



GOBIERNO DE LA CIUDAD DE BUENOS AIRES

Ministerio de Educación

Dirección de Formación Técnico Superior

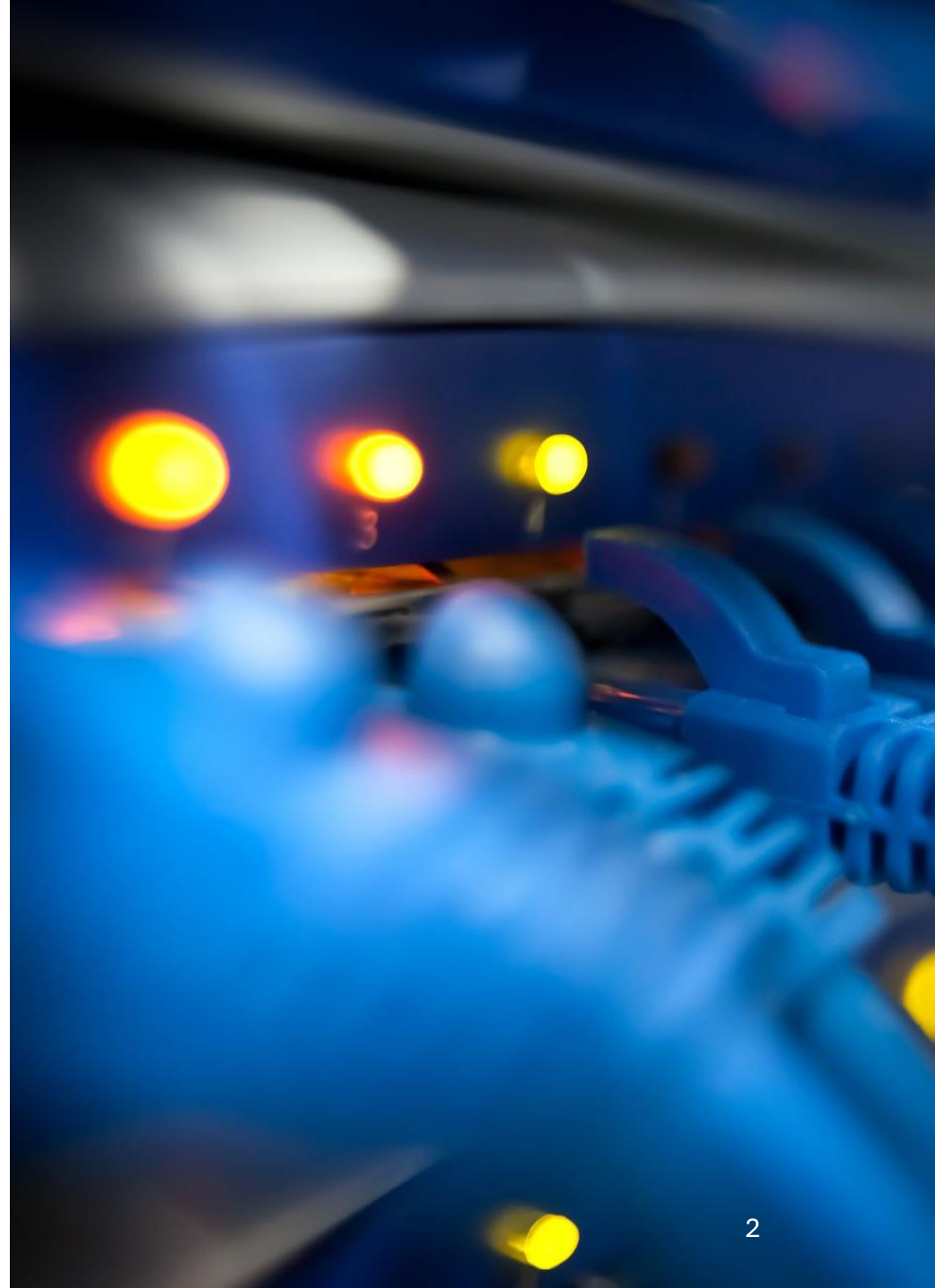
Instituto de Formación Técnico Superior N° 18

