Tabla de contenido

[Introducción a la Arquitectura de Microservicios (MSA) 2](#_Toc79446231)

[Temario 2](#_Toc79446232)

[1. Introducción. 2](#_Toc79446233)

[2. Enfoque monolítico vs enfoque microservicios. 2](#_Toc79446234)

[3. Características de la arquitectura de microservicios. 2](#_Toc79446235)

[4. Ventajas de la arquitectura de microservicios. 2](#_Toc79446236)

[5. Desventajas de la arquitectura de microservicios. 2](#_Toc79446237)

[6. ¿Cuándo utilizar una arquitectura de microservicios? 2](#_Toc79446238)

[7. Patrones relacionados. 2](#_Toc79446239)

[Microservicios 2](#_Toc79446240)

[Monolítico vs Microservicios 3](#_Toc79446241)

[Estilo de arquitectura monolítico: 3](#_Toc79446242)

[Ventajas del enfoque monolítico: 3](#_Toc79446243)

[Inconvenientes del enfoque monolítico: 4](#_Toc79446244)

[MSA - Características 6](#_Toc79446245)

[Principales características de una arquitectura de microservicios: 6](#_Toc79446246)

[Componentización a través de servicios. 7](#_Toc79446247)

[Organizada en torno a funcionalidades de negocio. 7](#_Toc79446248)

[El enfoque en microservicios es diferente: 8](#_Toc79446249)

[Productos, no proyectos. 9](#_Toc79446250)

[Enfoque de proyecto: 9](#_Toc79446251)

[Mentalidad de producto: (ídem a lo anterior pero no termina en producción, nosotros seguimos después) 9](#_Toc79446252)

[Smart endpoints and dumb pipes. 9](#_Toc79446253)

[Gobierno descentralizado 10](#_Toc79446254)

[Min 57’ 10](#_Toc79446255)

[La aplicación esta partida en microservicios, pero todo está partido en microservicios. 10](#_Toc79446256)

[Gestión de datos descentralizada. 10](#_Toc79446257)

[Teorema de CAP 11](#_Toc79446258)

[Automatización en Infraestructura 12](#_Toc79446259)

[Diseño tolerante a fallos 12](#_Toc79446260)

[Diseño evolutivo 13](#_Toc79446261)

[METRICAS 13](#_Toc79446262)

[MSA - Ventajas 14](#_Toc79446263)

[MSA – Inconvenientes 14](#_Toc79446264)

[¿Cuándo utilizar una MSA? 15](#_Toc79446265)

[Patrones Relacionados 15](#_Toc79446266)

[Referencias 16](#_Toc79446267)

# Introducción a la Arquitectura de Microservicios (MSA)

Es un enfoque para desarrollar una aplicación, la anterior era una arquitectura organizacional que miraba todas las aplicaciones, y esto es para una solución determinada y su conjunto se lo ve como pequeños servicios, se ahí la palabra microservicios. Y el microservicio se ejecuta como una aplicación, como otro proceso, entonces las comunicaciones, serán comunicaciones entre procesos y no intraprocesos, entonces esas comunicaciones entre procesos usaran mecanismos más livianos, la definición de cada microservicio se van a estar elaborando de acuerdo a la capacidad de negocio, y esta división de la aplicación en pequeños servicios arrastra al despliegue de la solución, es decir cada microservicio debe ser posible de forma independiente a través de procesos automatizados. Esos pequeños microservicios hay que gestionarlos, deben ser articulados para que sea establecido lo que haga, como tiene diferente interdependencia, cada uno puede ser escrito en diferentes lenguajes.

Antes las aplicaciones no estabas divididas en pequeños servicios, entonces el esquema anterior se lo llamo arquitectura monolítica o esquema monolítico de construcción de aplicaciones (monolito: piedra construida por una sola pieza, una unidad indivisible), entonces su aplicación es un todo, pero no impide que tenga partes, hay partes principales como página principal del cliente, en otro extremo la base de datos (relacional o no) y por otro lado el servidor que maneja las vistas html (que maneja solicitudes http y aplicara cierta lógica de negocio)

Hay aplicaciones monolíticas exitosas, pero ha aparecido un nivel de frustración importante especialmente cuando se implementa aplicaciones cloud.

Importante: cuando aparece un nuevo enfoque, es una mejora sobre la situación anterior, es decir resuelve mejor ciertos `roblemas, resuelve mas problemas, pero no es que la solución se haya deteriorado, sino por que la realidad cambia, y un esquema de antes no sirve para el ahora

## Temario

## 1. Introducción.

## 2. Enfoque monolítico vs enfoque microservicios.

## 3. Características de la arquitectura de microservicios.

## 4. Ventajas de la arquitectura de microservicios.

## 5. Desventajas de la arquitectura de microservicios.

## 6. ¿Cuándo utilizar una arquitectura de microservicios?

## 7. Patrones relacionados.

# Microservicios

Estilo de arquitectura en el que una aplicación se desarrolla como un conjunto de servicios que:

● Ejecutan en su propio proceso y se comunican con mecanismos ligeros.

● Se construyen en torno a capacidades de negocio.

● Pueden desplegarse de forma independiente mediante procesos automatizados.

● Poseen una mínima gestión centralizada.

● Pueden escribirse en diferentes lenguajes de programación y

utilizar diferentes tecnologías de almacenamiento de datos.

# Monolítico vs Microservicios

## Estilo de arquitectura monolítico:

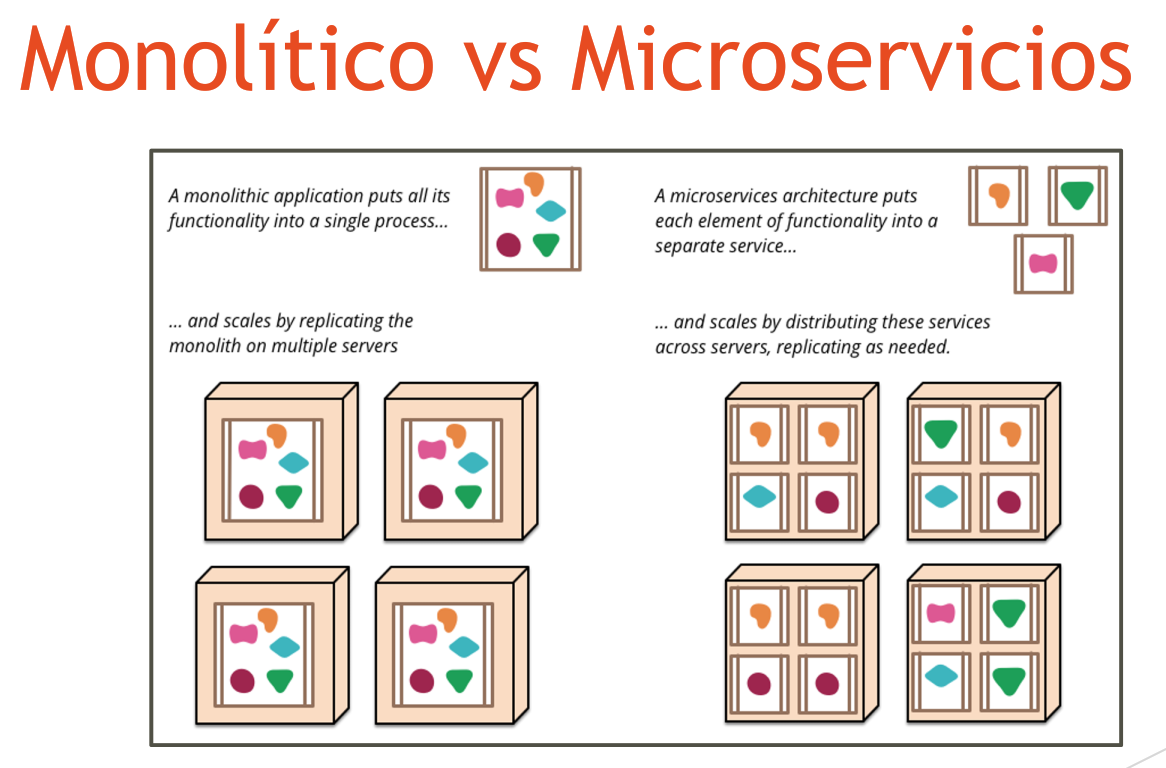
● La aplicación se construye como una unidad (*monolito*).

