2. Definición de límites y cesión del control

Necesitamos una arquitectura que promueva la autonomía. En última instancia, lo logramos mediante la creación de servicios autónomos con límites bien reforzados.

Pero antes de poder reforzar nuestros límites, necesitamos definirlos. Los definiremos en múltiples niveles: el nivel de subsistema, el de servicio y el de función. Definimos estos límites desde el principio para que los equipos autónomos puedan trabajar en paralelo con barreras de seguridad establecidas mientras experimentan dentro de sus propios límites.

Construcción sobre conceptos probados

La arquitectura serverless se apoya en los mismos conceptos probados que deberíamos aplicar independientemente del software que escribamos.

Utilizaremos Domain-Driven Design (DDD), aplicaremos los principios SOLID y nos esforzaremos por implementar una arquitectura hexagonal limpia. Una de las diferencias más interesantes, sin embargo, es que aplicaremos estos conceptos en múltiples niveles.

Los aplicaremos de manera diferente en:

- el nivel de subsistema (es decir, macro arquitectura),
- el nivel de servicio (es decir, micro arquitectura),
- y el nivel de función (es decir, nano arquitectura).

Domain-driven design

Cuando comencé mi carrera, nuestra industria luchaba con el cambio de paradigma hacia la programación orientada a objetos. Pero no fue hasta que el libro Domain-Driven Design de Eric Evans se popularizó que todos contamos con una semántica común para producir diseños más familiares.

Bounded context

Un bounded context define un límite claro entre distintas áreas de un dominio. Dentro de un bounded context, el modelo de dominio debe ser consistente y claro.

Entre bounded contexts, usamos context maps para definir las relaciones entre los modelos. La definición de estos límites es el eje central de este capítulo.

Domain aggregate

Un domain aggregate es una entidad de dominio de nivel superior que agrupa entidades relacionadas en una unidad cohesiva. Intercambiaremos datos entre servicios y subsistemas a este nivel de agregado.

Domain event

Un domain event representa un cambio de estado en un domain aggregate. Estos eventos son fundamentales en nuestra arquitectura orientada a eventos y el corazón de nuestros servicios autónomos.

Principios SOLID

Robert Martin fue el primero en promover estos principios para mejorar los diseños orientados a objetos. Desde entonces, han demostrado ser igualmente valiosos a nivel arquitectónico.

Principio de responsabilidad única

El Principio de Responsabilidad Única (SRP, por sus siglas en inglés) establece que un módulo debe ser responsable ante un único actor, y solo uno.

En apariencia, el nombre de este principio sugiere que un módulo debe hacer una sola cosa y nada más. Este malentendido lleva a la creación de flotas de microservicios muy pequeños, muy interdependientes y fuertemente acoplados.

La definición original y más extendida de este principio establece que un módulo debe tener **una, y solo una, razón para cambiar.** La palabra clave aquí es cambio.

En su definición más reciente del SRP, Uncle Bob se centra en el origen del cambio. Son los actores (es decir, las personas) quienes impulsan los cambios en los requisitos.

El objetivo de este principio es evitar una situación en la que tengamos demandas en competencia sobre un módulo de software que, con el tiempo, lo paralicen y, en última instancia, obstaculicen el cambio.

Logramos este objetivo definiendo límites arquitectónicos para los distintos actores.

Principio de abierto-cerrado

El Principio de Abierto-Cerrado (OCP, por sus siglas en inglés) establece que un módulo debe estar abierto para su extensión, pero cerrado para su modificación. Bertrand Meyer es el autor original de este principio.

Naturalmente ralentizamos el ritmo al modificar software existente, porque debemos considerar todos los impactos de ese cambio. Si nos preocupan los efectos secundarios no deseados, incluso podemos resistirnos al cambio por completo.

Por el contrario, **añadir nuevas capacidades a un sistema es mucho menos intimidante.** Somos naturalmente más confiados y estamos más dispuestos a avanzar cuando sabemos que los escenarios de uso existentes no han sido modificados en absoluto.

Los eventos serán nuestro principal mecanismo para lograr esta libertad. Podemos extender los servicios existentes para producir nuevos tipos de eventos sin efectos secundarios. Podemos añadir nuevos servicios que produzcan tipos de eventos existentes sin impactar a los consumidores actuales. Y podemos añadir nuevos consumidores sin modificar los productores existentes.

En la capa de presentación, los micro frontends serán nuestro principal mecanismo de extensión.

Principio de sustitución de Liskov

El Principio de Sustitución de Liskov (LSP, por sus siglas en inglés) establece que los objetos de un programa deben poder reemplazarse por instancias de sus subtipos sin alterar la corrección de dicho programa. Barbara Liskov es la autora original de este principio.

El principio de sustitución es esencial para crear una arquitectura evolutiva. La mayoría de las innovaciones consistirán en cambios incrementales. Sin embargo, algunas requerirán cambios significativos y precisarán ejecutar múltiples versiones de una capacidad de forma simultánea.

Utilizaremos eventos para definir los contratos de comunicación entre servicios. Este enfoque de diseño por contrato habilita la sustitución que potencia el **enfoque de branch-by-abstraction**. Podemos sustituir productores de eventos y reemplazar consumidores de eventos.

Principio de segregación de interfaces

El Principio de Segregación de Interfaces (ISP, por sus siglas en inglés) establece que ningún cliente debe verse obligado a depender de interfaces que no utiliza.

Nuestro objetivo es crear una arquitectura que habilite el cambio para poder reducir nuestros lead times, desplegar con mayor frecuencia y estrechar el ciclo de retroalimentación. Esto requiere la confianza de que nuestros despliegues no romperán el sistema. **Facilitamos esto creando interfaces limpias y concisas que segregamos de otras interfaces para evitar contaminarlas con preocupaciones innecesarias.**

Este es un motor principal detrás de la creación de interfaces específicas para cada cliente utilizando el patrón Backend for Frontend.