Mansilla 3643, (C1425BBW) Capital Federal

Arquitectura de sistemas en la Nube

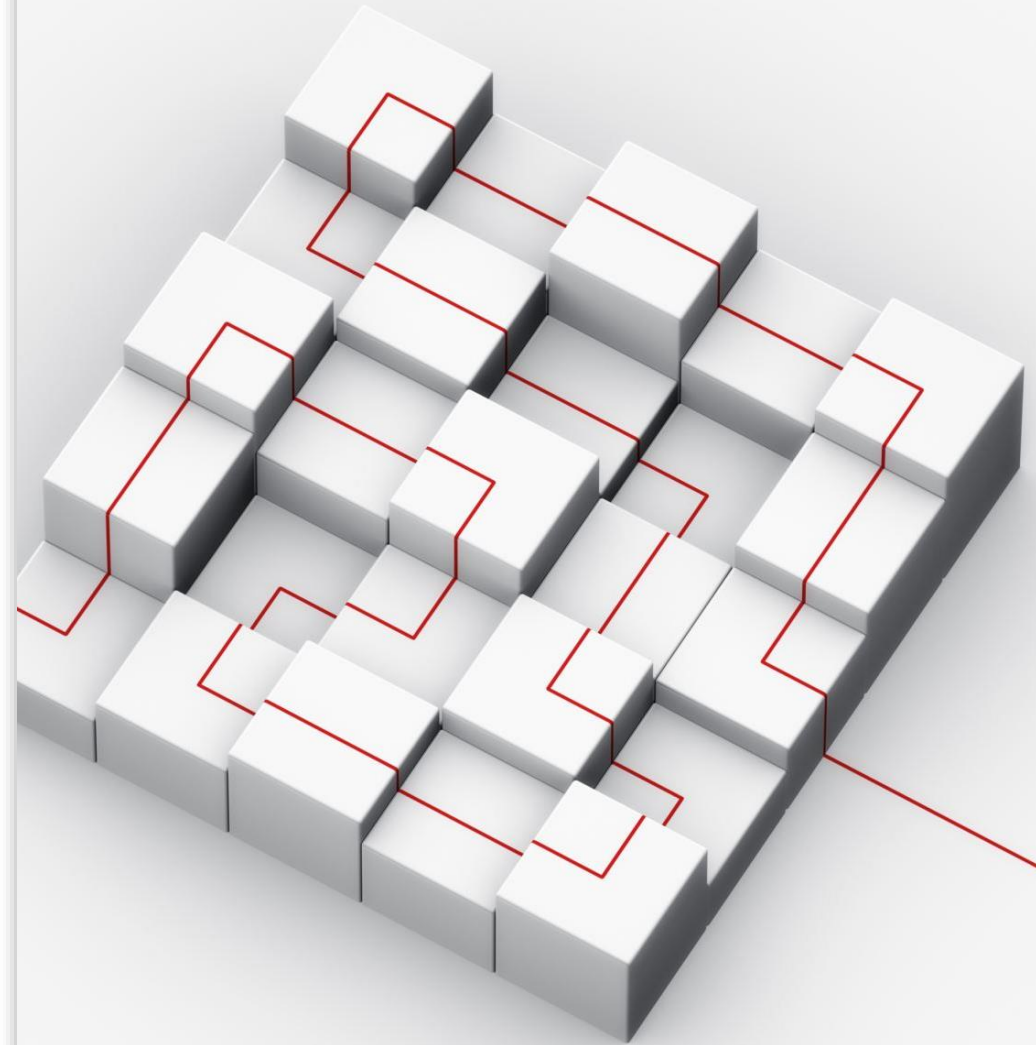
Diseño de Arquitecturas Escalables **TEMAS**

- Metodologías de Arquitectura: Well-Architected Framework
- Patrones de Diseño
- Disponibilidad y Escalabilidad
- Arquitectura de nube Front-end y Back-end



Well-Architected Framework

- El **Well-Architected Framework** es un conjunto de mejores prácticas diseñado originalmente por AWS (aunque Google Cloud y Azure tienen versiones muy similares) para ayudar a los arquitectos de nube a construir infraestructuras que sean lo más seguras, resilientes y eficientes posibles.



1. Excelencia Operativa

- Este pilar se centra en la capacidad de ejecutar y monitorear sistemas para entregar valor empresarial y mejorar continuamente los procesos y procedimientos.
- **Clave:** Automatizar cambios, responder a eventos y definir estándares para gestionar las operaciones diarias.
- **Enfoque:** "Realizar operaciones como código" (usando herramientas como Terraform o CloudFormation).

2.Seguridad

- Se enfoca en proteger la información, los sistemas y los activos, al tiempo que se ofrece valor empresarial mediante evaluaciones de riesgo y estrategias de mitigación.
- **Clave:** Implementar una base sólida de identidad (IAM), mantener la trazabilidad (logs), aplicar seguridad en todas las capas y proteger los datos en tránsito y en reposo.
- **Enfoque:** "Seguridad en todos los niveles", no solo en el perímetro.

3. Fiabilidad

- Asegura que un sistema sea capaz de recuperarse de interrupciones de infraestructura o de servicio, y que pueda adquirir recursos informáticos dinámicamente para satisfacer la demanda.
- **Clave:** Probar los procedimientos de recuperación (DRP), manejar fallos de manera automática y escalar horizontalmente para aumentar la disponibilidad.
- **Enfoque:** "Diseñar sistemas que se recuperen solos de los fallos".

4. Eficiencia del Rendimiento

- Trata sobre la capacidad de utilizar los recursos informáticos de manera eficiente para cumplir con los requisitos del sistema y mantener esa eficiencia a medida que la demanda cambia y las tecnologías evolucionan.
- **Clave:** Seleccionar los tipos de recursos y tamaños adecuados según la carga de trabajo, y tomar decisiones basadas en datos para optimizar el rendimiento.
- **Enfoque:** "Usar tecnologías avanzadas de forma democrática" (aprovechar servicios gestionados).

5. Optimización de Costos

- Se dedica a evitar gastos innecesarios. Se trata de entender dónde se gasta el dinero, seleccionar la cantidad adecuada de recursos y escalar en función de las necesidades del negocio sin pagar de más.
- **Clave:** Adoptar un modelo de consumo (pagar solo por lo que usas), medir la eficiencia y dejar de gastar dinero en operaciones de centros de datos físicos.
- **Enfoque:** "Gastar solo en lo que realmente genera valor".

6. Sostenibilidad

- se incorporó formalmente a finales de 2021 para responder a la creciente necesidad de reducir el impacto ambiental de las operaciones tecnológicas.
- En el contexto de la nube, la sostenibilidad no se trata solo de "ser verde", sino de maximizar la utilización de los recursos para reducir el desperdicio de energía y materiales. Es una extensión lógica de la optimización de costos y rendimiento, pero con el **medio ambiente** como métrica principal.

Patrones de Diseño: Evolución de la Complejidad



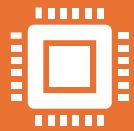
SOA

The diagram consists of three identical graphic elements arranged horizontally. Each element is a dark blue rounded rectangle with a light gray rounded rectangle inside it, offset to the top-left. The text 'SOA', 'Microservicios', and 'Serverless' are centered within the light gray rectangles.

Microservicios

Serverless

SOA



Una arquitectura orientada a servicios (SOA) es un paradigma de diseño que crea o complementa aplicaciones con bloques de construcción autónomos y reutilizables— servicios— que proporcionan funciones específicas y se pueden invocar a través de una red. Cada servicio tiene un ámbito definido, y otros componentes o servicios de la arquitectura pueden acceder a él para mejorar la funcionalidad general y el rendimiento del sistema.



