1. Arquitectura para la innovación

Necesitamos diseñar nuestra arquitectura de software para el cambio. Debemos innovar a la velocidad del negocio. Necesitamos adaptar y cambiar nuestro software en tiempo real. Necesitamos experimentar para encontrar las mejores soluciones en un entorno dinámico. Y necesitamos la confianza para movernos rápido e innovar, sin romper lo que ya funciona.

Entrega continua de valor de negocio

La entrega continua de valor de negocio debe ser nuestro foco principal. Sin embargo, necesitamos crear prácticas sostenibles y una arquitectura que las respalde.

Debemos innovar para mantenernos a la vanguardia y generar valor. En otras palabras, debemos experimentar con distintos cambios en nuestro software. Nuestra arquitectura debe habilitar el cambio.

Análisis del lead time

Nuestra capacidad de entregar valor de negocio de forma continua es función del lead time. Si medimos nuestros lead times en meses o incluso semanas, nuestra capacidad de entrega no es continua.

Para convertirnos en un equipo de alta velocidad, necesitamos desarrollar la memoria muscular que nos permita producir este flujo aparentemente ininterrumpido de ideas y experimentos.

Examinemos las fuerzas que influyen en nuestro lead time.

Mitigación de riesgos

El riesgo es un poderoso motivador. Tradicionalmente, lo mitigábamos desacelerando el ritmo e incrementando el lead time.

Necesitamos enmarcar nuestra idea original como una hipótesis que requiere validación. Luego, ponemos el software en manos de usuarios reales lo antes posible para determinar si vamos en la dirección correcta y corregir el rumbo antes de que sea demasiado tarde.

Sin embargo, ir más rápido introduce el riesgo de romper lo que ya funciona. Los usuarios dependen de las funcionalidades existentes y esperan que sigan funcionando. Siempre quieren más, pero no a costa de perder lo que ya tienen. **Por lo tanto, necesitamos movernos rápido sin romper nada.**

Nuestra arquitectura debe mitigar el riesgo de romper cosas limitando el radio de impacto de cualquier error a únicamente aquello que está cambiando.

Toma de decisiones

Tradicionalmente, las empresas gestionan los productos de software como proyectos. Para avanzar, un proyecto debe contar con un plan, un presupuesto y una aprobación. **El proceso para lograrlo consume mucho tiempo y es costoso, lo que genera un incentivo para crear menos proyectos, pero más grandes y más caros.**

Las empresas están aprendiendo que es más eficaz definir los resultados deseados y luego dar a los equipos la autonomía para tomar decisiones más pequeñas y focalizadas, y trabajar en función de esos resultados.

Metodología del ciclo de vida del desarrollo de software

La metodología del ciclo de vida del desarrollo de software (SDLC) que sigue un equipo es el factor más evidente que impacta el lead time.

Waterfall es la metodología clásica y la reina de los lead times largos, con lanzamientos de software medidos en meses o años.

Los métodos **Agile** representan una mejora significativa, con lead times medidos en semanas. Sin embargo, un sprint de Agile de dos semanas, por ejemplo, únicamente produce software listo para producción.

Los métodos **Kanban** y **Lean** son los que mejor ayudan a controlar el tamaño del lote, de modo que podemos poner software funcional en manos de usuarios beta reales con lead times medidos en horas o días.

Aprovisionamiento de hardware

El aprovisionamiento de hardware es la piedra angular del lead time. Históricamente, adquiríamos estos activos físicos (es decir, computadoras) como gastos de capital. Esto requería procesos de contabilidad, presupuestación y aprobación con lead times prolongados.

El método Waterfall era muy adecuado para estas condiciones, y los procesos Agile funcionaban bien si aprovisionábamos el hardware de forma proactiva.

Luego, la nube lo cambió todo. Podíamos aprovisionar hardware bajo demanda. El hardware pasó a ser un gasto operativo y ya no requería largos procesos de aprobación.

Despliegues

Antes de la nube, había pocas razones para automatizar el aprovisionamiento de hardware y los despliegues de software.