Para toda nuestra comunicación entre servicios, nuestros domain events individuales ya están bien segregados, pues podemos cambiar cada uno de ellos de forma independiente. Sin embargo, debemos considerar que muchos servicios downstream dependerán de estos eventos.

Lo gestionaremos dividiendo el sistema en subsistemas de servicios relacionados y utilizando domain events internos para la comunicación intra-subsistema y domain events externos para la comunicación inter-subsistema.

Principio de inversión de dependencias

El Principio de Inversión de Dependencias (DIP, por sus siglas en inglés) establece que un módulo debe depender de abstracciones, no de implementaciones concretas.

También se le conoce como Inversión de Control (IoC). Se manifiesta en el concepto de Inyección de Dependencias (DI), que se convirtió en una necesidad absoluta en los sistemas monolíticos.

Utilizaremos DI simple basada en constructores en nuestras funciones serverless, sin necesidad de un framework pesado.

La estabilidad ante el cambio es la motivación principal detrás de los servicios autónomos. Necesitamos la capacidad de modificar y evolucionar continuamente un sistema en ejecución manteniendo la estabilidad.

Esa constante en un sistema event-first son los domain events. Cuando modificamos cualquier servicio, todos los demás se apoyarán en la estabilidad de los tipos de eventos que comparten.

En última instancia, esto significa que los servicios upstream son responsables del flujo de control y los servicios downstream son responsables del flujo de dependencias. En otras palabras, **los servicios upstream controlan cuándo se produce un evento y los servicios downstream controlan quién consume esos eventos**.

Llevar el DIP al siguiente nivel implica que los servicios downstream reaccionan a los domain events, y los servicios upstream no invocan a los servicios downstream. Esto es una inversión de responsabilidad que conduce a sistemas más evolutivos.

Arquitectura hexagonal

Partiendo de los principios SOLID, necesitamos una arquitectura que nos permita ensamblar nuestro software de maneras flexibles. Nuestro software necesita la capacidad de ejecutarse en distintos entornos de ejecución, como:

- una función serverless,
- un contenedor serverless,
- o una herramienta de prueba.

Lo más importante es que necesitamos la capacidad de probar nuestra lógica de dominio de forma aislada respecto a cualquier recurso remoto. Para soportar esta flexibilidad, **necesitamos una arquitectura de software con bajo acoplamiento que oculte estas decisiones de diseño técnico de la lógica de dominio.**

La arquitectura hexagonal recibe su nombre de las formas hexagonales que utilizamos en los diagramas. Usamos hexágonos simplemente porque nos permiten representar sistemas como una estructura de panal compuesta por módulos con bajo acoplamiento.

En nuestra arquitectura serverless aplicaremos los conceptos hexagonales en tres niveles distintos: el nivel de función, el de servicio y el de subsistema, como se muestra en la Figura 2.1:

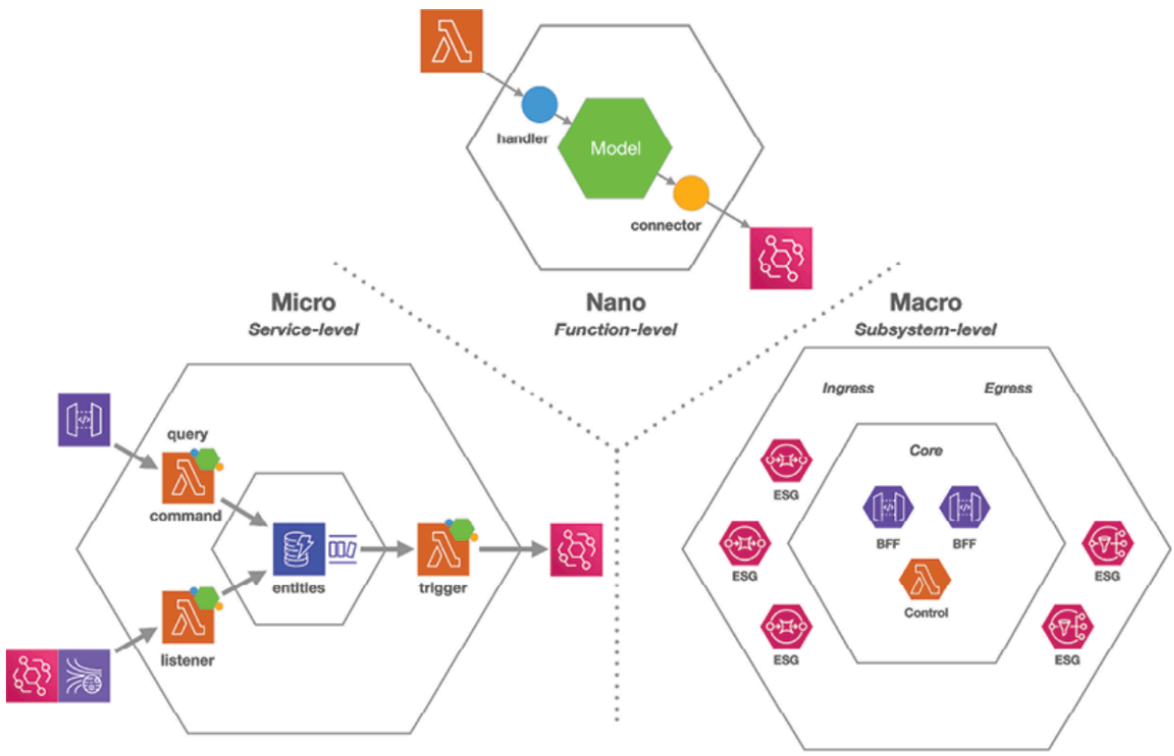


Figure 2.1: Hexagonal architecture

Nivel de función (nano)

La arquitectura hexagonal a nivel de función, o nano, describe la estructura y el propósito del código dentro de una función serverless. Este nivel es el más similar a la arquitectura hexagonal tradicional, **pero lo escalamos hacia abajo hasta el alcance de una función serverless individual**.