SOA crea una infraestructura de software modular y flexible, donde los servicios individuales se pueden desarrollar, probar y desplegar de forma independiente. La facilidad de mantenimiento y la escalabilidad son dos ventajas clave

• Ventajas

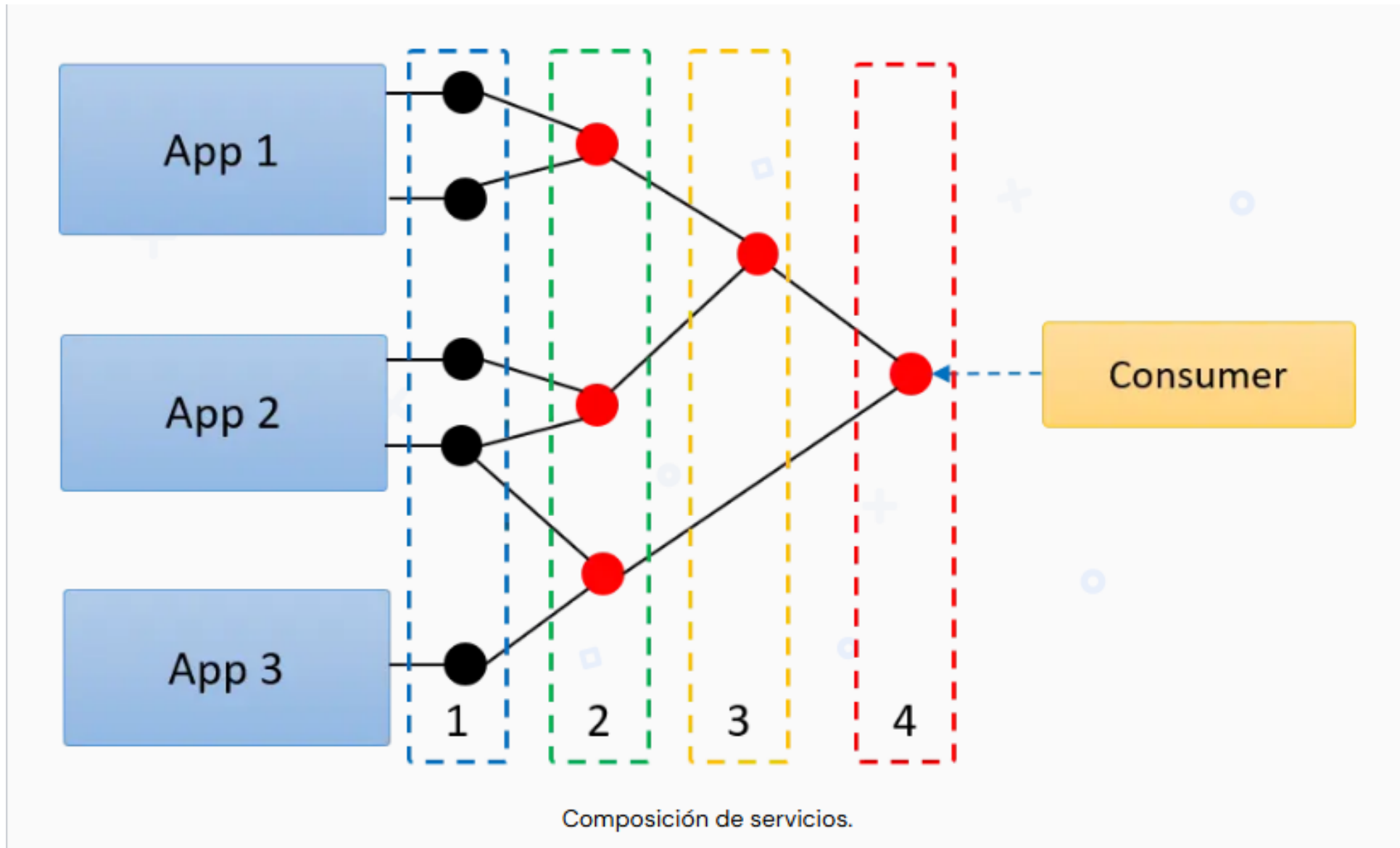
- **Reduce el acoplamiento:** La comunicación basada en contratos permite desacoplar las aplicaciones, ya que no existe referencias a la aplicación, si no a un servicio que puede estar en cualquier parte.
- **Testabilidad:** Las aplicaciones basadas en SOA son muy fáciles de probar, pues es posible probar de forma individual cada servicio.
- **Reutilización:** Los servicios SOA son desarrollados para hacer una sola tarea, por lo que facilita que otras aplicaciones reutilicen los servicios que existen.
- **Agilidad:** Permite crear rápidamente nuevos servicios a partir de otros existentes.
- **Escalabilidad:** Los servicios SOA son muy fáciles de escalar, sobre todo porque permite agregar varias instancias de un servicio y balancear la carga desde el ESB.
- **Interoperable:** La interoperabilidad es uno de los factores por lo que existe esta arquitectura y es que permite que aplicaciones heterogéneas se comuniquen de forma homogénea mediante el uso de estándares abiertos.

• Desventajas

- **Performance:** La arquitectura SOA no se caracteriza por el performance, pues la comunicación por la red y la distribución de los servicios agregar una latencia significativa en la comunicación.
- **Volumetría:** SOA no se lleva con grandes volúmenes por petición, por lo que para procesar mucha información es recomendable utilizar otras tecnologías.
- **Gobierno:** Para que SOA pueda ser implementado correctamente, es necesario crear un Gobierno que ponga orden sobre el uso y la creación de nuevos servicios, lo que puede representar una carga adicional a la empresa.
- **Más puntos de falla:** Debido a que SOA es una arquitectura distribuida, es necesario implementar estrategias de falla, pues se incrementa el número de puntos de falla.

