un nodo falle.
Enfoque en la disponibilidad.	Mejora disponibilidad de las peticiones de los clientes.	Mejora la disponibilidad de los datos y evita la pérdida.
Datos sincronizados.	No necesariamente implica que los datos estén sincronizados.	Asegura que todas las réplicas contengan los mismos datos.
Escalabilidad.	Se enfoca en escalar el número de servidores para manejar más tráfico.	Se enfoca en la disponibilidad y consistencia datos en varios nodos.
Tipo de fallos que cubre.	Fallos en la sobrecarga de consultas o fallos temporales de un nodo.	Fallos permanentes o catastróficos en el almacenamiento de datos.
Relación entre nodos.	Los nodos pueden no tener datos idénticos.	Los nodos tienen copias idénticas de la base de datos.

Escalabilidad específica Web – CDN

Una CDN es una red de servidores distribuidos geográficamente especializados en entregar contenido web rápidamente.

Almacenan recursos estáticos (que no cambian, o que lo hacen poco frecuentemente) de los sitios web. Archivos CSS, JavaScript, videos...

Beneficios de una CDN

- Reducción de la latencia: al estar más cerca geográficamente.
- Distribución de carga: el servidor principal delega en la CDN.
- Mayor disponibilidad: las CDN ayudan a la tolerancia a fallos.

Servicios CDN son CloudFare, Akamai, el servicio CloudFront de Amazon, o el servicio "Content Delivery Network" de Azure.

Arquitectura de servidores

En la implantación de aplicaciones web, se pueden utilizar multitud de arquitecturas de servidores.

No hay que confundir la arquitectura de la infraestructura con la arquitectura de la aplicación. Aunque relacionadas:

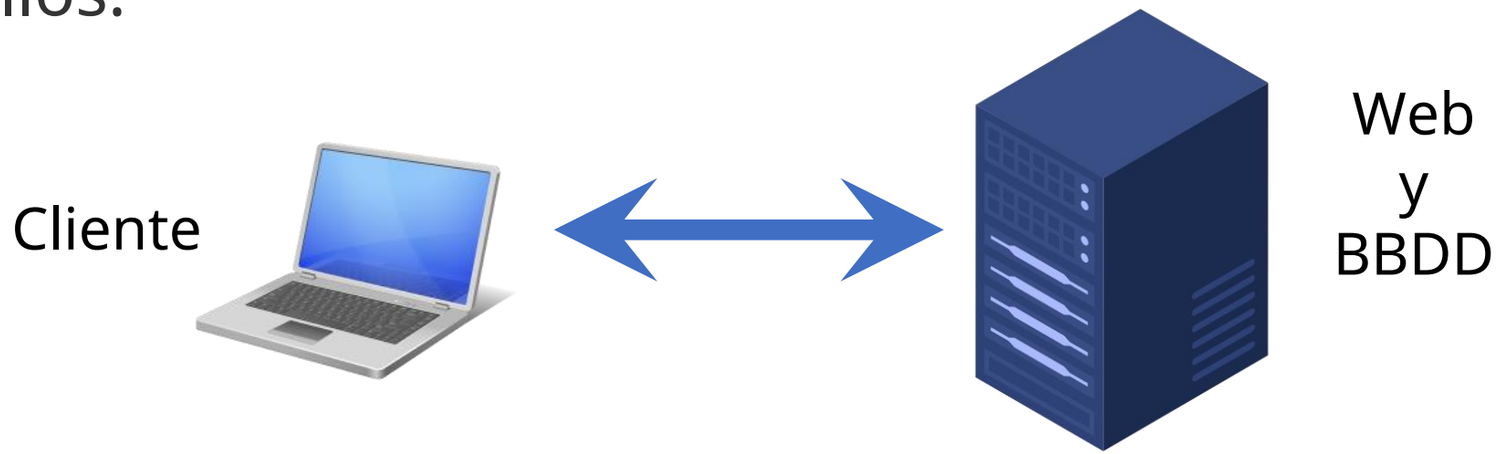
- La arquitectura de infraestructura o de seridores se refiere a cuantos servidores se utilizan, la función de cada uno y como interactúan.
- La arquitectura de la aplicación se refiere a cómo se estructura el código de una aplicación, en que capas o componentes, y como se relacionan.