El handler y el connector adaptan el Modelo a los servicios en la nube.

Nivel de servicio (micro)

La arquitectura hexagonal a nivel de servicio, o micro, describe la estructura y el propósito de los recursos dentro de un servicio autónomo.

Este nivel es menos tradicional porque distribuimos el código entre múltiples funciones serverless. **Las entidades del modelo de dominio interno residen en un datastore dedicado para que podamos compartirlas entre todas las funciones.**

- Las funciones listener y trigger adaptan el modelo de dominio interno a los domain events que se intercambian entre servicios.
- Las funciones command y query adaptan el modelo para la comunicación con el frontend.

Nivel de subsistema (macro)

La arquitectura hexagonal a nivel de subsistema, o macro, describe la estructura y el propósito de los servicios autónomos dentro de un subsistema autónomo.

Este nivel difiere de la arquitectura hexagonal tradicional porque los adaptadores son servicios completos en lugar de simples artefactos de código.

El núcleo del subsistema está compuesto por servicios Backend for Frontend (BFF) y servicios de Control que trabajan juntos para implementar el modelo de dominio, y los domain events internos intercambiados entre estos servicios definen los puertos del modelo.

Los servicios External Service Gateway (ESG) de ingreso y egreso adaptan los domain events externos a domain events internos.

Pensar en los eventos primero

Para cumplir esta promesa, necesitamos cambiar la manera en que actuamos, lo que significa que necesitamos una forma diferente de pensar.

1. Comenzar con event storming
2. Enfocarse en verbos en lugar de sustantivos
3. Tratar los eventos como hechos en lugar de mensajes efímeros
4. Invertir las APIs tratando los eventos como contratos
5. Invertir la responsabilidad de la invocación
6. Conectar servicios a través de un event hub

Inicio con event storming

El event storming es una técnica orientada a talleres que ayuda a los equipos a descubrir el comportamiento de su dominio de negocio.

Comienza con una lluvia de ideas. El equipo empieza por reunir un conjunto inicial de domain events en un tablero usando notas adhesivas naranjas. A continuación, ordenamos las tarjetas para representar el flujo de eventos.

El event storming facilita el descubrimiento de historias de usuario y los límites dentro de nuestra arquitectura de software.

Enfoque en verbos en lugar de sustantivos

Esta forma de pensar event-first es diferente porque se centra en los verbos en lugar de los sustantivos del dominio de negocio.

Necesitamos segregar los distintos comandos entre los diferentes actores. Al enfocarnos en los verbos del modelo de dominio, nos vemos naturalmente impulsados a crear servicios para los distintos actores que realizan las acciones.

Por supuesto, ahora que los actores son el punto focal de nuestros servicios, necesitaremos una forma de compartir los sustantivos (es decir, los domain aggregates) entre servicios sin incrementar el acoplamiento. **Necesitamos un registro de la verdad**.

Tratamiento de los eventos como hechos en lugar de mensajes efímeros

Pensamos en tiempo pasado y, por lo tanto, nos enfocamos en los hechos que el sistema producirá a lo largo del tiempo. Esto es poderoso en múltiples aspectos:

- Resulta que estamos incorporando implícitamente capacidades de analítica de negocio y observabilidad al sistema.
- También podemos utilizar esta información para validar la hipótesis de cada experimento lean que realizamos, para ayudar a garantizar que estamos construyendo el sistema correcto y cumpliendo con nuestros objetivos de negocio.

Para convertir los eventos en hechos, **debemos tratarlos como ciudadanos de primera clase** en lugar de mensajes efímeros.

En su lugar, trataremos los eventos como hechos inmutables y los almacenaremos en un event lake de forma permanente.

Inversión de las APIs mediante el tratamiento de eventos como contratos

Muchos sistemas orientados a eventos utilizan los eventos únicamente para notificaciones. Estos eventos anémicos solo contienen el identificador de la entidad del dominio de negocio que generó el evento.

Para capturar los hechos completamente, necesitamos que los eventos representen una instantánea del estado del domain aggregate en el momento en que ocurrió el evento.

Esto nos permite tratar los hechos como un log de auditoría análogo al log de transacciones de una base de datos.

Esencialmente estamos invirtiendo la base de datos para crear un registro de la verdad a nivel de todo el sistema que podemos aprovechar para gestionar el estado y la integridad del sistema completo.

Esto significa que estamos invirtiendo nuestras APIs utilizando eventos como contratos entre servicios. **Esto también implica una garantía de compatibilidad retroactiva,** y por lo tanto crearemos contratos sólidos entre servicios dentro de un subsistema y contratos aún más sólidos entre subsistemas.

Inversión de la responsabilidad de invocación

Elevamos el *Principio de Inversión de Dependencias (DIP)* al nivel arquitectónico utilizando eventos como la abstracción (es decir, contrato) entre servicios autónomos.

El uso de eventos para la comunicación entre servicios da lugar a una inversión de responsabilidad que hace que los sistemas sean reactivos.

El enfoque tradicional e imperativo de implementación de sistemas se centra en los comandos. Un componente determina cuándo invocar a otro.

Esta inversión de responsabilidad es una característica clave de los servicios autónomos. Reduce enormemente la complejidad de los servicios individuales porque reduce sus responsabilidades.

Los servicios downstream asumen la responsabilidad de cómo reaccionan a los eventos upstream. Esto desacopla por completo los servicios entre sí. Todos son autónomos.

La naturaleza reactiva del pensamiento event-first es un cambio de paradigma, pero los beneficios bien valen el esfuerzo. Un sistema queda libre para evolucionar de maneras imprevistas simplemente añadiendo consumidores.