Pero la nube convirtió los despliegues manuales en el nuevo cuello de botella, dado que el hardware ya estaba disponible bajo demanda. Teníamos un incentivo para optimizar este paso en el pipeline de entrega. Se volvió imperativo automatizar el despliegue de toda la pila: no solo el software, sino también la infraestructura. **Comenzamos a tratar toda la pila como infraestructura como código (IaC).**

Los lead times de despliegue cayeron en picada. La calidad y la predictibilidad de los despliegues aumentaron de forma drástica.

Estructura del software

La forma en que desplegamos el software tiene un impacto natural en cómo lo estructuramos y arquitecturamos.

A medida que comenzamos a automatizar los despliegues de software, quedó claro que combinar funcionalidades en monolitos era el siguiente cuello de botella en el pipeline de entrega de productos de software.

El propio software entorpecía nuestra capacidad de modificarlo y reducir los lead times. Podíamos iterar sobre la funcionalidad y automatizar los despliegues, pero temíamos ejecutar ese despliegue por miedo a las consecuencias no deseadas de redesplegar todo el monolito por un pequeño cambio en una sola parte de él.

Necesitábamos descomponer el monolito. Los despliegues automatizados hicieron práctico el despliegue de muchos microservicios. **Podíamos realizar pequeños cambios en servicios individuales y redesplegarlos de forma independiente.**

Pruebas y confianza

Necesitamos probar de forma continua para mitigar el riesgo de romper el software que ya funciona. Esto da a los equipos la confianza para ir más rápido porque, sin confianza, los equipos naturalmente reducirán la velocidad e incrementarán el lead time.

Aumentaremos la capacidad de prueba y la observabilidad. Automatizaremos las pruebas continuas y las desplazaremos hacia la izquierda en el pipeline de entrega para garantizar que no haya regresiones en el software que funciona.

Aprovecharemos los **feature flags** para controlar el acceso al nuevo software. Luego, pondremos el nuevo software en manos de usuarios beta para que puedan validar nuestra hipótesis.

Dependencias y comunicación entre equipos

Las dependencias y la comunicación entre equipos tienen un impacto significativo en el lead time. Cuando un equipo depende de otro, no podemos comprimir los lead times más allá del tiempo que tardan los otros equipos en responder a nuestras solicitudes.

Necesitamos destinar tiempo y esfuerzo adicionales para coordinar y alinear los calendarios y prioridades de los equipos.

La Ley de Conway establece que las organizaciones están condicionadas a producir diseños que son copias de su estructura de comunicación. En otras palabras, para reducir los lead times, la estructura del equipo y la estructura de nuestra arquitectura de software deben funcionar en conjunto. Los equipos no pueden estar a merced de otros equipos.

Deben ser propietarios de toda la pila. Deben ser multifuncionales. Los equipos deben ser autónomos, y nuestra arquitectura de software debe habilitarlos.

Para maximizar la autonomía de los equipos y tomar control de los lead times, necesitamos un enfoque para crear funcionalidades autónomas. Necesitamos un mecanismo de comunicación entre funcionalidades que nos ayude a mitigar riesgos y dé a los equipos la confianza para experimentar.

Análisis de los estilos de integración

Nuestros sistemas de software están compuestos por muchas funcionalidades que deben trabajar en conjunto para generar valor para los usuarios finales. La forma en que estas funcionalidades se comunican ha sido uno de los temas más relevantes de nuestra industria. **Los canales de comunicación dentro de un sistema impactan el flujo oportuno de información, pero también nuestra capacidad de entregar soluciones a tiempo.**

Un sistema altamente integrado maximizará el flujo de información. Sin embargo, si integramos las funcionalidades de forma demasiado estrecha, se obstaculiza nuestra capacidad de adaptarnos a los requisitos cambiantes.

Integración por lotes

Las empresas tenían múltiples sistemas en sus organizaciones, y no estaban bien integrados. La valiosa información de negocio quedaba aislada en estos silos (es decir, sistemas).

Las integraciones que existían eran:

- orientadas a lotes. En el mejor de los casos, estos procesos se ejecutaban de forma nocturna.
- Esto significaba que los datos solían ser obsoletos e incluso estaban en conflicto.
- La reconciliación de datos entre los silos era frecuentemente una pesadilla.