● Se utilizan las herramientas del lenguaje de implementación para modularizar la aplicación (clases, funciones, namespaces).

● Todos lo módulos ejecutan dentro de un mismo proceso y sobre un mismo hardware.

● Cualquier cambio implica construir y desplegar una nueva versión de toda la aplicación.

● El escalado requiere que escale toda la aplicación.



## Ventajas del enfoque monolítico:

● Simplicidad. : mas fácil de manejar

● Facilidad de desarrollo.: donde todos tienen un mismo enfoque

● Facilidad de testeo.: idem

● Facilidad de despliegue.: idem, pero no significa que sea rápido o que no genere ruido. Mas bien es si eso se pone en producción entonces esa tarea no es complicada

● Facilidad de operación.: una vez que se despegó para la gente que es responsable que esa aplicación siga funcionando, en monolítico, es mas sencillo que operar que otros.

● Facilidad de escalado.: permite escalado horizontal, por ejemplo, ampliar la capacidad de atención. Esto no siempre es fácil, no siempre es lo que queremos.

## Inconvenientes del enfoque monolítico:

● Ciclos de cambios acoplados.: debido a que la puesta en producción es bloque en conjunto, entonces hay una tendencia a no quiero hacer un gran trabajo en producción, no quiero hacerlo solo por un pequeño cambio, cualquier cambio que haga, un cambio o una docena de cambios, siempre voy a tener el mismo costo. Entonces hay un gran acoplamiento de cambios por la puesta en producción, no de punto de vista lógico

● A medida que la aplicación crece: la aplicación como el todo, cada vez es más complicado, es esquema modular tiene dificultades para mantenerse.

○ Aumenta la complejidad del monolito.

○ Aumenta la dificultad de mantener el modularidad.

○ Se ralentiza el tiempo de inicio de la aplicación.

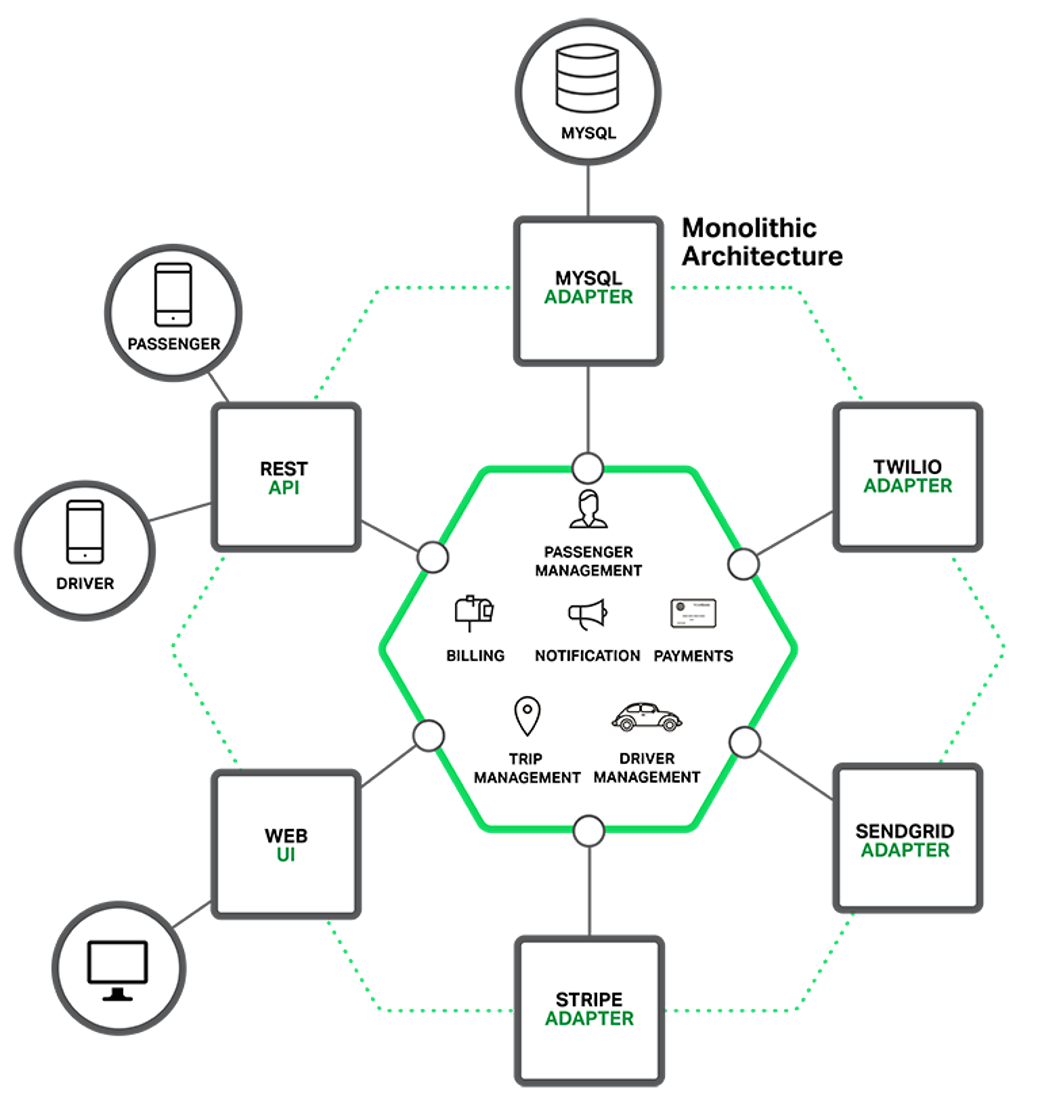
○ Se dificulta la incorporación de nuevas tecnologías.

○ Disminuye la fiabilidad; un error en cualquier módulo puede potencialmente afectar la disponibilidad de toda la aplicación.

● Escalado ineficiente.: el escalado es fácil si, pero es ineficiente en el sentido de que el escalado es del monolito, puedo tener 2,3 y 4, pero no puedo tener un décimo, independientemente de que haya una parte de la aplicación sea demandante. **Si algo no da abasto es la aplicación en monolítico.**

○ Escala toda la aplicación.