Conexión de servicios a través de un event hub

El pensamiento event-first nos permite crear sistemas de complejidad arbitraria conectando subsistemas en una topología fractal simple.

En el corazón de nuestra arquitectura event-first se encuentra el event hub. Conecta todo y bombea eventos a través del sistema. Cada subsistema tiene un event hub en su centro, y cada servicio autónomo se conecta al event hub a través de puertos bien definidos (es decir, eventos).

> Desde un único servicio hasta muchos servicios en un subsistema, y desde un único subsistema hasta muchos subsistemas en un sistema, este simple patrón de conectar servicios y subsistemas autónomos produciendo y consumiendo eventos se repite ad infinitum.

División de un sistema en subsistemas autónomos

El objetivo de la arquitectura de software es definir límites que permitan a los componentes del sistema cambiar de forma independiente.

Necesitamos dar un paso atrás y ver el panorama general. Debemos descomponer un problema complejo en problemas cada vez más pequeños que podamos resolver individualmente y luego combinar en la solución definitiva.

Necesitamos dividir el sistema en un conjunto manejable de subsistemas de alto nivel, cada uno con una única razón para cambiar. Estos subsistemas constituirán los principales bounded contexts del sistema.

También necesitamos que nuestros subsistemas sean autónomos, de la misma manera en que creamos servicios autónomos. Esto dará a las organizaciones autónomas la confianza para innovar continuamente dentro de sus subsistemas.

Examinemos algunas formas en que podemos dividir un sistema.

División por actor

Un punto de partida lógico para dividir un sistema en subsistemas es siguiendo los límites externos con los actores externos. **Estos actores son los usuarios y los sistemas externos que interactúan directamente con el sistema.**

Durante el taller de event storming, también identificaríamos a los usuarios (amarillo) y los sistemas externos (rosado) que producen o consumen esos eventos.

División por unidad de negocio

Un organigrama típico puede proporcionar perspectivas útiles. Cada unidad será en última instancia la propietaria de negocio de sus subsistemas y, por lo tanto, tendrá un impacto significativo en cuándo y cómo cambia el sistema.

Sin embargo, la estructura organizativa de una empresa puede ser inestable. Una empresa puede reorganizar sus unidades de negocio por diversas razones. Por lo tanto, debemos examinar con mayor profundidad el trabajo que las unidades de negocio realmente realizan.

División por capacidad de negocio

En última instancia, queremos trazar nuestros límites arquitectónicos en torno a las capacidades de negocio reales que la empresa provee. Cada subsistema autónomo debe:

- encapsular una única capacidad de negocio
- o, como máximo, un conjunto de capacidades altamente cohesivas.

Volviendo a nuestro enfoque de event storming y nuestro pensamiento event-first, buscamos agrupaciones lógicas de eventos relacionados (es decir, verbos). Habrá una alta cohesión temporal dentro de estos conjuntos de domain events. **Serán iniciados por un grupo de actores relacionados que trabajan juntos para completar una actividad.**

La clave aquí es que la cohesión temporal de las actividades dentro de una capacidad ayuda a garantizar que los componentes de un subsistema tiendan a cambiar juntos.

División por ciclo de vida

A lo largo de la vida de un dato, los actores que lo utilizan e interactúan con él cambiarán, y también lo harán sus requisitos.

Incorporar el ciclo de vida del dato a la ecuación ayudará a descubrir algunos subsistemas pasados por alto. Generalmente encontraremos estos subsistemas cerca del inicio y del final del ciclo de vida del dato.

Volviendo al pensamiento event-first, damos un paso atrás y nos tomamos un momento para enfocarnos en los sustantivos (es decir, los domain aggregates) con el fin de descubrir más verbos (es decir, domain events) y los actores que producen esos eventos.

División por sistema legado

Sin entrar en detalles aquí, estamos creando una capa anti-corrupción alrededor de los sistemas legados que les permite interactuar con el nuevo sistema produciendo y consumiendo domain events.

Esto crea una ruta de migración evolutiva que minimiza el riesgo manteniendo los sistemas legados activos y sincronizados hasta que estemos listos para retirarlos.

Esencialmente estamos tratando los sistemas legados como un subsistema autónomo con barreras que diseñamos para eliminar el acoplamiento entre lo antiguo y lo nuevo, y para proteger la infraestructura legada controlando la superficie de ataque y proporcionando contrapresión.

Creación de barreras en subsistemas

Nuestro objetivo es reforzar todos los límites arquitectónicos de un sistema para que los equipos autónomos puedan avanzar con sus experimentos, con la confianza de que el radio de impacto quedará contenido cuando los equipos cometan errores.

Examinemos cómo podemos reforzar nuestros subsistemas autónomos con barreras.

Cuentas de nube separadas

Las cuentas de nube forman barreras naturales que debemos aprovechar tanto como sea posible para protegernos de nosotros mismos.

Con demasiada frecuencia sobrecargamos nuestras cuentas de nube con demasiadas cargas de trabajo no relacionadas, lo que pone en riesgo todas las cargas de trabajo. Como mínimo, los entornos de desarrollo y producción deben estar en cuentas separadas.

Pero podemos hacerlo mejor teniendo cuentas separadas, por subsistema y por entorno. Esto ayudará a controlar el radio de impacto cuando se produzca un fallo en una cuenta.

Estos son los beneficios de utilizar múltiples cuentas:

- Controlamos la deuda técnica que se acumula naturalmente a medida que crece el número de recursos dentro de una cuenta. **Etiquetar los recursos ayuda, pero son propensos a omisiones.**
- Mejoramos nuestra postura de seguridad limitando la superficie de ataque de cada cuenta.
- Tenemos menos competencia por los recursos limitados. Muchos recursos de nube tienen límites flexibles a nivel de cuenta que restringen el acceso cuando los volúmenes de transacciones superan un umbral.
- La asignación de costos es simple y resistente a errores porque asignamos todo lo que hay en una cuenta a un único cubo de costos sin necesidad de etiquetado. **También minimizamos los costos ocultos de los recursos huérfanos porque son más fáciles de identificar.**
- La observabilidad y el gobierno son más precisos e informativos porque las herramientas de monitoreo etiquetan todas las métricas por cuenta.