Pero los silos eran autónomos.

Integración espagueti

Para empeorar las cosas, las integraciones por lotes eran punto a punto. Cada proceso consideraba únicamente un sistema de origen y un sistema de destino. Esto derivó en lo que comúnmente denominamos integración espagueti.

Integración en tiempo real

La industria buscaba un nuevo enfoque de integración, y el modelo cliente-servidor parecía la solución. Comenzamos a escribir integraciones en tiempo real entre las nuevas aplicaciones con interfaz gráfica de usuario (GUI) y los sistemas heredados.

Habíamos acoplado todo fuertemente. La disponibilidad de un sistema ahora dependía de la disponibilidad de los sistemas con los que se integraba.

El resultado neto fue un aumento del lead time. El nuevo estilo de integración en tiempo real seguía siendo integración espagueti punto a punto. Necesitábamos más comunicación y coordinación entre equipos para mantener funcionando esas integraciones frágiles.

Integración de aplicaciones empresariales

La desregulación de los años noventa derivó en numerosas adquisiciones y fusiones. Las empresas se encontraron con sistemas duplicados. **Esto llevó al surgimiento de la industria y el paradigma de la integración de aplicaciones empresariales (EAI).**

Los sistemas duplicados debían integrarse de manera que no fuera necesario modificarlos. Cada uno seguiría siendo el maestro de sus propios datos, por lo que necesitábamos soportar múltiples maestros heterogéneos. **Implementamos sincronización bidireccional entre todos los sistemas.**

El enfoque resultante fue una sincronización basada en eventos, en tiempo casi real y eventualmente consistente, con un modelo hub-and-spoke que soportaba plug-and-play.

Este enfoque fue una mejora significativa.

- Los sistemas permanecieron desacoplados y autónomos, sin un único punto de fallo.
- La baja latencia del tiempo casi real redujo considerablemente la necesidad de reconciliación para compensar los errores de la consistencia eventual.
- La deuda técnica y la complejidad de la integración espagueti punto a punto fueron eliminadas,
- y el lead time se mantuvo bajo control, ya que los equipos no tenían que coordinarse con todos los demás.

Sin embargo, este enfoque no alcanzó una adopción masiva.

Base de datos compartida

En cambio, la tendencia se desplazó hacia la consolidación de datos en una enorme base de datos compartida y monolítica. La base de datos relacional se convirtió en el centro del sistema. **Todas las aplicaciones interactuaban con los mismos datos.**

Pero la base de datos era ahora el único punto de fallo. Cualquier tiempo de inactividad de la base de datos significaba tiempo de inactividad para todo el sistema.

Arquitectura orientada a servicios (SOA)

Mientras tanto, la arquitectura orientada a servicios (SOA) capturó la imaginación de la industria, y el middleware de bus de servicios empresariales (ESB) inundó el mercado.

Sin embargo, el middleware sobrecargado aumentó el lead time y redujo el rendimiento. El ESB era otro recurso compartido que necesitábamos adquirir, aprovisionar, configurar y gestionar.

Microservicios

Finalmente, las deficiencias de SOA, combinadas con el desajuste de impedancia de los despliegues de software monolíticos, llevaron a la adopción de los microservicios.

Por fin comenzábamos a regresar en dirección a los sistemas autónomos. Cada microservicio era desplegable de forma independiente y tenía su propia base de datos. Los microservicios no compartían recursos.

Un único equipo era propietario de un microservicio y de todos sus recursos. Esto ayudó a mantener bajo control la necesidad de comunicación entre equipos.