Aunque sean aspectos claramente diferentes, están interrelacionados, uno condiciona el otro, y viceversa.

Arquitectura de servidores
Servidor único

Un único servidor aloja la aplicación web y la base de datos.

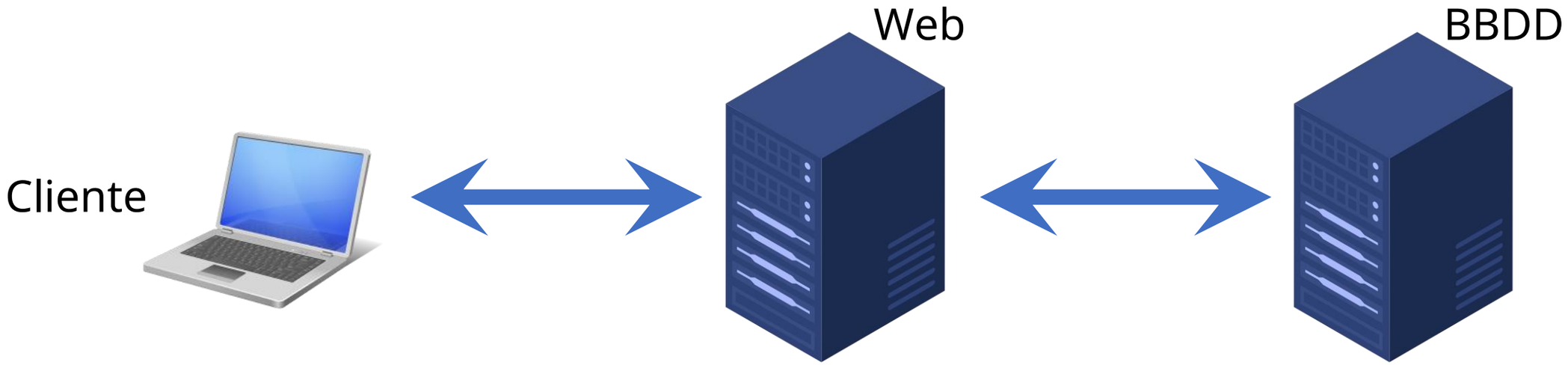
- Adecuado para aplicaciones pequeñas o en fase de desarrollo.
- Bajo coste inicial
- Simplicidad de administración
- Limitaciones en escalabilidad y tolerancia a fallos.



Arquitectura de servidores
Servidor web + servidor de BBDD

Un servidor dedicado a la aplicación web, y otro para la base de datos

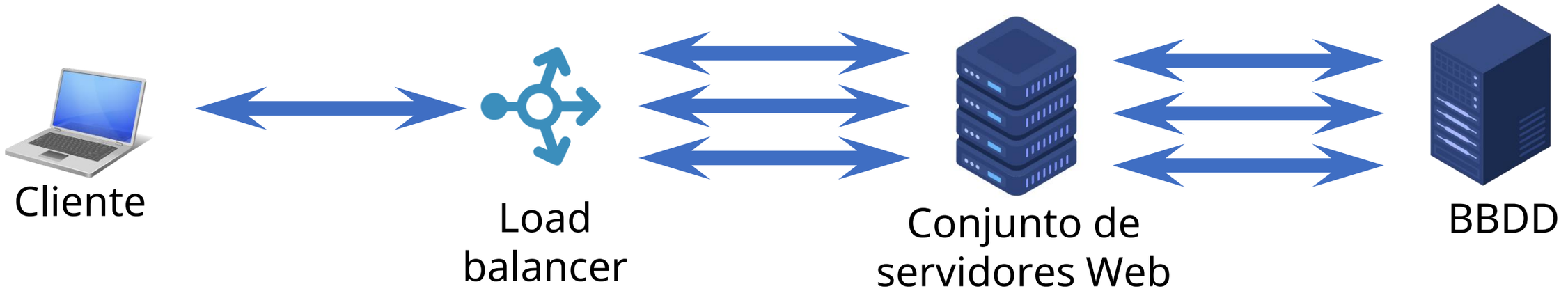
- Mejora el rendimiento y permite distribuir los recursos.
- Mejora la seguridad, al separar servicios.
- Mejora levemente la escalabilidad. No mejora tolerancia a fallos.