Domain events externos

Ya hemos discutido los beneficios de usar eventos como contratos, la importancia de la compatibilidad retroactiva y cómo la comunicación asíncrona mediante eventos crea una barrera a lo largo de nuestros límites arquitectónicos.

Dentro de un subsistema, sus servicios se comunicarán mediante domain events internos. Las definiciones de estos eventos son relativamente fáciles de cambiar porque los equipos autónomos que poseen los servicios trabajan juntos en la misma organización autónoma.

Las definiciones de los eventos comenzarán siendo imprecisas, pero evolucionarán y se estabilizarán rápidamente a medida que el subsistema madure.

Los eventos también contendrán información en bruto que queremos conservar para fines de auditoría, pero que no tiene importancia fuera del subsistema.

Por el contrario, a través de los límites de los subsistemas, necesitamos una comunicación y coordinación entre equipos más regulada para facilitar los cambios en estos contratos.

Queremos limitar el impacto que esto tiene en el lead time interno, para ser libres de innovar dentro de nuestros subsistemas autónomos. En esencia, queremos ocultar la información interna y no exponer nuestros asuntos internos en público.

En cambio, realizaremos toda la comunicación entre subsistemas mediante domain events externos (a veces llamados eventos de integración). Estos eventos externos tendrán contratos mucho más estables, con requisitos de compatibilidad retroactiva más estrictos. Pretendemos que estos contratos cambien lentamente para contribuir a crear una barrera entre subsistemas. Domain-Driven Design (DDD) se refiere a esta técnica como context mapping, como cuando usamos domain aggregates con los mismos términos en múltiples bounded contexts, pero con significados diferentes.

Cada subsistema tratará a los subsistemas relacionados como sistemas externos.

- Cada subsistema definirá gateways de egreso que especifican qué eventos está dispuesto a compartir y ocultarán todo lo demás.
- Los subsistemas definirán gateways de ingreso que actúan como una capa anti-corrupción para consumir domain events externos upstream y transformarlos (es decir, adaptarlos) a sus formatos internos.

Análisis de un subsistema autónomo

En este punto, hemos dividido nuestro sistema en subsistemas autónomos. Cada subsistema es responsable ante un único actor dominante que impulsa el cambio.

Diagrama de contexto

Aplicaremos un conjunto de patrones de servicios autónomos para descomponer un subsistema en servicios.

Necesitamos conocer todos los actores externos con los que interactuará el subsistema. Durante el event storming, identificamos el comportamiento del sistema y los usuarios y sistemas externos involucrados.

Un diagrama de contexto simple puede contribuir en gran medida a que todos estén alineados respecto al alcance de un subsistema. **El diagrama de contexto enumera todos los actores externos del subsistema usando tarjetas amarillas para los usuarios y tarjetas rosadas para los sistemas externos.**

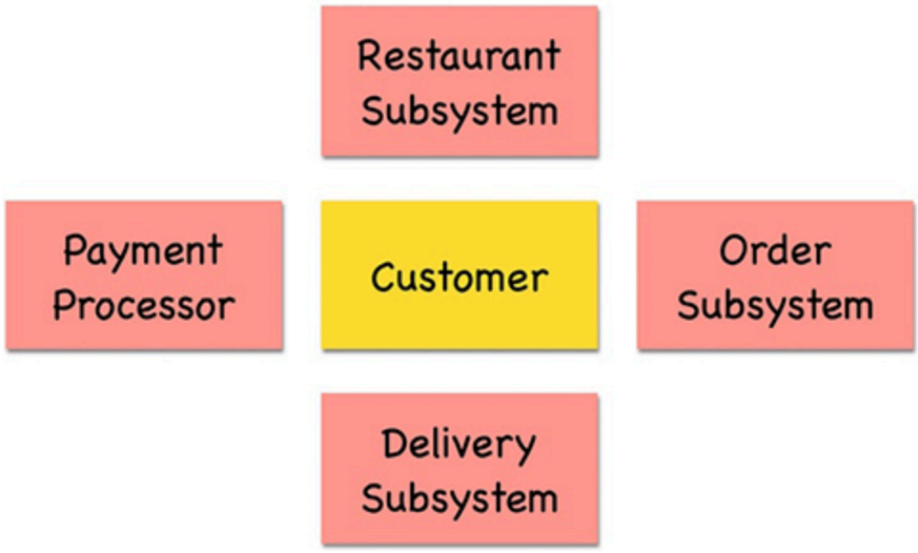


Figure 2.7: Subsystem context diagram

Ahora que el contexto está claro, podemos comenzar a descomponer el sistema en su frontend, servicios y componentes comunes.

Micro frontend

Cada subsistema autónomo es responsable ante un único usuario primario o un único grupo cohesivo de usuarios.

Cada subsistema proporcionará su propio punto de entrada independiente para que no quede sujeto a los requisitos cambiantes de otro subsistema.

La interfaz de usuario no será monolítica. Implementaremos el frontend utilizando micro-apps autónomas que se despliegan de forma independiente. El punto de entrada principal actuará como un sistema de ensamblaje y menú basado en metadatos.

Event hub

Cada subsistema autónomo contendrá su propio event hub independiente para soportar la comunicación asíncrona entre servicios dentro del subsistema.

Los servicios publicarán domain events en el event hub cuando su estado cambie. El event hub recibirá los eventos entrantes en un bus. Enrutará todos los eventos al event lake para su almacenamiento permanente, y los enrutará a uno o más canales para que los consuman los servicios downstream.