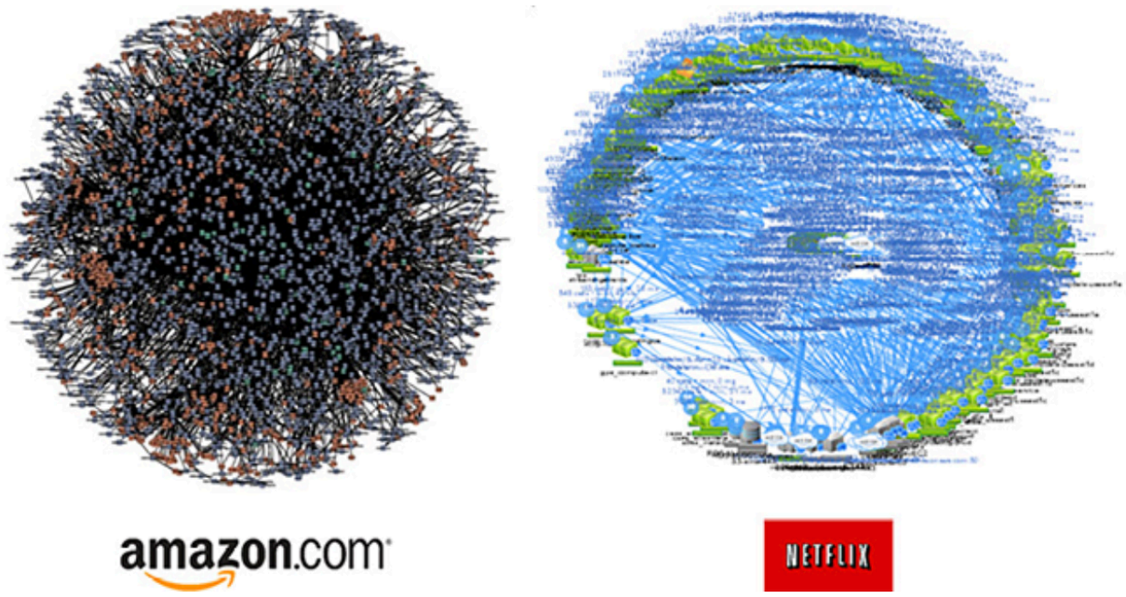


Figure 1.4: Microservices death star

La figura 1.4 muestra ejemplos de la inevitable estrella de la muerte de microservicios que se crea cuando dejamos todas estas dependencias sin control. Es el equivalente moderno de la integración espagueti.

Los propios microservicios individuales se convierten en los recursos compartidos. Una interrupción en un microservicio altamente dependiente se propagaría por todo el sistema, y un cambio en dicho microservicio podría requerir modificaciones en muchos otros microservicios.

El lead time mejoró para cambios pequeños y autocontenidos, pero un error en cualquier cambio seguía representando un riesgo para el sistema.

Equipos autónomos mediante servicios autónomos

Necesitamos una nueva visión arquitectónica. Nuestra arquitectura debe habilitar a los equipos autónomos para moverse rápido y limitar el radio de impacto de los errores involuntarios.