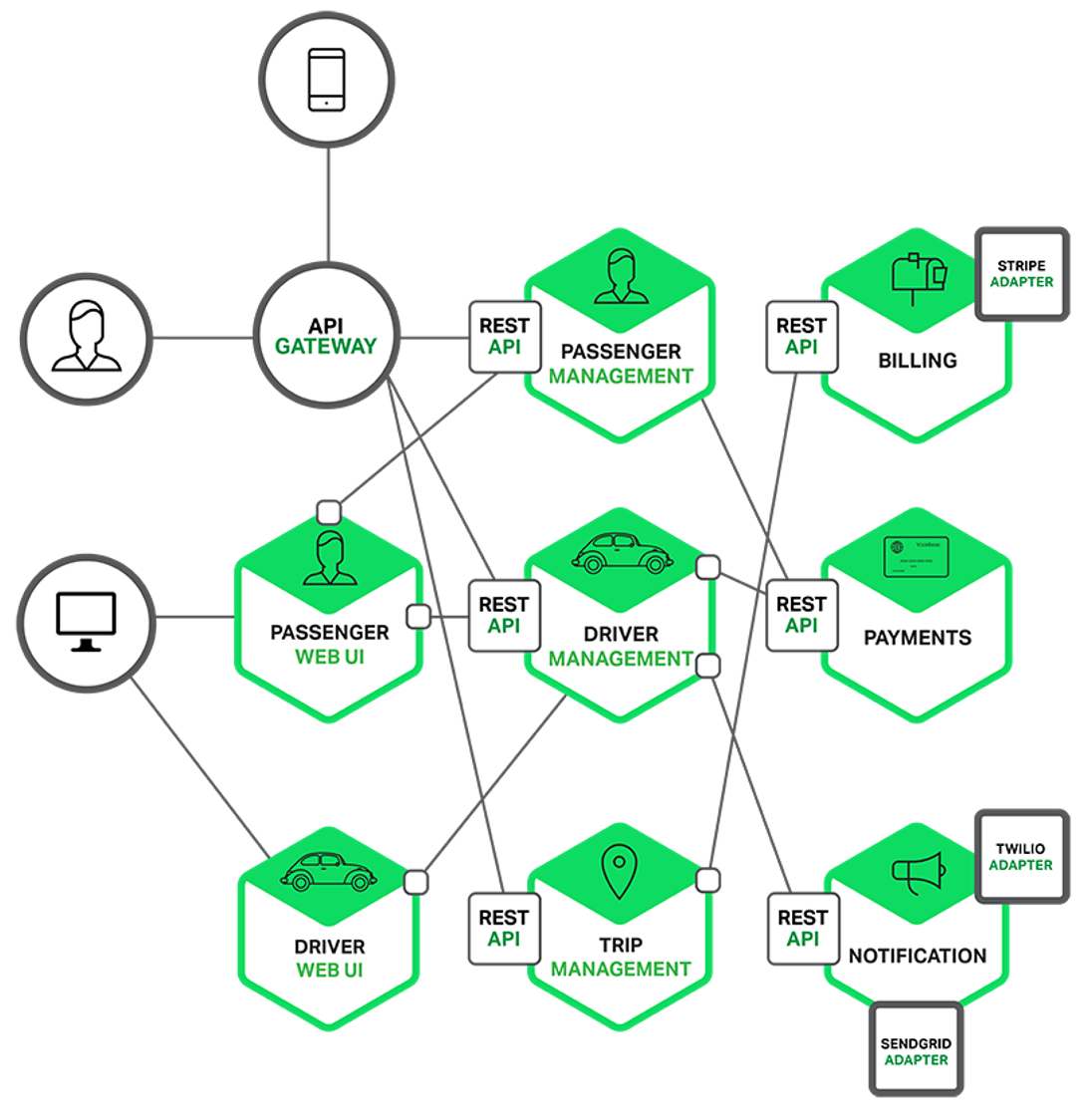
■ Todos los módulos ejecutan sobre el mismo hardware.

ejemplo de arquitectura modular hexagonal:

la parte marcada en verde (el nucleo) tiene la logica del negocio que se implementa mediante modulos que define servicios , objetos de dominio, eventos, etc

alrededor tenemos distintos adaptadores, cuadrados, que permiten la interaccion con el mundo externo, asi tenemos adaptadores a BD, una Api que implementa una interfaz de usuario por ejemplo.

El esquema modular a la hora de desplegar la aplicación, todo esto constituye un slo paquete, un bloque unico, el formato dependera del lenguaje, del framework



Algunas organizaciones como amazon, netfliz, resolvieron esto como arquitectura de microservicios, entonces en lugar de crear un aplicación monolitico, entonces se enfrentaria un problema grande o de categoria mounstruo, entonces se define un conjunto de pequeños servicios interconectados, un servicio o micro, implemeneta un conjunto de carctaristicas de gestion de negocio como gestion de pedidos, gestion de clientes, es una mini aplicación que ahí representamos como un hexagono para seguir el esquema que tenemos antes, ahora los hexagonos estan a microservicios

Los microservicios consumen una api, tambien exponen una api que es usada por otros microservicios, ent tiempo de ejecucion, cada instancia es una maquina virtual en la nube o un contenedor en dock o docker.

Por ahora decimos que cada microservicio corre en una maquina virtual

# MSA - Características

## Principales características de una arquitectura de microservicios:

● Componentización a través de servicios.

● Organizada en torno a funcionalidades de negocio.: es decir como armamos distintos servicios, se resuelve teniendo en cuenta a partir de funcionalidades de negocio. Cuantos microservicios y que va hacer cada uno no es tarea sencilla, ni por su puesto trivial.

● Productos, no proyectos.: enfoque orientado a productos

● Smart endpoints and dumb pipes. : tubería que conecta dos extremos, la inteligencia esta en los extremos, y la tubería que hay en los extremos que hace que se comuniquen entre si, y lo que decimos que la inteligencia esta en los extremos, la tubería de los extremos tiene poco grado de inteligencia, no necesariamente tonto.

● Gobierno descentralizado.

● Gestión de datos descentralizada.

● Automatización de infraestructura.

● Diseño tolerante a fallos.: diseño que contemple fallos

● Diseño evolutivo.

## Componentización a través de servicios.

MSA piensa a las aplicaciones como un conjunto de servicios que interactúan entre sí, eso es el sello de este enfoque

Los servicios corren de un proceso, cada servicio entonces:

Comunicación entre servicios== comunicación entre procesos

Comunicación entre componentes== comunicación intraprocesos, lo mismo con las bibliotecas

Las MSA utilizan bibliotecas, pero su forma principal de crear componentes de su propio software es segmentarlo en servicios.

● Componente: unidad de software que se puede reemplazar y actualizar independientemente.

● Bibliotecas: componentes vinculados a un programa que se invocan mediante llamadas a funciones en memoria.

● Servicios: componentes fuera del proceso que se comunican

mediante mecanismos como peticiones a web services o RPC.

Utilización de servicios como componentes.

● Se basa en que los servicios son independientemente desplegables.

● Resulta en interfaces de componentes más explícitas.

● Tiene algunas desventajas:

○ Las llamadas remotas son más costosas que las invocaciones

en memoria.

○ Se dificulta el cambio de asignación de responsabilidades entre componentes cuando los movimientos de comportamientos cruzan los límites del proceso.