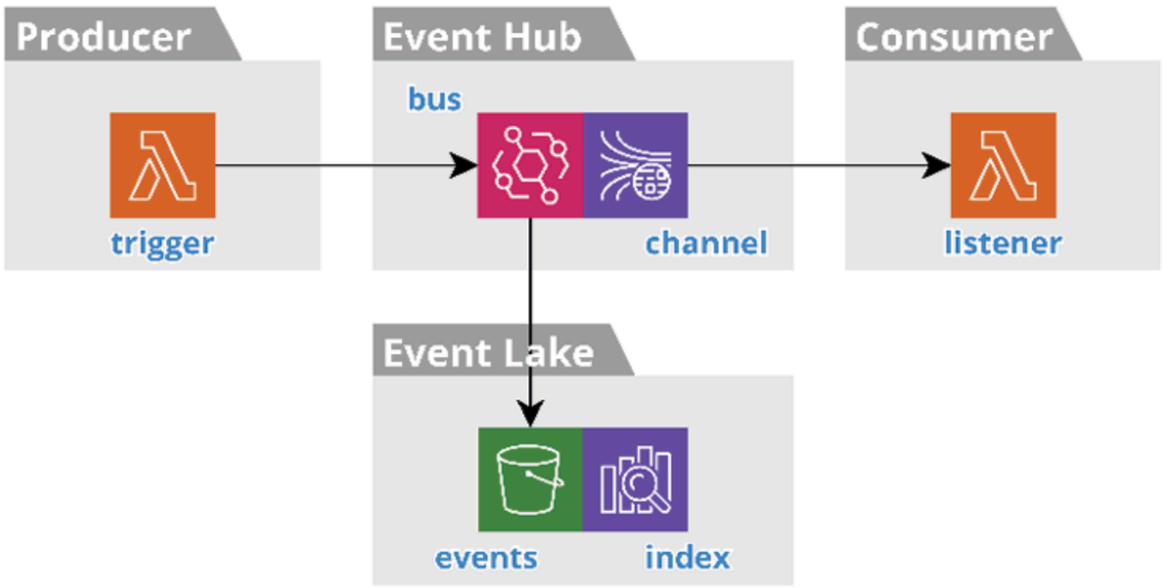


Figure 2.8: Event hub

Patrones de servicios autónomos

En los límites de nuestros subsistemas autónomos se encuentran:

- el patrón Backend For Frontend (BFF)
- y el patrón External Service Gateway (ESG).

Entre los patrones de límite se encuentra el patrón de servicio de Control.

Backend for frontend

El patrón Backend For Frontend (BFF) opera en el límite del sistema para dar soporte a los usuarios finales. Cada servicio BFF soporta una micro-app de frontend específica, que a su vez soporta a un actor específico.

Una función listener consume domain events del event hub y almacena en caché entidades en vistas materializadas que soportan las consultas. La API síncrona proporciona operaciones de comando y consulta que soportan la interfaz de usuario específica.

Una función trigger reacciona a las mutaciones provocadas por los comandos y produce domain events hacia el event hub.

External service gateways

El patrón External Service Gateway (ESG) opera en el límite del sistema para proporcionar una capa anti-corrupción que encapsula los detalles de la interacción con otros sistemas, como sistemas de terceros, sistemas legados y subsistemas hermanos.

Una función de egreso consume eventos internos del event hub y luego transforma y reenvía los eventos hacia el otro sistema.

Una función de ingreso reacciona a eventos externos en otro sistema y luego transforma y reenvía esos eventos hacia el event hub.

Servicios de control

El patrón de servicio de Control ayuda a minimizar el acoplamiento entre servicios mediando la colaboración entre los servicios de límite.

Estos servicios encapsulan las políticas y reglas que administran los propietarios del negocio. Son completamente asíncronos. Consumen eventos, ejecutan lógica y producen nuevos eventos para registrar los resultados y desencadenar el procesamiento downstream.

Utilizamos estos servicios para realizar procesamiento complejo de eventos y orquestar procesos de negocio. **Se apoyan en el patrón de event sourcing a nivel de sistema y dependen de las propiedades ACID 2.0 (Asociativo, Conmutativo, Idempotente y Distribuido).**

Una función listener consume eventos de orden inferior del event hub y los correlaciona y agrupa en un micro events store. Una función trigger aplica reglas a los eventos correlacionados y publica eventos de orden superior de vuelta al event hub.

Análisis de un servicio autónomo

Cada equipo autónomo tiene la responsabilidad última de tomar las decisiones que mejor sirvan a los servicios que posee.

Adoptar una mentalidad de políglota en todo y empoderar a los equipos para tomar estas decisiones les otorga la libertad que necesitan para maximizar la innovación.

Repositorio

Cada servicio tiene su propio repositorio de código fuente. Esto se debe, en parte, a que las herramientas modernas de control de código fuente distribuido, como Git, facilitan enormemente la creación y distribución de nuevos repositorios.

Además, las herramientas modernas de pipelines CI/CD asumen que el repositorio es la unidad de despliegue. Todos estos factores nos orientan hacia esta buena práctica.

Sin embargo, la razón más importante por la que cada servicio tiene su propio repositorio es la autonomía. Queremos reducir nuestros lead times, y compartir un repositorio con otros equipos inevitablemente generará fricción y ralentizará a los equipos.

Pipeline CI/CD y GitOps

Cada servicio tiene su propio pipeline CI/CD, definido mediante un archivo de configuración en la raíz de su repositorio. **Los pipelines CI/CD modernos potencian el concepto de GitOps, que es la práctica de utilizar pull requests de Git para orquestar el despliegue de la infraestructura.**

Pruebas

Las pruebas automatizadas desempeñan un papel fundamental para dar a los equipos la confianza necesaria para desplegar de forma continua.

Con este fin, los casos de prueba constituyen la mayor parte del código base de una funcionalidad, y las pruebas unitarias representan la mayoría de los casos de prueba.