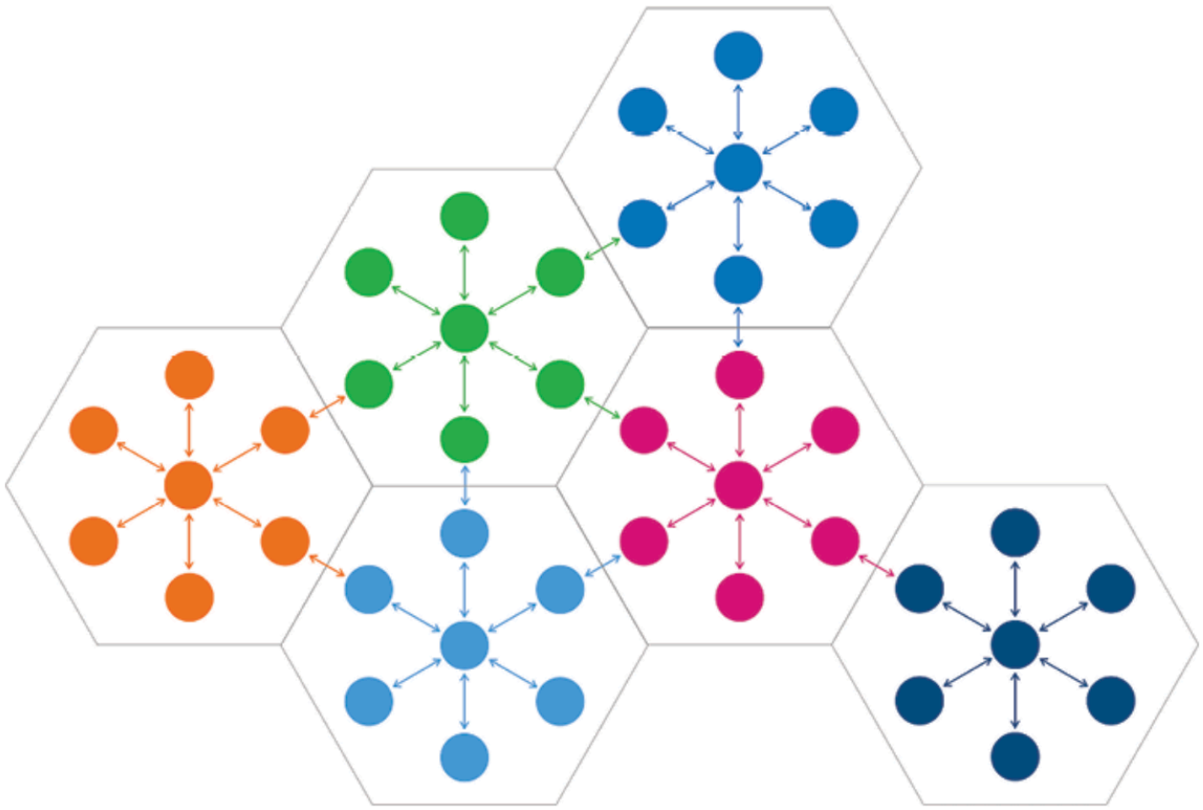


Figure 1.5: Event-first topology of autonomous services and subsystems

Cada servicio autónomo posee todos los recursos que necesita para continuar funcionando incluso cuando otros servicios no pueden. Agrupamos los servicios autónomos en subsistemas autónomos.

En el corazón de cada subsistema hay un event hub que elimina la comunicación directa entre servicios. Luego, conectamos los subsistemas autónomos entre sí para crear sistemas más complejos, simplemente intercambiando los eventos pertinentes.

Servicios autónomos: creación de barreras

Para movernos rápido sin romper nada, necesitamos servicios resilientes. **Necesitamos servicios que continúen funcionando cuando los servicios upstream y downstream relacionados no puedan.**

Queremos componer nuestros sistemas a partir de bloques de construcción simples y repetibles que ensamblamos entre sí. Queremos que el sistema evolucione de forma natural simplemente añadiendo y eliminando **servicios desacoplados**.

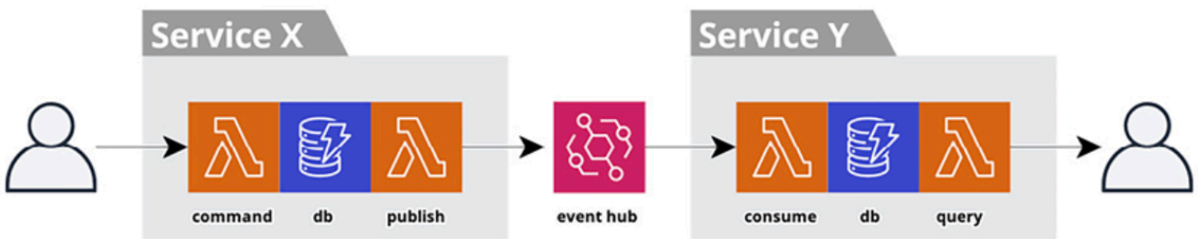
Cada servicio autónomo se enmarcará dentro de uno de nuestros tres patrones de servicio autónomo de orden superior:

- Backend for Frontend (BFF),
- External Service Gateway (ESG),
- o Control Service.

Los desarrolladores pueden trabajar fácilmente en un servicio autónomo sin necesidad de incorporar otros. Pueden razonar fácilmente sobre los elementos internos del servicio que están modificando. Más importante aún, pueden comprender con facilidad el impacto de un cambio determinado sobre los puntos de interacción externos bien definidos.