Ejemplo

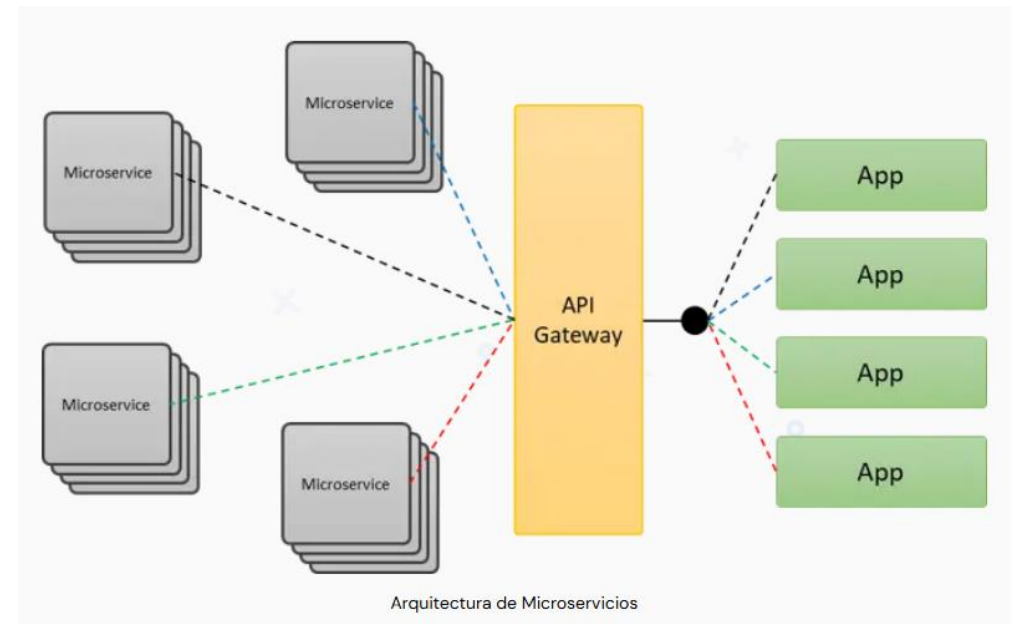
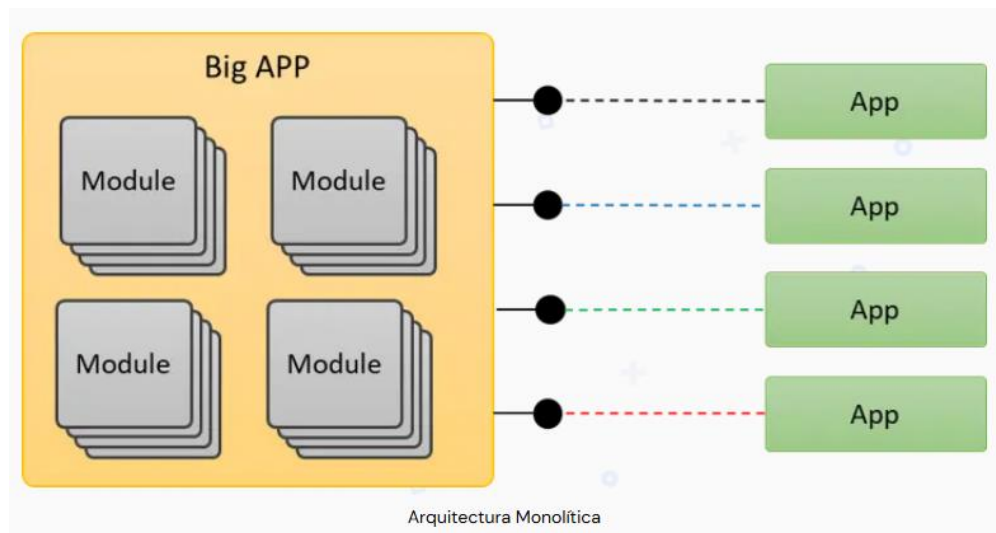
- Por ejemplo, una aplicación para runners sería una arquitectura orientada a servicios de consumo. En lugar de crear herramientas personalizadas para el seguimiento de distancias y mapas, la aplicación puede integrar servicios para el dispositivo GPS del teléfono y un servicio de mapas existente, como Google Maps. También puede mostrar el clima local del Servicio Meteorológico Nacional. El trabajo de la aplicación es presentar sin problemas datos de estos servicios en beneficio del runner.