## Organizada en torno a funcionalidades de negocio.

Ley de Conway:

*“Cualquier organización que diseñe un sistema (definido ampliamente) producirá un diseño cuya estructura es una copia de la estructura de comunicación de la organización.”*

- Melvyn Conway, 1967

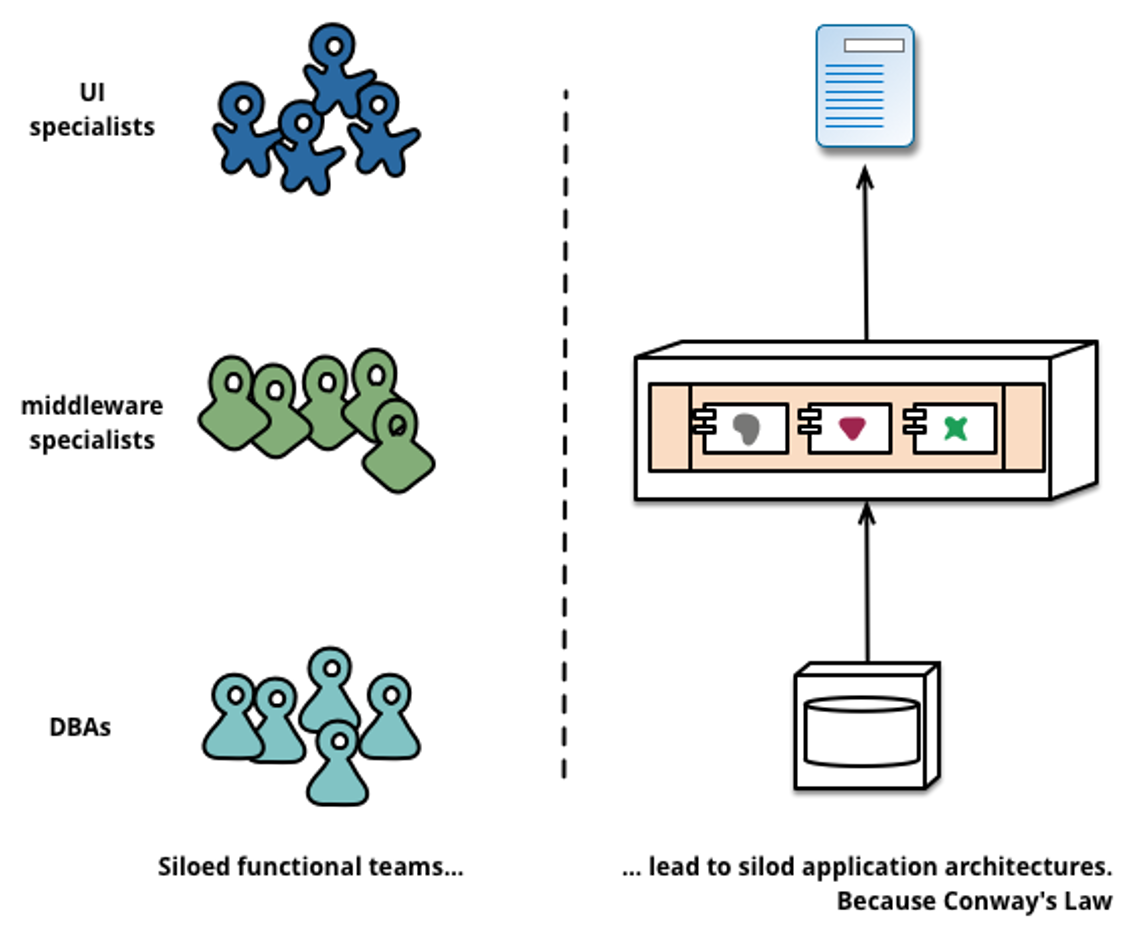
En una organización, las distintas personas que la constituyen, sectores y áreas de la misma, su forma de comunicar, afecta a la forma en la que las personas que forman parte de la organización, ven a la organización, entonces cuando arman la solución, suelen replicar bastante el esquema de la comunicación de la organización, en como se comunican los sectores de la organización.

Equipos funcionales aislados:

● Incluso los cambios simples pueden llevar a que un proyecto entre equipos tome tiempo y requiera aprobación presupuestaria.

● Los equipos suelen intentar optimizar su rendimiento introduciendo lógica en la parte sobre la que tienen control

directo.



“Tradicionalmente” Se agrupa los equipos de acuerdo a la tecnología, suele haber equipos orientado a lógica de negocio, UI, a datos, etc.

El enfoque en microservicios es diferente: (es decididamente diferente, por que los equipos son multifuncionales, entonces para el armado de un microservicio, necesito que el equipo sea multifuncional)

● Los servicios son definidos en torno a funcionalidades de negocio.

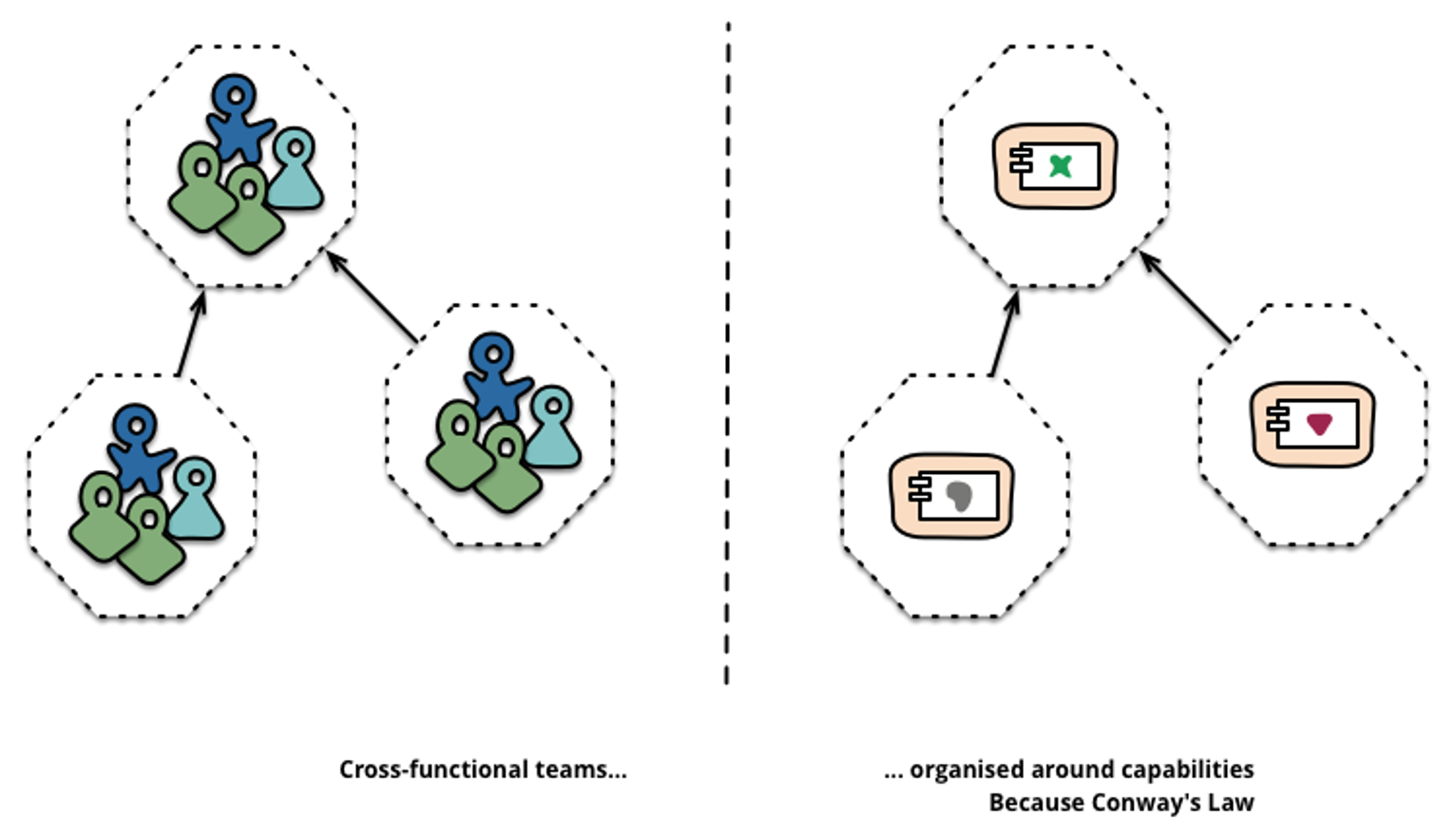
● Cada servicio requiere una implementación amplia de software

para el área de negocio a la que pertenece por lo que los equipos son **multifuncionales** y poseen la gama completa de habilidades requeridas.