Todo esto lo logramos simplemente creando barreras entre los servicios. **Estas barreras permiten a los servicios almacenar en búfer los cambios salientes y en caché los datos entrantes.**

Llamo a estos bloques de construcción el flujo Command, Publish, Consume, Query (CPCQ):



El flujo CPCQ divide las interacciones entre servicios en una serie de acciones atómicas. Estas acciones atómicas crean barreras entre los servicios que eliminan la necesidad de comunicación síncrona entre ellos.

Los patrones principales que habilitan este flujo CPCQ son los patrones de event hub y de event sourcing a nivel de sistema.

Comunicación asíncrona entre servicios

En la sección Análisis de los estilos de integración, vimos cómo el estilo de integración síncrona en tiempo real incrementa el acoplamiento entre los componentes de un sistema.

Para dotar a los servicios de autonomía, necesitamos imponer restricciones al uso de la comunicación síncrona. **Restringiremos toda la comunicación síncrona a las interacciones intra-servicio**, y utilizaremos comunicación asíncrona para todas las interacciones entre servicios.

Esto garantizará la capacidad de respuesta de la experiencia de usuario (UX) y mantendrá la simplicidad de la experiencia del desarrollador (DX), al tiempo que elimina el acoplamiento con otros servicios.

Fronteras (boundaries) reforzadas

La comunicación síncrona entre servicios también aumenta el riesgo de interrupciones generalizadas, ya que un fallo en un servicio puede tener un efecto dominó en todos los servicios dependientes.

Una barrera (bulkhead) es un elemento ubicado entre dos componentes que actúa como capa protectora para evitar que un problema se propague de un componente a otro.

Por un lado, la arquitectura es el arte de definir fronteras dentro de un sistema. Por otro lado, la arquitectura es la ciencia de ubicar barreras a lo largo de estas fronteras. **Podemos resumir esto diciendo que la arquitectura de un sistema está definida por sus fronteras reforzadas.**

Las barreras son los almacenes de datos serverless, altamente disponibles y completamente gestionados, que ubicamos a lo largo del sistema para almacenar en búfer y en caché los datos a medida que se mueven por el sistema. Incluyen tanto barreras de salida como de entrada.

- El **event hub** proporciona la **barrera de salida** que protege a los servicios upstream de los servicios downstream. Un servicio puede publicar eventos al event hub sin conocimiento de los consumidores downstream ni de su disponibilidad. El event hub persistirá y almacenará los eventos en búfer hasta que los servicios downstream puedan consumirlos.

- El **almacén de datos** de un servicio autónomo individual proporciona la **barrera de entrada** que protege a un servicio downstream de los servicios upstream. Un servicio downstream consume eventos del hub y crea una caché replicada liviana en su propio almacén de datos. El servicio downstream ya no depende de la disponibilidad de los servicios upstream.

Cierto acoplamiento permanece, ya que los servicios downstream dependen de la estructura de los eventos de dominio producidos por los servicios upstream. Sin embargo, podemos mantener este acoplamiento bajo control practicando el **principio de robustez**.

Event-first: los hechos como valor

La arquitectura orientada a eventos (EDA) ha tenido dificultades para alcanzar una adopción masiva y aprovechar su potencial. La programación basada en eventos es indispensable en la capa de GUI, y el modelo orientado a eventos de Function-as-a-Service (FaaS) es un gran avance en la dirección correcta. **Sin embargo, las empresas continúan creando frágiles estrellas de la muerte de microservicios porque creen que son menos complejas.**

Inversión de responsabilidad

Una preocupación en torno a la EDA gira en torno a la impresión de que estos sistemas son más complejos y difíciles de depurar y gestionar.

En la parte frontal de estos servicios tenemos las acciones síncronas tradicionales que proporcionan una experiencia de desarrollo simple y directa. **En la parte posterior, tenemos las acciones asíncronas que producen eventos y hacen que nuestros sistemas sean más flexibles.**

Al enfocarnos primero en los eventos que producen nuestros servicios, creamos una inversión de responsabilidad que delega las decisiones clave de diseño aguas abajo. Diseñamos cada servicio para dar soporte a un actor específico.