Arquitectura de servidores
Balanceo de carga sólo en web

Múltiples servidores para la plataforma web, pero sólo uno para la BD.

- Mejora el tiempo de respuesta y la tolerancia a fallos de la parte web.
- El servidor de BD es el punto crítico. Si falla, falla todo el sistema.
- Mejora la escalabilidad, mejora la tolerancia a fallos en la parte web, pero no mejora tolerancia a fallos en lo que respecta a la BBDD.



Arquitectura de servidores
Balanceo de carga en web y en BBDD

Múltiples servidores para la plataforma web, y también en la BD.

- Mejora el tiempo de respuesta y la tolerancia a fallos de la parte web, y también de la parte de acceso a datos.
- Mejora la escalabilidad, mejora la tolerancia a fallos en la parte web, y también en la parte de acceso a la base de datos.
- Ante un fallo en un servidor de BBDD, la información estará replicada en otros.

Arquitectura n-niveles

A veces se confunde n-niveles con n-capas, y no es lo mismo:

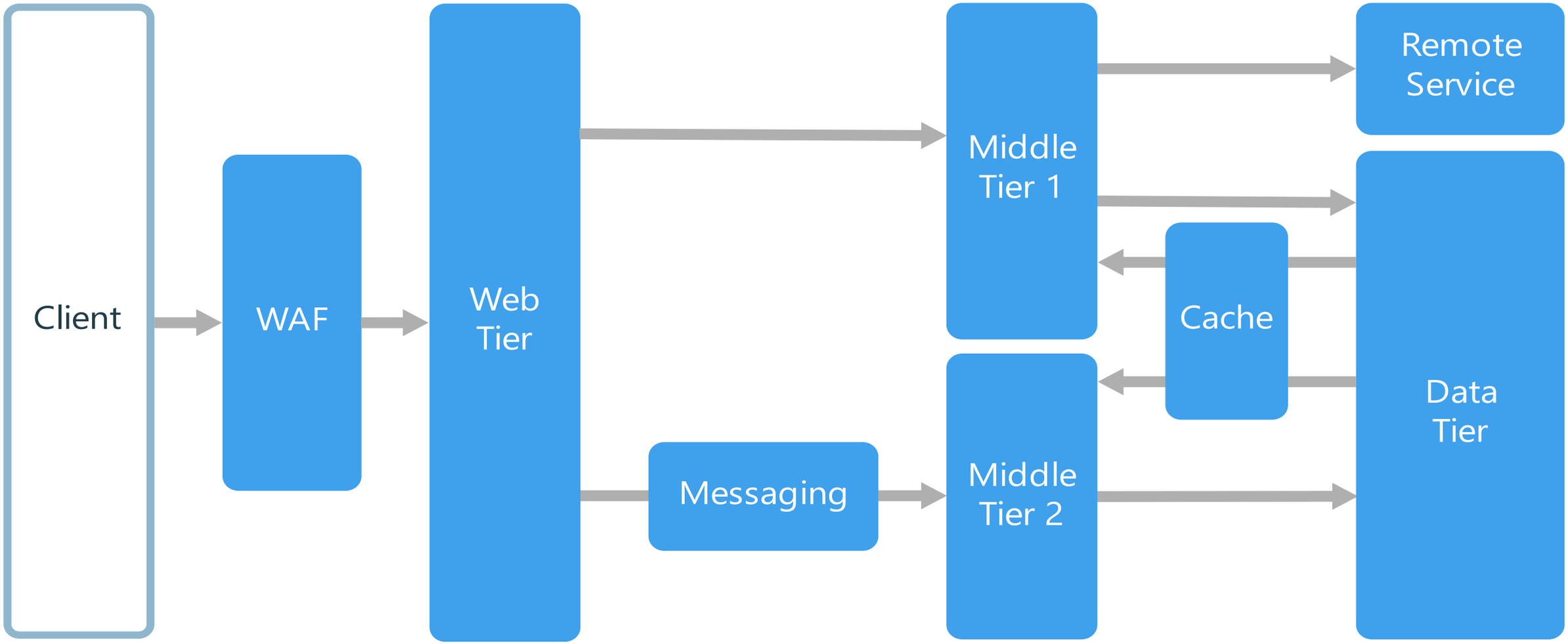
Capas:

- **División lógica del software.** Cada capa tiene una función.
- Ejemplo: presentación, lógica de negocio, acceso a datos.
- Una aplicación monolítica puede estar organizada en n-capas.

Niveles:

- **División física** de los componentes del sistema. Pueden estar en diferentes máquinas o entornos, incluso distintas ubicaciones.
- Ejemplo: cliente (UI), servidor de aplicaciones (negocio) y servidor BD.
- Comunicación: Generalmente a través de una red o servicios.

Arquitectura n-niveles



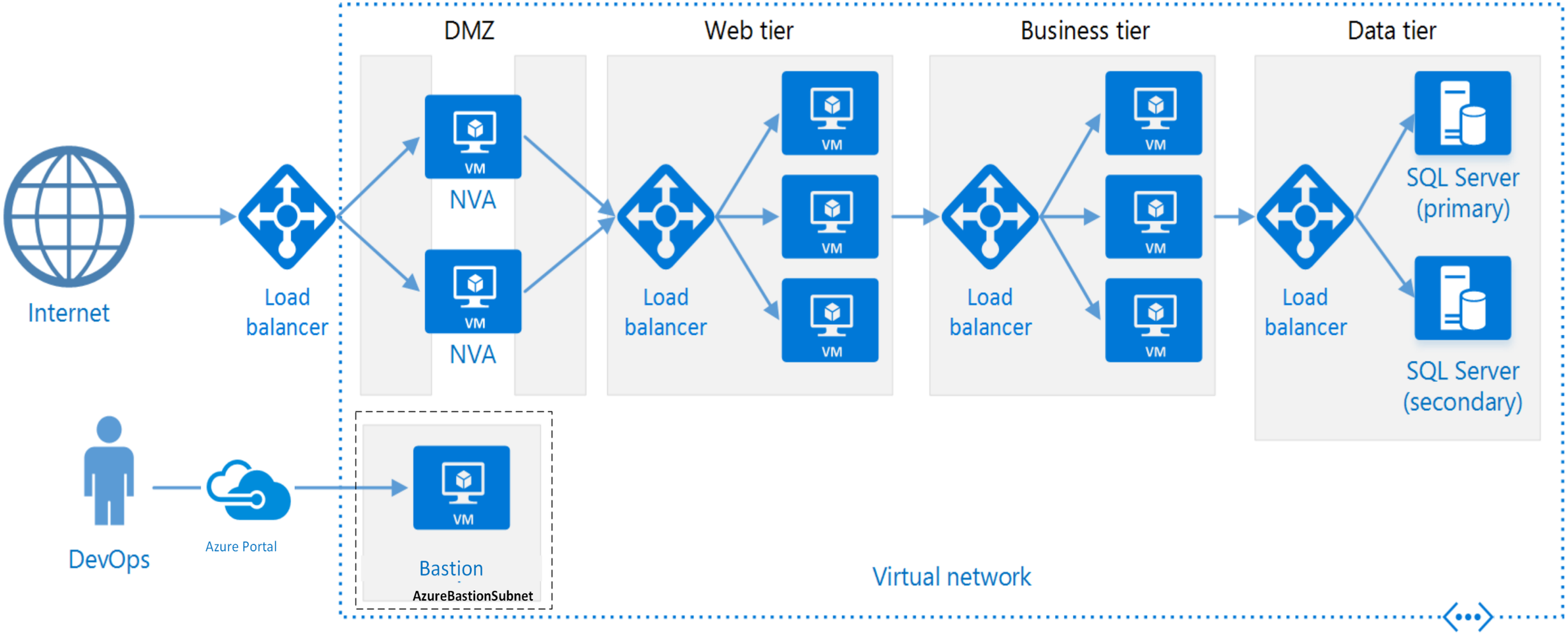
Arquitectura n-niveles

- Los niveles están físicamente separados (en máquinas diferentes).
- Dos modos: estricto o relajado.
  - Estricto: La solicitud debe pasar por los niveles adyacentes, uno a uno, y no puede saltarse ningún nivel intermedio.
  - Relajado: La solicitud puede saltarse algunos niveles si es necesario.
- El estricto tiene más latencia y sobrecarga. El relajado más acoplamientos y es más difícil de cambiar.
- Puede ser híbrido: niveles relajados o estrictos según sea necesario.