Ejecutamos pruebas unitarias, pruebas de integración, pruebas de contrato y pruebas transversales de extremo a extremo en el pipeline CI/CD, en aislamiento de todos los recursos externos.

Stack

Desplegamos cada servicio como un conjunto de recursos de nube al que denominaremos stack. Definimos declarativamente los recursos de un stack en un archivo de configuración serverless.yml en la raíz del repositorio.

El servicio de gestión de despliegues administra el ciclo de vida de los recursos como un grupo. Compara el estado actual del stack con las últimas declaraciones y aplica los cambios necesarios.

Persistencia

Cada servicio poseerá y gestionará sus propios datos. Siguiendo las prácticas de persistencia políglota, cada servicio utilizará el tipo de base de datos que mejor se adapte a sus necesidades. Estos recursos serverless se gestionan como parte del stack.

Los servicios consumirán domain events de los servicios upstream y almacenarán en caché los datos necesarios como vistas materializadas ligeras. **La alta disponibilidad de estos datastores serverless crea una barrera de entrada que garantiza que los datos necesarios estén disponibles incluso cuando los servicios upstream no lo estén.**

Aprovecharemos el mecanismo de **Change Data Capture (CDC)** de un datastore para desencadenar la publicación de domain events cuando el estado de los datos cambie.

API trilateral

Cada servicio tendrá hasta tres APIs:

- una para los eventos que consume,
- otra para los eventos que produce,
- y una para su interfaz síncrona.

Eventos

Siguiendo nuestro enfoque event-first, las APIs para eventos son las más importantes, pues dictan cómo un servicio interactuará con otros servicios.

Un servicio debe documentar los eventos que consume y los que produce. Esto puede ser tan simple como un listado en el archivo README en la raíz del repositorio. Podemos documentar la estructura JSON de los domain events internos usando la notación de interfaces de TypeScript.

Para los domain events externos, un estándar como OpenAPI (https://www.openapis.org) o JSON Schema (https://json-schema.org) puede resultar útil.

La infraestructura de nube puede proporcionar un servicio de registro para capturar los esquemas de todos los tipos de eventos del subsistema.

También puede utilizar una herramienta como Event Catalog (https://www.eventcatalog.dev) para facilitar la exploración de los productores y consumidores en su subsistema.

API Gateway

Los servicios que operan en los límites del sistema, como los BFFs, tendrán una interfaz síncrona. Los implementaremos usando un API Gateway.

Diseñamos la API de un BFF específicamente para una única micro-app de frontend. Un equipo es propietario tanto del frontend como del BFF, por lo que la documentación oficial de la API puede no ser necesaria.

Algunos servicios ESG también requerirán un API Gateway, como al implementar un webhook para recibir eventos de un sistema de terceros o al proporcionar una Open API para el propio sistema SaaS.

Funciones

Implementaremos la lógica de negocio de un servicio autónomo como funciones serverless utilizando la oferta Function-as-a-Service (FaaS) del proveedor de nube.

Es importante señalar que aunque cada función es independiente, **gestionaremos las funciones como un grupo dentro del stack que posee todos los recursos del servicio.**

Arquitecturamos nuestras funciones en dos niveles: el nano o nivel de función y el micro o nivel de servicio. En otras palabras, necesitamos arquitecturar el código dentro de cada función y las funciones dentro de cada servicio.

Nano arquitectura

En el nivel de función o nano, escalamos hacia abajo nuestra arquitectura hexagonal para que cada función serverless ejecute de manera limpia un fragmento de la lógica de negocio dentro de un servicio autónomo.