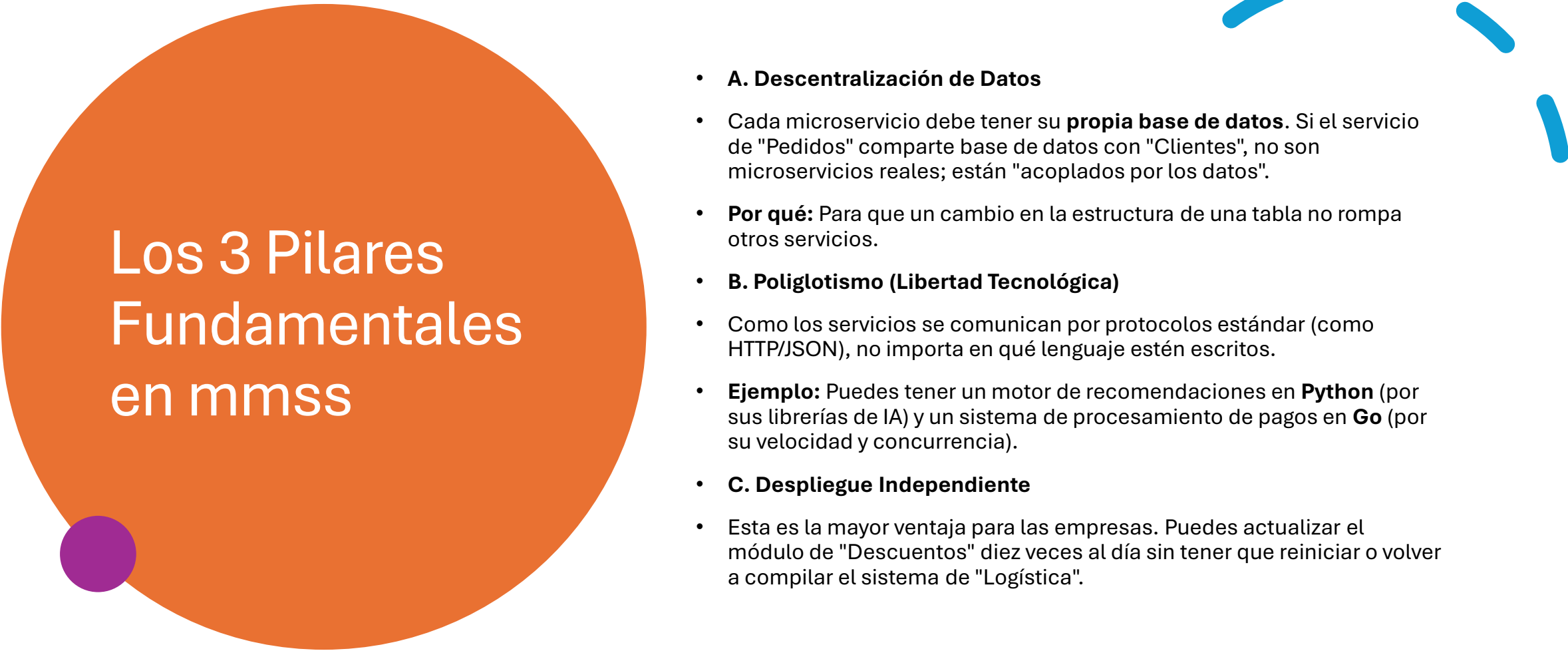




Microservicios

- Monolito vs. Microservicios
- En un **Monolito**, todas las funcionalidades (usuario, pagos, inventario) comparten el mismo código, el mismo proceso de servidor y la misma base de datos. Si una parte falla o necesita crecer, hay que afectar a toda la aplicación.
- En una **Arquitectura de Microservicios**, la aplicación se divide en pequeños servicios independientes que se comunican entre sí. Cada servicio tiene una **responsabilidad única** y se puede desarrollar, desplegar y escalar por separado.





Los 3 Pilares Fundamentales en mmss

- **A. Descentralización de Datos**
 - Cada microservicio debe tener su **propia base de datos**. Si el servicio de "Pedidos" comparte base de datos con "Clientes", no son microservicios reales; están "acoplados por los datos".
 - **Por qué:** Para que un cambio en la estructura de una tabla no rompa otros servicios.
- **B. Poliglotismo (Libertad Tecnológica)**
 - Como los servicios se comunican por protocolos estándar (como HTTP/JSON), no importa en qué lenguaje estén escritos.
 - **Ejemplo:** Puedes tener un motor de recomendaciones en **Python** (por sus librerías de IA) y un sistema de procesamiento de pagos en **Go** (por su velocidad y concurrencia).
- **C. Despliegue Independiente**
 - Esta es la mayor ventaja para las empresas. Puedes actualizar el módulo de "Descuentos" diez veces al día sin tener que reiniciar o volver a compilar el sistema de "Logística".



Ventajas

- **Alta escalabilidad:** Los Microservicios es un estilo arquitectónico diseñado para ser escalable, pues permite montar numerosas instancias del mismo componente y balancear la carga entre todas las instancias.
- **Agilidad:** Debido a que cada Microservicios es un proyecto independiente, permite que el componente tenga ciclo de desarrollo diferente del resto, lo que permite que se puedan hacer despliegues rápidos a producción sin afectar al resto de componentes.
- **Facilidad de despliegue:** Las aplicaciones desarrolladas como Microservicios encapsulan todo su entorno de ejecución, lo que les permite ser desplegadas sin necesidad de dependencias externas o requerimientos específicos de Hardware.
- **Testabilidad:** Los Microservicios son especialmente fáciles de probar, pues su funcionalidad es tan reducida que no requiere mucho esfuerzo, además, su naturaleza de exponer o brindar servicios hace que sea más fácil de crear casos específicos para probar esos servicios.
- **Fácil de desarrollar:** Debido a que los Microservicios tiene un alcance muy corto, es fácil para un programador comprender el alcance del componente, además, cada Microservicios puede ser desarrollado por una sola persona o un equipo de trabajo muy reducido.
- **Reusabilidad:** La reusabilidad es la médula espinal de la arquitectura de Microservicios, pues se basa en la creación de pequeños componentes que realice una única tarea, lo que hace que sea muy fácil de reutilizar por otras aplicaciones o Microservicios.
- **Interoperabilidad:** Debido a que los Microservicios utilizan estándares abiertos y ligeros para comunicarse, hace que cualquier aplicación o componente pueda comunicarse con ellos, sin importar en que tecnología está desarrollado.



Desventajas

- **Performance:** La naturaleza distribuida de los Microservicios agrega una latencia significativa que puede ser un impedimento para aplicaciones donde el performance es lo más importante, por otra parte, la comunicación por la red puede llegar a ser incluso más tardado que el proceso en sí.
- **Múltiples puntos de falla:** La arquitectura distribuida de los Microservicios hace que los puntos de falla de una aplicación se multipliquen, pues cada comunicación entre Microservicios tiene una posibilidad de fallar, lo cual hay que gestionar adecuadamente.
- **Trazabilidad:** La naturaleza distribuida de los Microservicios complica recuperar y realizar una traza completa de la ejecución de un proceso, pues cada Microservicio arroja de forma separa su traza o logs que luego deben de ser recopilados y unificados para tener una traza completa.
- **Madurez del equipo de desarrollo:** Una arquitectura de Microservicios debe ser implementada por un equipo maduro de desarrollo y con un tamaño adecuado, pues los Microservicios agregan muchos componentes que deben ser administrados, lo que puede ser muy complicado para equipo poco maduros.