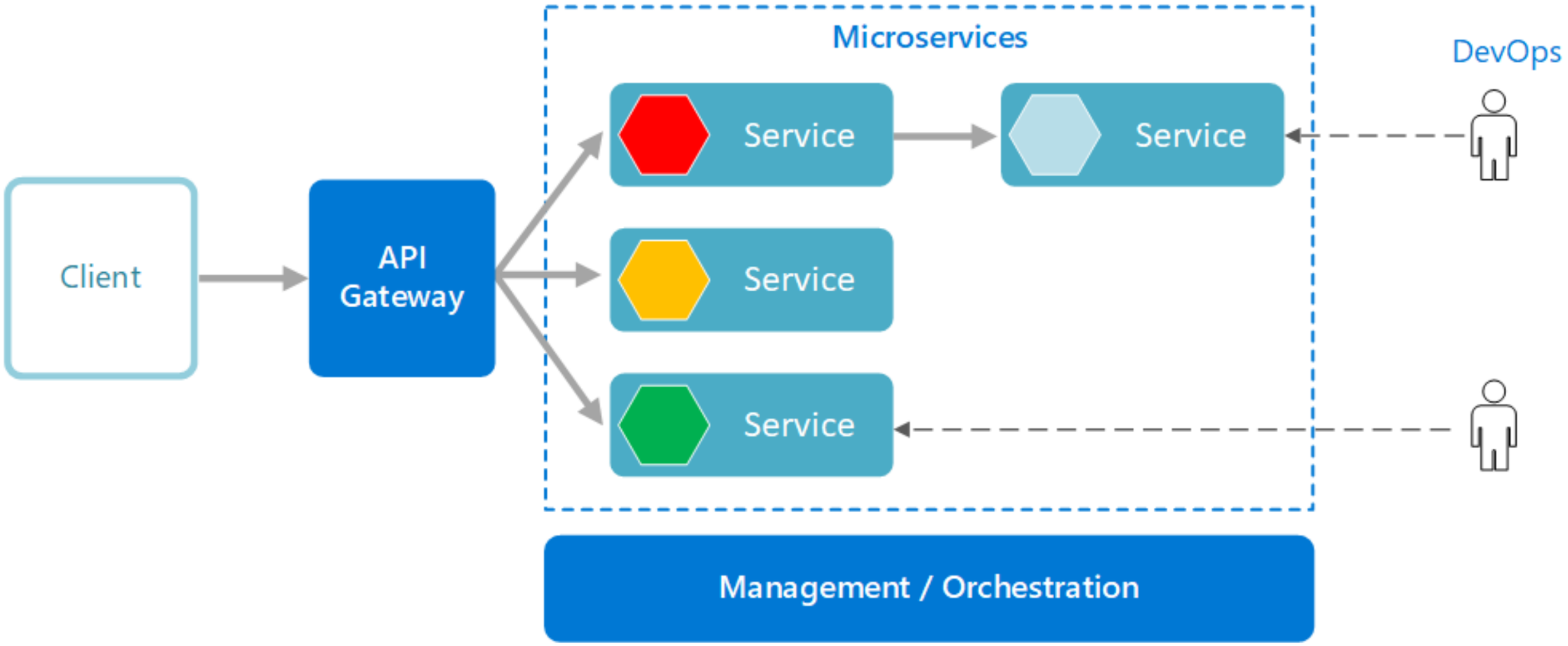
Arquitectura n-niveles

- Un nivel puede llamar a otro nivel directamente o usar patrones de mensajería asíncrona, usando, por ejemplo, colas de mensajes.
- Aunque cada capa (lógica) se puede hospedar en su propio nivel, esto no es necesario. Se pueden hospedar varias capas en el mismo nivel.
- La separación física de los niveles mejora la escalabilidad y la resistencia, pero también agrega latencia.
- Esquema tradicional: nivel de presentación, un nivel intermedio y un nivel de datos. El nivel intermedio es opcional. Las aplicaciones más complejas pueden tener más de tres niveles.

Arquitectura n-niveles - Ejemplo



Arquitectura de microservicios



Arquitectura de microservicios

- Los microservicios son pequeños e independientes.
- Acoplados de forma imprecisa. No hay grandes dependencias entre ellos. Un único equipo reducido puede escribir y mantener un servicio.
- Cada servicio es un código base independiente, que puede administrarse por un equipo de desarrollo pequeño.
- Los servicios pueden implementarse de manera independiente. Se puede actualizar un servicio sin volver a generar toda la aplicación.
- Los servicios son los responsables de conservar sus propios datos o estado. Esto difiere del modelo tradicional, donde una capa de datos independiente controla la persistencia de los datos.

Arquitectura de microservicios

- Los servicios se comunican entre sí mediante API bien definidas. Los servicios que dependen unos de otros dependen de la interfaz de la API, no de los detalles.