● Grandes aplicaciones monolíticas también se pueden modularizar en torno a funciones de negocio.

○ Las líneas modulares requieren una gran disciplina para cumplirse.

○ En arquitecturas de microservicios la separación necesariamente más explícita entre servicios componentes facilita la delimitación de los equipos.



## Productos, no proyectos.

Proyecto es algo que tiene un objetivo determinado, un principio y fin, entonces teniendo este esquema: …

### Enfoque de proyecto:

● El objetivo es entregar una pieza de software que en algún momento se considerará terminada.

● Al finalizar, el software se asignará a una organización de mantenimiento y el equipo que lo creó se disuelve.

### Mentalidad de producto: (ídem a lo anterior pero no termina en producción, nosotros seguimos después)

● Un equipo debe poseer un producto durante toda su vida útil.

○ “*You build it, you run it*” (Amazon).

■ El equipo de desarrollo asume toda la responsabilidad del software en producción.

● Se establece una mejor relación entre el producto y la capacidad

de negocio a la que pertenece.

● Se concibe al software en una relación contínua en la que se busca como éste puede ayudar a sus usuarios a mejorar sus capacidades en el negocio.

## Smart endpoints and dumb pipes.

La aplicación misma vive en comunicación con servicios. Las partes que componen los microservicios hablan entre si, sino no funciona.

● La lógica de la aplicación se encuentra en los servicios y no en los mecanismos de comunicación.

● Los mensajes son coreografiados utilizando protocolos simples en lugar de protocolos complejos como WS-Choreography o BPEL u alguna herramienta de orquestación central.

● Los protocolos más utilizados son:

○ HTTP request / response con resource APIs.

○ Lightweight messaging.

## Gobierno descentralizado

## Min 57’

## La aplicación esta partida en microservicios, pero todo está partido en microservicios.

● Un gobierno centralizado tiende a estandarizar una única plataforma tecnológica.

● La segmentación de una aplicación en servicios permite elegir

para cada uno de ellos cuál es el stack tecnológico que mejor se adapta a sus necesidades.

● Se suelen implementar patrones de diseño específicos para

gestionar los contratos de los servicios buscando reducir su acoplamiento, limitando la necesidad de un gobierno central de los contratos.

● El concepto “*You build it, you run it*” es quizás la mayor

expresión de gobierno descentralizado.

## Gestión de datos descentralizada.

● Descentralización del modelo conceptual.

○ Bounded Context (DDD): Segmenta un dominio complejo en múltiples dominios delimitados y mapea las relaciones entre ellos.

Cada servicio tiene sus propios datos.

Persistencia poliglota: los datos los almacenamos según la forma que nos convenga.

Ejemplo: catalogo en no SQL, y para datos de facturación relacional.

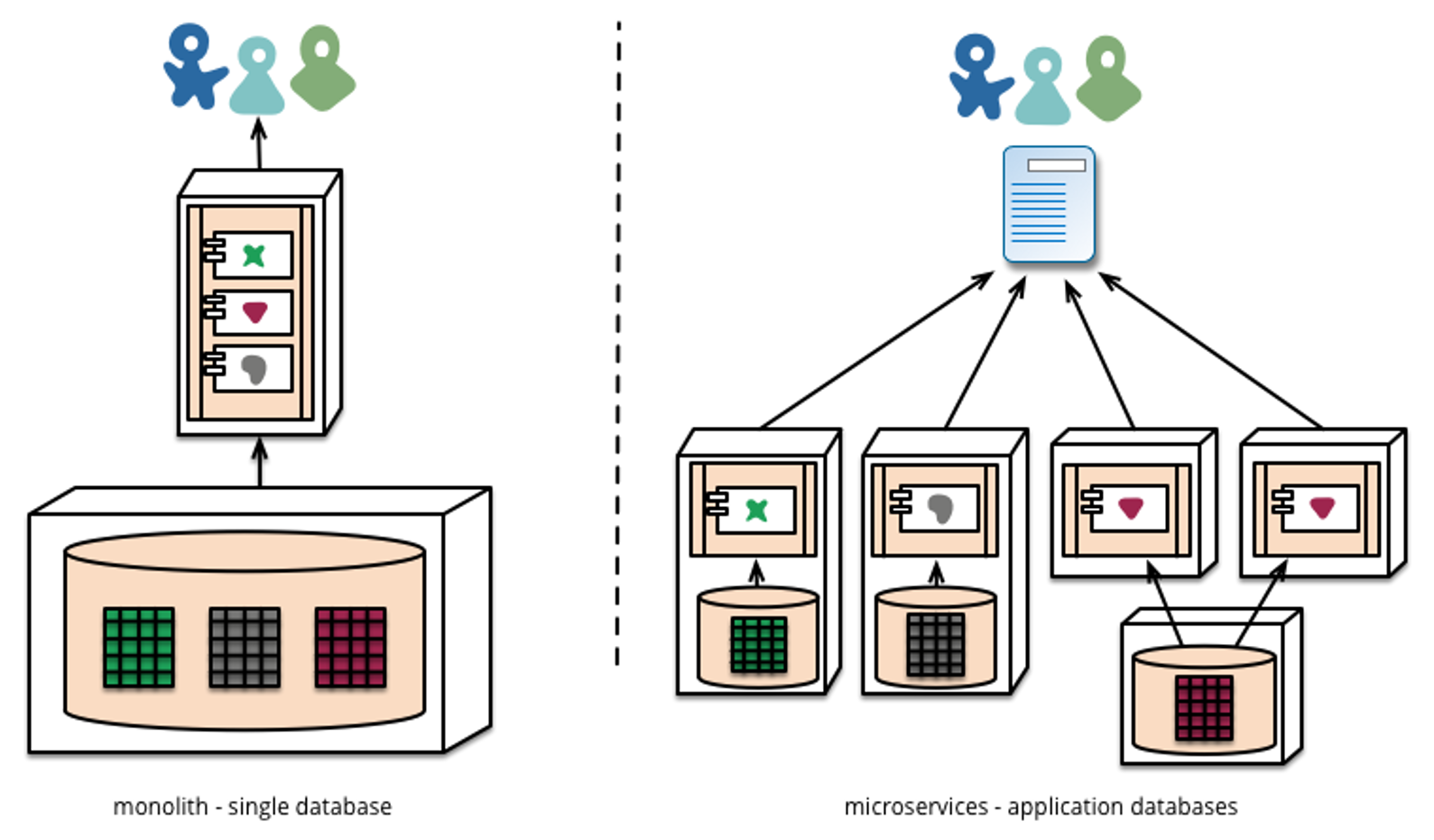
Lo que importa es que haya mas de un esquema de almacenamiento

● Descentralización de las decisiones de persistencia de datos.

○ Cada servicio administra su propia base de datos.

■ Diferentes instancias de una misma tecnología.

■ Persistencia políglota.