Diferencia la SOA de los microservicio

- La SOA y los microservicios comparten un objetivo similar: proporcionar a las aplicaciones funcionalidad a través de llamadas de funciones basadas en red. Sin embargo, la SOA considera el uso en un nivel amplio, y generalmente superior. Los microservicios consideran el uso a nivel de aplicación y proporcionan servicio para que una única tarea extremadamente especializada se repita siempre que sea necesario. Por ejemplo, un plugin de autenticación para conexión segura se consideraría parte de SOA. Sin embargo, un servicio repetible que cifre la contraseña sería un microservicio. Los microservicios se pueden invocar dentro de un servicio SOA o se pueden integrar en un proyecto más grande. En este ejemplo, el servicio de autenticación utilizará el microservicio de cifrado para realizar su trabajo. El alcance es la principal diferencia entre los dos.

Serverless (Informática sin servidor)

- Es un modelo de **abstracción total de la infraestructura**.
- Es un modelo de ejecución en la nube donde el proveedor (AWS, Google, Azure) se encarga de gestionar la infraestructura de forma invisible.
- El desarrollador solo sube su código y el proveedor se ocupa de:
- **Aprovisionar** la máquina.
- **Parchear** el sistema operativo.
- **Escalar** si hay mucho tráfico.
- **Apagarlo todo** cuando nadie lo usa.

Los Tres Pilares Fundamentales

- **Abstracción de Servidores:** No eliges cuánta CPU o RAM tiene el "servidor", eliges cuánta memoria necesita tu **función**.
 - No hay direcciones IP ni nombres de host que gestionar.
- **Escalabilidad Horizontal Automática:** Si pasan de 1 usuario a 100,000 en un segundo, el proveedor levanta miles de pequeñas
 - instancias del código instantáneamente. No configuras reglas de *Auto-scaling*; es inherente al modelo.
- **Pago por Valor (FinOps):** Este es el punto más fuerte. En una VM pagas por hora, aunque esté al 1% de uso.
 - En Serverless, si el código no se ejecuta, el costo es **0**. Se paga por milisegundo de ejecución efectiva.

Disponibilidad

- Es la métrica que indica el porcentaje de tiempo que un sistema está operativo y funcional.
- Se mide comúnmente en "nueves".
- **Estrategias Clave:**
 - **Redundancia:** No tener puntos únicos de fallo (SPOF - Single Point of Failure). Si un componente falla, debe haber otro listo para tomar su lugar.
 - **Mecanismos de Failover:** El proceso automático de transferir la carga a un sistema redundante cuando el principal falla.
 - **SLA (Service Level Agreement):** Es el compromiso contractual. Por ejemplo, "cuatro nueves" (99.99%) permite solo 52.56 minutos de inactividad **al año**.

El cálculo de la disponibilidad:

- La disponibilidad de un sistema compuesto se calcula multiplicando la disponibilidad de sus componentes en serie,
- o mediante la fórmula de componentes en paralelo:
- $A = 1 - (1 - a)^n$
- Donde a es la disponibilidad de un componente y n la cantidad de componentes en paralelo.

Escalabilidad (Scalability)

- Es la capacidad del sistema para manejar un aumento en la carga (usuarios, transacciones, datos) añadiendo recursos.
- **Tipos de Escalado:**
- **Escalado Vertical (Scale Up):** Aumentar la potencia de un nodo existente (más CPU, más RAM).
 - *Limitación:* Tiene un techo físico y suele requerir reiniciar la máquina.
- **Escalado Horizontal (Scale Out):** Añadir más nodos al sistema (más servidores).
 - *Ventaja:* Es prácticamente infinito y favorece la alta disponibilidad. Requiere de un **Load Balancer** (Balanceador de Carga).

Elasticidad vs. Escalabilidad

- Es común confundirlos. La **escalabilidad** es la *capacidad* de crecer; la **elasticidad** es la *automatización* de ese crecimiento (y decrecimiento) en tiempo real según la demanda (Auto-scaling). Para FinOps, la elasticidad es vital porque evita el desperdicio de dinero en infraestructura ociosa.
- **Relación con el Teorema de CAP**
- En sistemas distribuidos, la disponibilidad entra en juego con la consistencia. Según el Teorema de CAP, en presencia de una partición de red, debes elegir entre:
- **Disponibilidad (A):** El sistema responde, pero los datos podrían no estar actualizados.
- **Consistencia (C):** El sistema no responde hasta que todos los datos estén sincronizados.

Analogía Pedagógica

- Si estuviéramos en un restaurante:
- **Disponibilidad:** Es tener siempre al menos dos cocineros. Si uno se enferma, el restaurante sigue abierto.
- **Escalabilidad Vertical:** Comprar una cocina más grande y rápida para el mismo cocinero.
- **Escalabilidad Horizontal:** Contratar más cocineros cuando el salón se llena de gente.

Balanceador de Carga

- Un **Balanceador de Carga** (Load Balancer) es un dispositivo o servicio que actúa como un "director de tráfico" frente a tus servidores. Su función principal es distribuir las solicitudes entrantes de los usuarios entre varios servidores para asegurar que ninguno se vea sobrepasado, optimizando así el rendimiento y la disponibilidad.

- En una arquitectura de servidor único, si el tráfico aumenta demasiado o el servidor falla, el servicio queda inaccesible (**Single Point of Failure**). El balanceador permite escalar horizontalmente, añadiendo más servidores al grupo (*Pool* o *Farm*).
- **Disponibilidad:** Si un servidor cae, el balanceador redirige el tráfico a los que siguen activos.
- **Escalabilidad:** Permite añadir o quitar nodos según la demanda sin interrumpir el servicio.
- **Eficiencia:** Maximiza el uso de los recursos de hardware existentes.