- Los **servicios upstream** publican eventos para registrar el hecho de que un actor realizó una acción, sin preocuparse por lo que ocurre después.
- Los **servicios downstream** asumen la responsabilidad de reaccionar a estos eventos según consideren conveniente.

Este enfoque en los verbos del dominio de negocio, en lugar de sus sustantivos, desacopla naturalmente las funcionalidades de un sistema y lo hace más flexible y adaptable. Habilita el cambio y la innovación a través de la evolución orgánica. **Simplemente añadimos o eliminamos servicios para cambiar cómo reacciona el sistema.**

Los eventos como ciudadanos de primera clase

Tradicionalmente hemos tratado los eventos como mensajes efímeros; es decir, los eliminábamos después de procesarlos. El único registro que teníamos de que estos eventos existieron alguna vez eran los efectos secundarios que tenían sobre sus consumidores.

Estos sistemas no eran autocorrectivos. Requerían intervención manual. Requerían reconciliación de datos. **Esta fue, en gran medida, la razón de la mala reputación de la EDA.**

Esta vez, los eventos representan el contrato del sistema, y les otorgamos un alto valor como los hechos del sistema. **Almacenamos los eventos en un event lake de forma permanente.**

Este repositorio de eventos es un registro de verdad para un sistema. Proporciona un registro de auditoría completo del historial del sistema y es una fuente valiosa de métricas.

Idempotencia y tolerancia al orden

En generaciones anteriores de EDA, las cosas solían salir mal porque **depositábamos nuestra confianza en la falsa suposición de que podíamos garantizar la llegada ordenada de los mensajes.**

En cambio, nuestros servicios autónomos son idempotentes y tolerantes al orden. Cubriremos los patrones y técnicas que nos ayudan a hacer que nuestros servicios sean adaptables y autocorrectivos.

Autoservicio

Serverless es autoservicio. Podíamos delegar las responsabilidades de los equipos centrales al proveedor de nube. No necesitábamos compartir recursos. Los equipos podían aprovisionar recursos a voluntad. Estos recursos eran altamente disponibles y altamente observables. Los equipos podían ponerse en marcha con nuevas tecnologías rápidamente. Podíamos llegar a producción de forma segura sin la curva de aprendizaje habitual.

Arquitectura desechable

Sin embargo, no estábamos creando dependencia del proveedor (vendor lock-in). Estas decisiones eran desechables. **Los equipos podían comenzar con serverless-first y cambiar más adelante si tenía sentido.**

Y estas decisiones no son monolíticas: podemos tomarlas y deshacerlas servicio por servicio.

Debemos comenzar con una solución serverless y generar conocimiento. Aprendemos sobre la tecnología. Pero, más importante aún, aprendemos lo más rápido posible si estamos o no en el camino correcto hacia el valor de negocio.

Quizás algunas funcionalidades necesitarán en última instancia tecnología autogestionada, pero eso es difícil de predecir de antemano y obstaculizaría la innovación. Diferir el compromiso es simplemente el costo de generar ese conocimiento.

Una vez que un equipo sabe que su funcionalidad proporciona valor de negocio, puede aprovechar las métricas de observabilidad y costo de su tecnología serverless para tomar una decisión informada sobre si vale la pena realizar el cambio.

En definitiva, **serverless-first hace que los equipos autónomos sean viables.** Los equipos controlan sus propios lead times. No tienen que depender de otros equipos. Tienen la tranquilidad de que la tecnología es estable, escalable, segura, desechable y está bajo su control. Pueden enfocar su tiempo y esfuerzo en iterar hacia una solución óptima y entregar valor de negocio.

Ciclo de vida del dato: combate de la gravedad de los datos

Con nuestros servicios autónomos y el enfoque de eventos primero, nos enfocaremos en los verbos. Aprovecharemos el patrón BFF y adaptaremos los servicios a actores específicos para realizar funciones concretas (es decir, verbos) a lo largo del ciclo de vida del dato. Estos servicios encapsularán sus propias versiones livianas del modelo de datos lógico y optimizarán los datos en cada etapa de su vida. Los patrones de event sourcing a nivel de sistema y CQRS proporcionarán los mecanismos para mover los datos a lo largo de su ciclo de vida.