- Los detalles de la implementación interna de cada servicio se ocultan frente a otros servicios.
- Admite múltiples lenguajes de programación. No es necesario que compartan pila de tecnología, bibliotecas o frameworks.
- Si los servicios se tienen que exponer al exterior, se suele usar una “API Gateway”, que se encarga de “enrutar” las llamadas a los microservicios.

Arquitectura de microservicios

Ventajas:

- Agilidad en la modificación y la actualización al ser independientes.
- Equipos de desarrollo más pequeños y enfocados.
- Base de código más pequeña. Es más fácil de mantener.
- Mezcla de tecnologías. Pueden incorporarse nuevas tecnologías al sistema sin necesitada de rehacerlo por completo.
- Escalabilidad. Se pueden escalar de forma independiente.
- Aislamiento de datos. Cada servicio debe manejar sus datos. Se puede modificar el esquema de datos sin afectar a otros servicios.

Arquitectura de microservicios

Posibles inconvenientes:

- Complejidad. El sistema tiene más partes independientes, que una por una son más sencillas, pero en conjunto es más complejo.
- Desarrollo y pruebas. Si hay dependencias entre servicios, probarlos puede ser algo más complejo que en una aplicación monolítica.
- Falta de homogeneidad. Se puede acabar con una variedad de lenguajes y tecnologías muy diversas.
- Integridad de datos. Hay que diseñarlos teniendo en cuenta la independencia de los servicios.
- Administración. Debe existir alguien que conozca bien la arquitectura.