Teorema de CAP: es imposible para un **sistema distribuido de persistencia de datos** proveer simultáneamente:

El teorema de CAP se ocupa de los datos de los sistemas distribuidos, si la base de datos esta roto, y como es distribuido, entonces por ejemplo se puede pedir en distintos lugares para obtener los datos pedidos (mientras los datos replicados están accesibles en otro lugar, si es que tengo copia)

El Teorema de CAP no habla de cuestiones tecnológicas, simplemente con datos en sistemas distribuidos entonces analiza tres aspectos

Asegurar en todo momento es solo dos de estas tres cosas, el tercero a veces pero no siempre, entonces la estrategia es cual de los dos conviene asegurar siempre

● Consistencia: Cada lectura recibe la escritura más reciente o un error.

No importa de donde se busque los datos, importa que siempre de la misma respuesta

● Disponibilidad: Cada request recibe una respuesta no errónea, sin garantizar que contenga la escritura más reciente.

Lo importante es que se garantice una respuesta, es decir si una parte no anda, la otra parte este repondiendo, si alguien pide algo entonces que reciba la respuesta

● Tolerancia al particionamiento: el sistema sigue funcionando incluso si un número arbitrario de mensajes son descartados (o retrasados) entre nodos de la red.

Es que se pueda resolver el problema de por ejemplo dos datacenters que se dejan de comunicar la cual han evolucionado con inconsistencia entre ellas, entonces aunque han pedido la conexión de un datacenter con otro, hay tolerancia si podemos solucionar eso y volver a estar funcionando igual como normalmente

Manejo de actualizaciones.

● Enfoque tradicional de transacciones.

○ Alto nivel de consistencia.

○ Impone un fuerte acoplamiento temporal.

● Transacciones distribuidas.

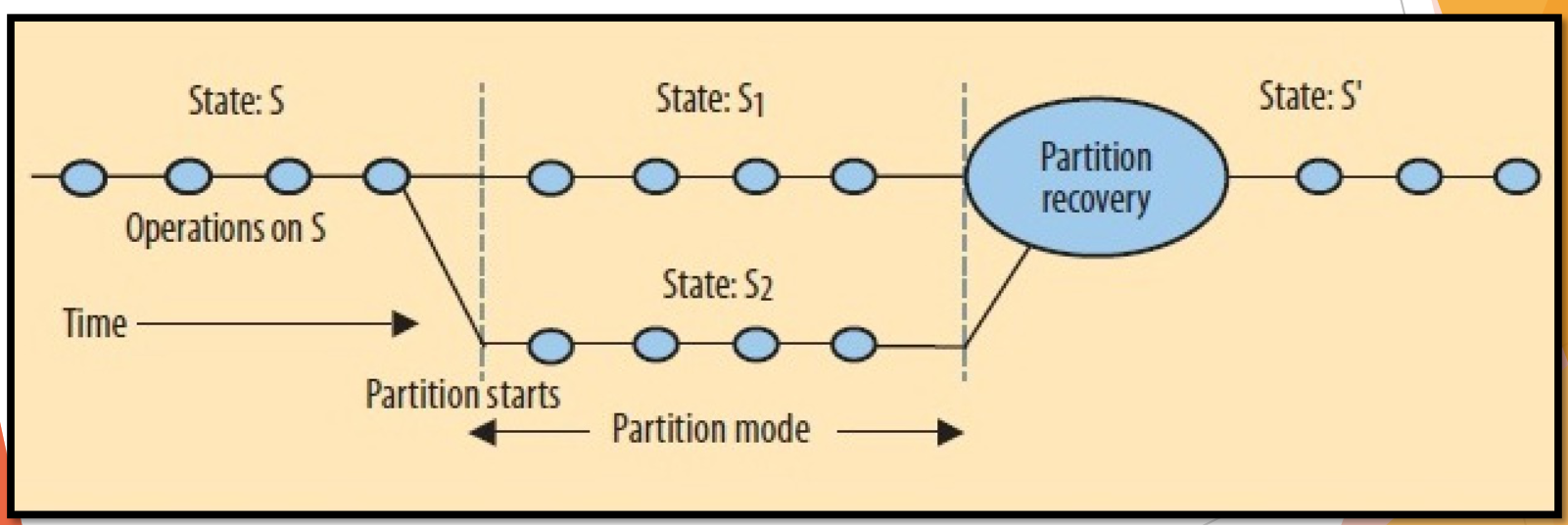
○ Alto costo de implementación.

● Coordinación de servicios sin transacciones.

○ Reconocimiento explícito de que la consistencia puede ser sólo **consistencia eventual.**

○ Los problemas se resuelven con operaciones de

compensación.



En la imagen anterior, se muestra como se cumple las tres, pero la partición de manera eventual. La consistencia y disponibilidad

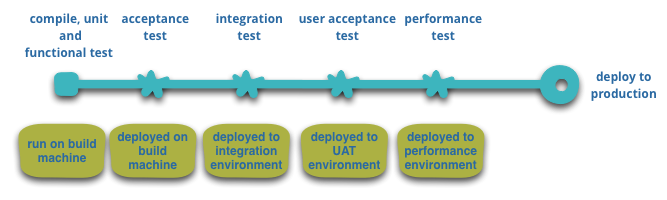
Los sistemas distribuidos, en lo que pueda fallar, tratemos

1. de que la tolerancia de partición tenga una posibilidad de ocurrencia mucho menor
2. en gestión de riesgos, trabajo en la mitigación para bajar la Probabilidad de impacto, pero también el trabajo en la contingencia, pero tener plan de contingencia implica prepararse que implica trabajar tener dinero

entonces en las apps distribuidas la app tiene que contemplar fallas en la infraestructura en la cual está montada, si este riesgo pasa, entonces debemos recuperarnos lo más antes posible, esto depende de nosotros y no del teorema de cap

## Automatización en Infraestructura

Son apps grandes pero que consiste en pequeños servicios, pero el despliegue de cada ambiente, es importante contar con mecanismo automatizados en cada despliegue, es decir despliegues automatizados



● Pipeline de entrega contínua (Continuous Delivery)

**Cuando continuamente se entregan cosas**

○ Integración contínua del software desarrollado, construyendo ejecutables y ejecutando pruebas automatizadas sobre éstos.

○ Despliegue de componentes en entornos cada vez más

similares a producción.

● Automatización en la gestión de microservicios en producción.

○ Orquestación de contenedores.

## Diseño tolerante a fallos

Esto se demostró con el teorema de CAP

● Las comunicaciones sobre redes son inherentemente poco confiables.

● Las aplicaciones se diseñan para ser resilientes y manejar errores, no sólo para prevenirlos.

○ Se aplican patrones de diseño específicos.

● Fuerte énfasis en el monitoreo en tiempo real de la aplicación.

○ Monitoreo de elementos de arquitectura.

○ Monitoreo de métricas relevantes del negocio.

## Diseño evolutivo

(es autoexplicativo)

● El diseño modular busca mantener las cosas que cambian al mismo tiempo dentro de un mismo módulo.

○ La propiedad clave de un componente es la noción de su independencia de cambio y actualización.

● Los servicios evolucionan buscando reducir al mínimo el

impacto de los cambios en sus consumidores.