Tipos de Balanceo según el Modelo OSI

- Dependiendo de qué capa del modelo OSI analicen para tomar decisiones, se dividen principalmente en dos:
- **Capa 4 (Transporte)**
- **Protocolos:** TCP/UDP.
- **Funcionamiento:** Solo mira la IP de origen/destino y los puertos. No inspecciona el contenido del paquete.
- **Ventaja:** Es extremadamente rápido y consume pocos recursos de procesamiento.
- **Capa 7 (Aplicación)**
- **Protocolos:** HTTP/HTTPS.
- **Funcionamiento:** Inspecciona el contenido del tráfico (URLs, Cookies, Headers). Puede enviar las peticiones de /imagenes a un servidor y las de /api a otro.
- **Ventaja:** Permite decisiones mucho más inteligentes y personalizadas.

Algoritmos de Distribución

Algoritmo	Descripción
Round Robin	Las peticiones se envían secuencialmente (1, 2, 3, 1...). Ideal si todos los servidores tienen la misma capacidad.
Least Connections	Envía el tráfico al servidor con menos conexiones activas en ese momento.
IP Hash	Utiliza la dirección IP del cliente para determinar a qué servidor enviarlo, asegurando que un usuario específico siempre vaya al mismo nodo.
Weighted (Pesado)	Se asigna un "peso" a cada servidor. Los más potentes reciben un porcentaje mayor de tráfico.

Inteligencia aplicada en un balanceador

- **Health Checks (Pruebas de Salud):** El balanceador monitorea constantemente a los servidores.
 - Si uno no responde a un "ping" o a una petición de prueba, lo marca como "fuera de servicio"
 - y deja de enviarle tráfico automáticamente.
- **Persistencia de Sesión (Sticky Sessions):** En aplicaciones donde el servidor guarda datos locales del usuario (como un carrito de compras no persistido en DB), el balanceador debe asegurar que ese usuario específico siempre vuelva al mismo servidor durante su sesión.

Auto-scaling

- El **Auto-scaling** es uno de los pilares de la computación en la nube. Es lo que permite pasar de una infraestructura rígida a una verdaderamente elástica. En términos simples, es la capacidad de ajustar automáticamente la cantidad de recursos computacionales en función de la demanda en tiempo real.

Escalamiento Horizontal vs. Vertical

- **Vertical Scaling (Scaling Up/Down):** Consiste en añadir más potencia (CPU, RAM) a una instancia existente.
- *Limitación:* Tiene un techo físico y usualmente requiere reiniciar la máquina.
- **Horizontal Scaling (Scaling Out/In):** Consiste en añadir más instancias (nodos) al grupo.
- *Ventaja:* Es prácticamente infinito y permite alta disponibilidad. El auto-scaling moderno se enfoca casi exclusivamente en este modelo.

Los Componentes del Auto-scaling

- **Configuración de Lanzamiento (Launch Template):** El "molde" de la instancia. ¿Qué sistema operativo tiene? ¿Qué scripts de inicio debe ejecutar? ¿Qué tamaño de instancia es?
- **Grupo de Auto-scaling (ASG):** Define los límites lógicos.
 - *Tamaño Mínimo:* Cuántas instancias deben estar vivas siempre (para asegurar disponibilidad).
 - *Capacidad Deseada:* El estado ideal en un momento dado.
 - *Tamaño Máximo:* El tope para evitar que una falla o un ataque disparen la factura (un control vital para FinOps).
- **Políticas de Escalamiento:** Las reglas que deciden cuándo actuar.

Tipos de Políticas (Estrategias)

- **Escalamiento Reactivo (Dynamic Scaling)**
 - Se basa en métricas en tiempo real. Si el uso de CPU supera el 70% durante 5 minutos, añade 2 instancias.
 - **Métricas comunes:** Utilización de CPU, memoria, tráfico de red o, en aplicaciones web, el número de solicitudes por segundo.
- **Escalamiento Programado (Scheduled Scaling)**
 - Ideal cuando conocemos de antemano el comportamiento de la carga.
 - *Ejemplo:* Un sitio de e-commerce que sabe que los "Cyber Monday" el tráfico sube a las 00:00 hs. No esperamos a que la CPU suba; levantamos los servidores 15 minutos antes.
- **Escalamiento Predictivo (Predictive Scaling)**
 - Utiliza Machine Learning para analizar patrones históricos y anticiparse a la demanda. Si todos los jueves a las 18:00 hs hay un pico, el sistema empieza a provisionar recursos automáticamente antes de que ocurra.

Conceptos Avanzados: Cooldown y Flapping

- Un error común al diseñar auto-scaling es no configurar el **Cooldown Period** (Periodo de enfriamiento).
- **El problema del Flapping:** Si una métrica sube y baja constantemente cerca del umbral, el sistema podría entrar en un bucle de "prender y apagar" servidores. Esto es ineficiente y costoso.
- El **Cooldown** le dice al sistema: *"Una vez que hayas escalado, espera X segundos antes de volver a tomar otra decisión de escalado"*. Esto permite que las nuevas instancias terminen de arrancar y empiecen a absorber carga antes de que el monitor vuelva a evaluar la situación.

El Vínculo con el Balanceador de Carga (ELB)

- El Auto-scaling por sí solo crea instancias, pero alguien tiene que decirles a los usuarios a dónde ir. Aquí entra el **Load Balancer**.
- El Auto-scaling crea la instancia.
- La instancia se registra automáticamente en el Balanceador.
- El Balanceador empieza a enviarle tráfico una vez que pasa los controles de salud (*Health Checks*).
- Para un enfoque de **FinOps**, el auto-scaling es la herramienta principal para eliminar el desperdicio: solo pagamos por lo que usamos en el momento exacto en que lo necesitamos.