Figure 1.7: Data life cycle

El enfoque serverless-first hace que esta persistencia políglota sea práctica para todos los equipos. Daremos vuelta a nuestras bases de datos tradicionales de adentro hacia afuera, distribuyendo los datos a través de la capacidad de la nube y, en última instancia, convirtiendo la nube en la base de datos.

Eliminaremos la gravedad de los datos y liberaremos los datos para que nuestra arquitectura habilite el cambio en lugar de aplastar toda la innovación bajo el peso de los datos.

Micro frontends: equiparación de capas

Las innovaciones en la capa de presentación son interminables. El número de alternativas técnicas para implementar la UX está en constante cambio.

Una innovación en particular es la **single-page application** (SPA) escrita en JavaScript. Este enfoque ha creado numerosas oportunidades para reducir el lead time.

Ya no tenemos que ejecutar ningún servidor para la capa de presentación: es completamente serverless. Esto es una gran ventaja para los equipos autónomos, autosuficientes y full-stack.

Pero aún queda trabajo por hacer en la capa de presentación. Necesitamos descomponer el monolito de la capa de presentación.

Los servicios autónomos son una gran victoria para la reducción del lead time, ya que permiten desplegar funcionalidades de forma independiente. Sin embargo, **cuando seguimos creando aplicaciones frontend monolíticas, marginamos esta mejora.**

Para maximizar su rendimiento, los equipos autónomos necesitan control total e igualitario de todas las capas: presentación, servicio y datos.

Los micro frontends son otra innovación importante. Este enfoque divide la capa de presentación en un conjunto de microaplicaciones desplegables de forma independiente, pero las une para ofrecer una UX fluida. La combinación de los patrones de micro frontends, BFF y CQRS brinda a los equipos autónomos la libertad de experimentar e iterar hacia el valor de negocio óptimo.

Observabilidad: optimización integral

La observabilidad es un concepto crítico en el aseguramiento de calidad (QA) que ayuda a dar a los equipos la confianza para aumentar su frecuencia de despliegue.

Un sistema emite esencialmente señales que podemos evaluar en busca de anomalías. Podemos alertar al equipo sobre los indicadores clave de rendimiento (KPIs) para que puedan actuar de inmediato cuando una funcionalidad falla.

El simple hecho de saber que estamos recopilando y monitorizando activamente esta información refuerza la confianza del equipo. En lugar de temer los despliegues, los equipos pueden enfocarse en mejorar su tiempo medio de recuperación (Mean Time To Recover MTTR) reduciendo los lead times y desarrollando memoria muscular.

La observabilidad habilita la gobernanza continua. Podemos monitorizar los cambios en la configuración de recursos en la nube para verificar el cumplimiento normativo, especialmente en materia de seguridad.

Los equipos autónomos son libres de innovar, con estos monitores de configuración actuando como red de seguridad. Las métricas sobre el rendimiento de los equipos, como el lead time y el MTTR, también ofrecen información sobre la madurez de los equipos.

También podemos usar métricas sobre el tráfico de usuarios y la utilización de funcionalidades para evaluar el éxito o el fracaso de nuestros experimentos.

La observabilidad también habilita el desarrollo basado en el valor. Los equipos pueden aprovechar las métricas de costo y rendimiento para decidir cómo aplicar mejor su capacidad limitada.

En definitiva, necesitamos medir y observar todo aquello que queramos optimizar.

Evolución orgánica: adopción del cambio

Como arquitectos, debemos sentirnos cómodos trabajando en un entorno dinámico de cambio continuo.

Nuestro trabajo es aprovechar esta disrupción creativa e impulsar el cambio en toda la organización. Necesitamos realizar estos cambios para poder entregar valor de negocio de manera oportuna. **Esto incluye cambios en la arquitectura, las metodologías y las prácticas, y, en última instancia, en la cultura y la burocracia.**