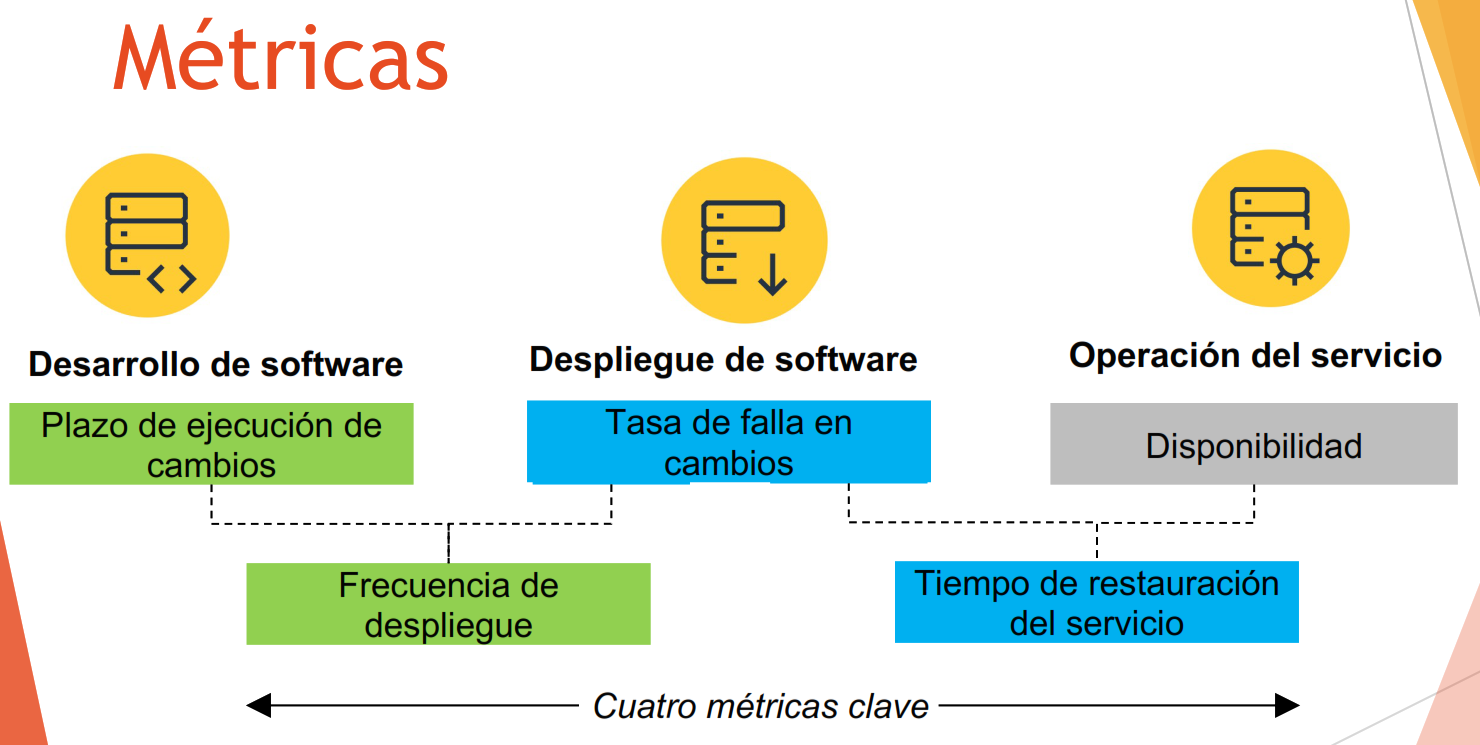
● Los servicios se diseñan buscando el mínimo acoplamiento

posible al contrato de sus proveedores.

● La utilización de servicios como componentes posibilita planeamientos de despliegues más granulares.

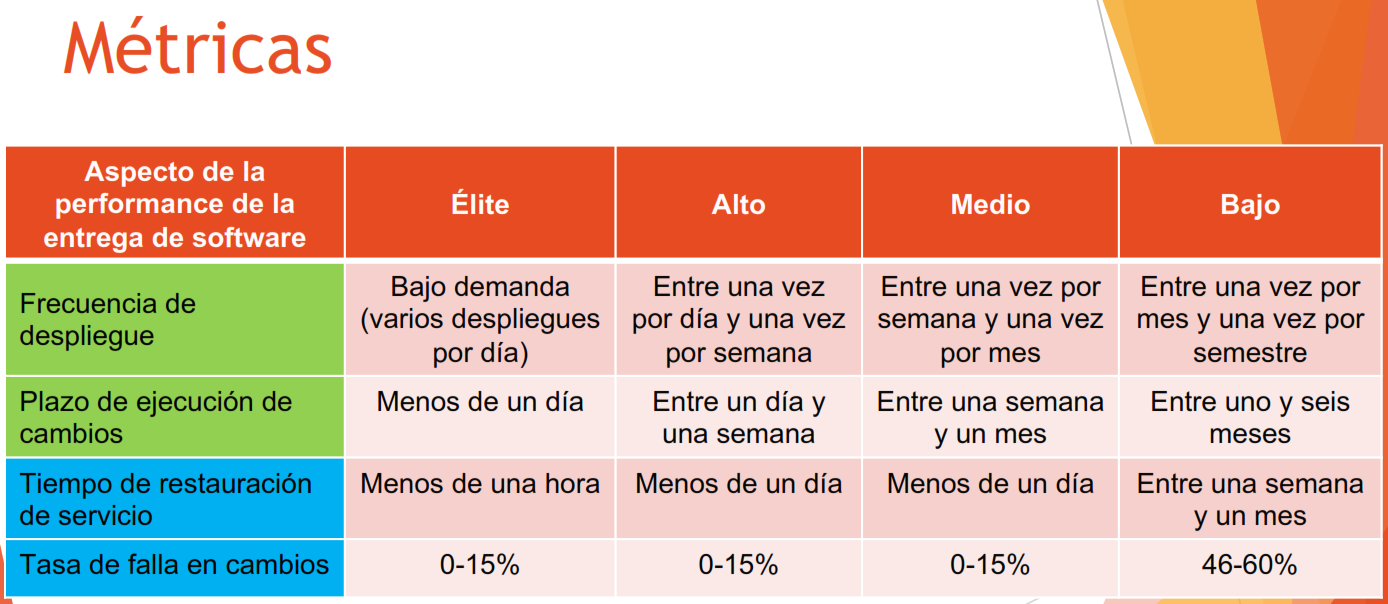
# METRICAS

Acá lo importante es el plazo de ejecución de los cambios



En disponibilidad es que siempre las aplicaciones estén en funcionamiento y sobre todo que siempre estén disponibles, que nunca paren

Y la frecuencia de despliegue, que cada cuanto se hacen



Ejemplo de frecuencia de despliegue: cada todos los días o varias veces al día, de acuerdo a la arquitectura

Bajo demanda: el despliegue de ninguna forma es un cuello de botella

Para desplegar mas rápido hay que producir más rapido.

# MSA - Ventajas

● Facilita el *continuous delivery* de aplicaciones grandes y complejas.

○ Mejora la mantenibilidad.

■ Descomposición modular de la complejidad.

■ Ciclos de evolución de componentes más desacoplados.

■ Límites y contratos de servicios más explícitos.

○ Facilita la incorporación de nuevas tecnologías.

○ Permite el despliegue y escalado independiente de servicios.

○ Permite organizar los esfuerzos de desarrollo en torno a

múltiples equipos autónomos.

● Los servicios son relativamente pequeños:

○ Son más fáciles de entender para un desarrollador.

○ La aplicación inicia en menor tiempo.

■ Aumento de productividad de desarrollo.

■ Mejora los tiempos de implementación.

● Mejora el aislamiento de fallos.

● Elimina compromisos a largo plazo con un stack tecnológico.