DRP (Disaster Recovery Plan)

- **DRP vs. BCP**
- **BCP (Business Continuity Plan):** Es el plan global. Responde a: "*¿Cómo seguimos operando si la oficina se inunda?*" (Incluye RRHH, procesos legales, comunicación, etc.).
- **DRP (Disaster Recovery Plan):** Es una **subsección técnica** del BCP. Se enfoca exclusivamente en la recuperación de la infraestructura TI y los datos. Es el "botón de reinicio" técnico.

Indicadores críticos: RTO y RPO

Concepto	Definición	Pregunta clave
RPO (Recovery Point Objective)	La cantidad máxima de datos que la empresa puede permitirse perder (medido en tiempo).	¿Desde hace cuánto fue nuestro último backup?
RTO (Recovery Time Objective)	El tiempo máximo que puede pasar la empresa con los sistemas caídos.	¿Cuánto tardamos en volver a estar online?

En una implementación de DRP



Análisis de Impacto al Negocio (BIA)

- Antes de configurar servidores, identifica qué sistemas son críticos. No es lo mismo que se caiga el software de facturación a que se caiga el sistema de reserva de la cafetería.
- **Identificación de activos.**
- **Cálculo de pérdidas por hora de inactividad.**

Estrategias de Recuperación

- **Cold Site:** Un espacio físico con energía pero sin equipo. Barato, pero muy lento de activar.
- **Warm Site:** Equipos listos, pero los datos deben ser cargados. Equilibrio costo/beneficio.
- **Hot Site:** Una réplica exacta en tiempo real. Espejo total. Caro, pero recuperación casi instantánea.
- **DRaaS (Disaster Recovery as a Service):** La tendencia actual. Usar la nube (AWS, Azure, Google) para replicar servidores virtuales de forma elástica.

Las Pruebas

- Un DRP que no se prueba es solo un documento decorativo en un PDF. La profundidad real de un plan se ve en su ejecución controlada:
- **Checklist Walkthrough:** Revisar el plan en una mesa con los responsables.
- **Simulación:** Escenario hipotético de crisis donde se verifican comunicaciones.
- **Prueba Paralela:** Levantar los sistemas en el sitio de recuperación sin apagar el principal.
- **Full Interruption:** Apagar el sistema principal y ver si el DRP realmente funciona. (Solo para valientes y sistemas bien maduros).

El factor humano

- **Cadena de mando clara:** Quién tiene la autoridad de declarar "Desastre" (no es una decisión ligera).
- **Plan de comunicación:** Cómo hablar con clientes, empleados y proveedores si el correo interno no funciona.
- **Dato curioso:** La mayoría de los DRP fallan no por la tecnología, sino porque la persona que tenía la contraseña maestra no estaba localizable o el manual de recuperación estaba guardado... en el servidor que se cayó.



Arquitectura de nube Front-end y Back-end



El Front-end: La Capa de Presentación y Cliente

- En la nube, el front-end no es solo "HTML/CSS", sino cualquier **dispositivo o interfaz** que consume servicios.
- **Tipos de Clientes:**
 - **Web (SPAs):** Aplicaciones de una sola página (React, Angular) que se descargan una vez y luego solo intercambian datos vía JSON.
 - **Mobile Nativo:** Apps en iOS/Android que requieren una integración eficiente para no agotar batería ni datos.
 - **IoT (Internet of Things):** Dispositivos con recursos limitados que envían ráfagas de telemetría.
- **Distribución (Edge Computing):** El front-end moderno se sirve a través de **CDNs (Content Delivery Networks)**. Los activos estáticos se almacenan en servidores "borde" para reducir la latencia hacia el usuario final.

El Back-end: Procesamiento y Lógica de Negocio

- Aquí es donde reside la inteligencia. En la arquitectura de nube, el procesamiento se desacopla del hardware físico mediante:
- **Modelos de Procesamiento**
- **Serverless (FaaS):** Ideal para tareas dirigidas por eventos (ej. AWS Lambda). El back-end solo "vive" durante la ejecución de la función.
- **Contenedores (CaaS):** Uso de Docker y Kubernetes para empaquetar microservicios que escalan de forma independiente.
- **Instancias Virtuales (IaaS):** Control total sobre el sistema operativo, usado para aplicaciones legadas o que requieren configuraciones de red muy específicas.

Integración

- La comunicación entre el cliente y la nube se realiza mayoritariamente a través de **APIs**
- **Capas de Intermediación**
- **API Gateway:** Actúa como el punto de entrada único. Gestiona la autenticación, el filtrado de tráfico, el límite de velocidad (Rate Limiting) y el enrutamiento a diferentes microservicios.
- **Load Balancers:** Distribuyen el tráfico entrante entre múltiples instancias de procesamiento para asegurar la alta disponibilidad.
- **Protocolos:**
 - **REST:** El estándar para operaciones CRUD sobre HTTP.
 - **GraphQL:** Permite al front-end pedir exactamente los datos que necesita, reduciendo el ancho de banda.
 - **WebSockets:** Para comunicación bidireccional en tiempo real (ej. chats o dashboards financieros).

Consideraciones de Arquitectura

- Para que esta integración sea exitosa, se deben observar tres pilares:
- **Seguridad:** Implementación de protocolos como **OAuth2/OpenID Connect** para que el front-end maneje tokens de identidad en lugar de credenciales.
- **Escalabilidad:** El back-end debe ser capaz de crecer (Auto-scaling) automáticamente cuando el número de dispositivos clientes aumenta.
- **Observabilidad:** Uso de logs y trazas distribuidas para entender qué sucede desde que un usuario hace clic en el front-end hasta que el dato se guarda en la base de datos.

- <https://www.youtube.com/watch?v=06y3zORr5jA>