# MSA – Inconvenientes

**Como es enfoque distribuido, eso da complejidad**

● Aumento significativo de la complejidad.

○ Complejidad adicional propia de un sistema distribuido.

■ Requiere la implementación de mecanismos de comunicación entre servicios y el manejo de fallos parciales.

■ Se dificulta el testing de interacción entre servicios.

■ Aumenta la complejidad de implementación, gestión y monitoreo.

■ Se dificulta la detección de errores en tiempo de ejecución.

■ Las herramientas/IDEs están orientados al desarrollo de aplicaciones monolíticas.

● Complejidad de la arquitectura de persistencia de datos particionada.

○ Son muy comunes las transacciones de negocio que requieren actualizaciones en repositorios pertenecientes a múltiples servicios.

■ Las transacciones distribuidas no siempre son una opción.

■ El enfoque de persistencia eventual es más complejo que el enfoque tradicional transaccional.

● Las soluciones que abarcan múltiples servicios son más complejas

y requieren una ajustada coordinación entre equipos.

● Aumenta el consumo de memoria de la aplicación.

# ¿Cuándo utilizar una MSA?

Primero que bueno es en la industria

Segundo, interrogantes como: Es bueno para nosotros, nos va a servir, en que condiciones, tenemos esas condiciones

●Como toda arquitectura, tiene su compensación.

●**En diversas situaciones una arquitectura monolítica es la mejor opción**.

●Generalmente, las primeras versiones de una aplicación no tienen

los problemas que resuelve este enfoque.

●Si se requiere de una aplicación en el menor tiempo posible se debe considerar que el uso de una arquitectura elaborada y distribuida ralentizará el desarrollo.

●A medida que una aplicación va creciendo y surgen los desafíos de escalado y segmentación funcional, se complejiza la

descomposición de un monolito en un conjunto de servicios.

●Las MSA tienen serias consecuencias en la operación de la aplicación por lo que existen un conjunto de capacidades a considerar al evaluar su factibilidad de implementación:

■ Rápido aprovisionamiento.: es decir tenemos que desplegar nuestra solución en una infraestructura, pero eso normalmente depende de otros sectores, pero el problema es que se suele resolverse tarde

■ Monitoreo básico.: lo tenemos que poder monitorear

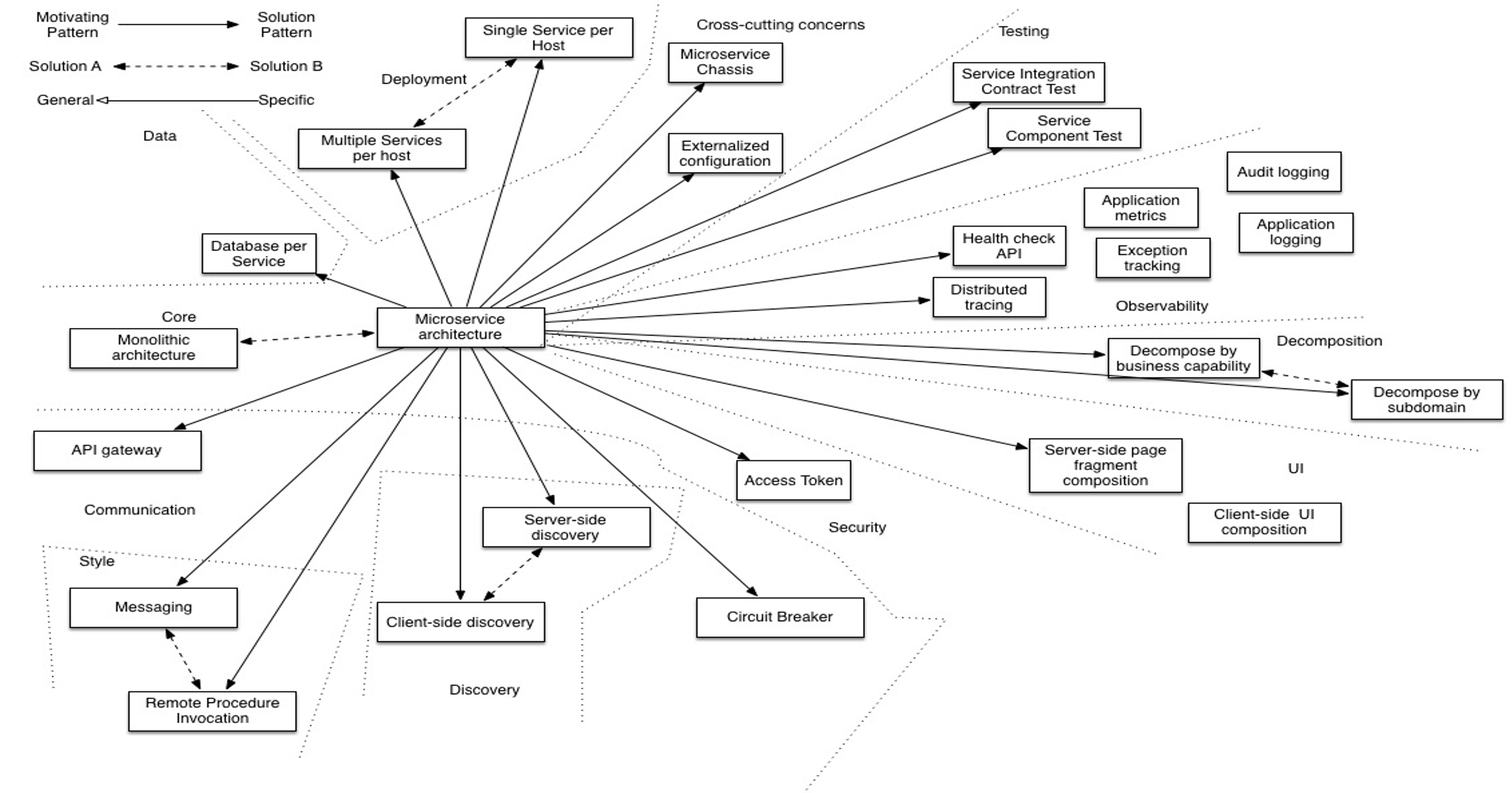
● Técnico (latencia, disponibilidad del servicio, etc.)

● De negocio (cantidades de pedidos)

■ Rápida implementación.: despliegue rápido de las aplicaciones, hay herramientas de automatización para esto, el equipo que se encarga del producto, el equipo es multifuncional, esto ultimo es muy importante, no es fácil, no es pedir y ya está. El equipo multifuncional debe ser resuelto antes.

● Generalmente, mediante un pipeline.

# Patrones Relacionados



En SOA la clave está en la interfaz, puesto que define los parámetros requeridos y la naturaleza del resultado. Esto significa que define la naturaleza del servicio y no la tecnología utilizada. Esta función permite realizar dos de los puntos críticos: los servicios son realmente independientes y pueden ser manejados.

# Referencias

● https://martinfowler.com/articles/microservices.html

● https://microservices.io/patterns/microservices.html

● https:/[/www.nginx.com/blog/introduction-to-microservices/](http://www.nginx.com/blog/introduction-to-microservices/)

● https://developers.redhat.com/promotions/microservices-for-java-developers/

● https://martinfowler.com/bliki/ContinuousDelivery.html

● https://martinfowler.com/articles/microservice-trade-offs.html

● https://martinfowler.com/bliki/MicroservicePrerequisites.html

● https:/[/www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/](http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed/)

● https://services.google.com/fh/files/misc/state-of-devops-2019.pdf