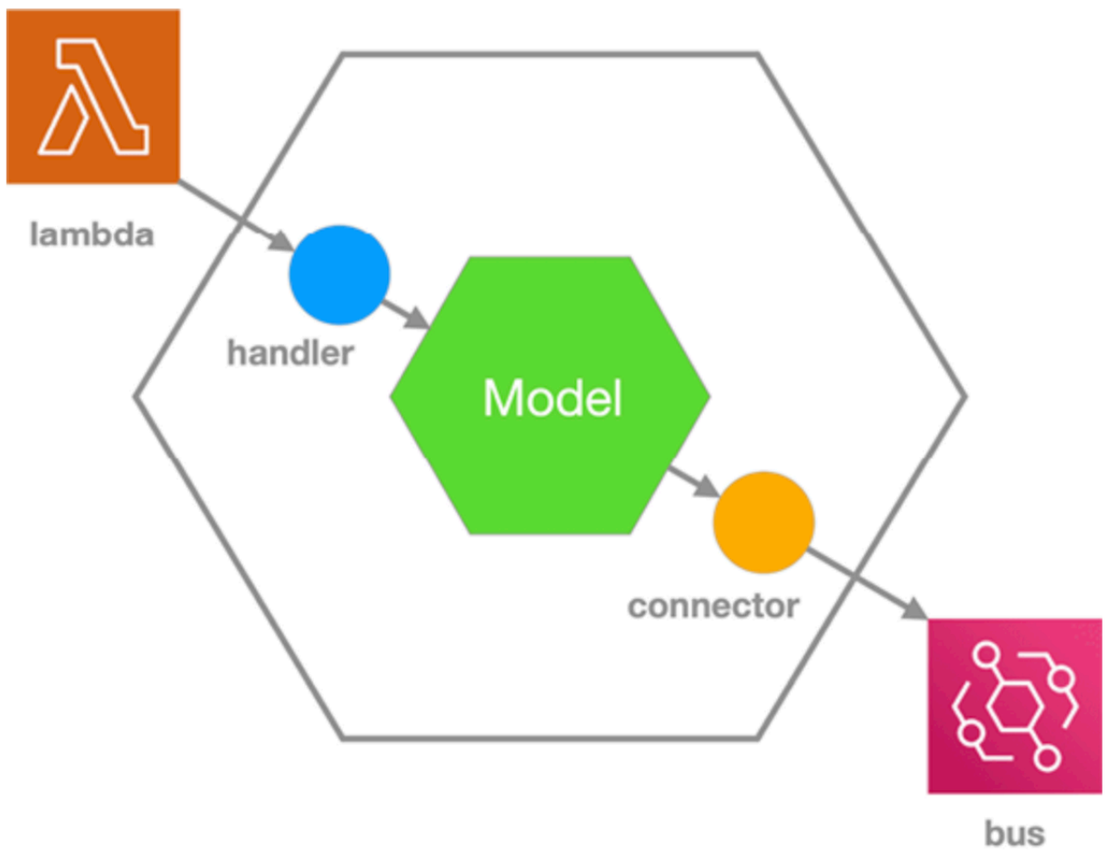


Figure 2.11: Function-level – nano hexagonal architecture

El servicio FaaS (es decir, lambda) invoca una función serverless y dicta la firma de los parámetros de entrada. No queremos que esta firma contamine nuestra lógica de negocio. A su vez, nuestra lógica de negocio realiza las llamadas salientes a los servicios de nube, como un bus o un datastore.

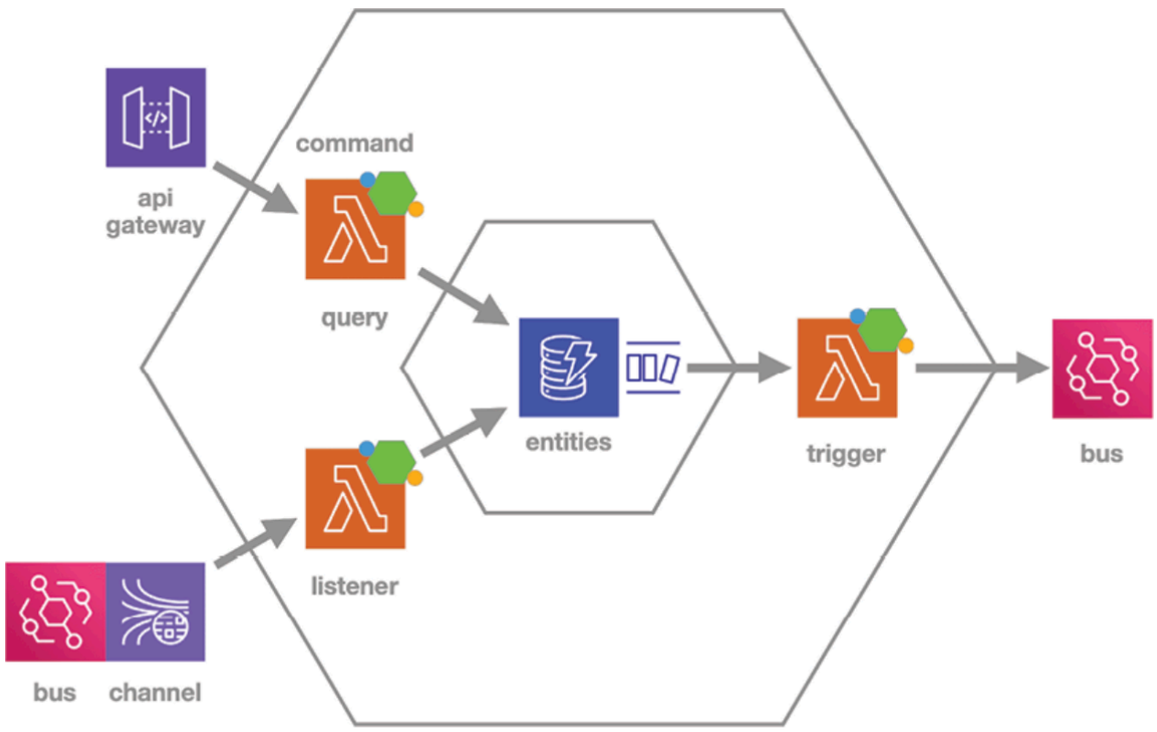
Nuevamente, no queremos que las firmas de estos servicios de nube contaminen nuestra lógica de negocio. Por ello, separaremos la lógica de negocio en un modelo y aislaremos estas dependencias en adaptadores.

- Implementaremos la lógica de negocio (es decir, el Modelo) como clases y funciones que exponen interfaces agnósticas a la nube (es decir, puertos).
- Implementaremos una función handler que adapta la firma de lambda al modelo.
- El handler inyectará al modelo una clase connector que adapta las llamadas salientes del modelo a la firma del servicio de nube, como el bus.

Esta nano arquitectura nos permitirá mover fácilmente el modelo a un entorno de ejecución diferente o incluso a otro proveedor de nube sustituyendo distintas implementaciones para los handlers y connectors.

Micro arquitectura

En el nivel de servicio o macro, escalamos nuestra arquitectura hexagonal para mostrar cómo múltiples funciones serverless trabajan juntas para implementar la lógica de negocio del servicio autónomo.



- Almacenamos el modelo de dominio interno en un datastore dedicado (entidades) para poder compartir los datos entre todas las funciones.
- Estos datos representan el modelo de dominio interno del servicio, y ese formato de datos define la interfaz (es decir, los puertos).
- Las funciones serverless actúan como adaptadores para mapear entre el modelo interno y el modelo externo.

Un servicio autónomo tendrá típicamente entre dos y cinco funciones serverless. Cada función tiene su propia nano arquitectura con el handler y el connector adecuados según los servicios de nube con los que interactúa.

La función listener se sitúa en el lado conductor porque consume domain events de un canal y recopila los datos que son un prerrequisito para las capacidades del servicio. **Adapta los datos de los domain events entrantes al modelo de dominio interno** y guarda los resultados en el datastore de entidades local.

La función command/query se sitúa en el lado conductor porque soporta las acciones del usuario que dirigen el trabajo a través del subsistema. **El nano modelo implementa la lógica de negocio para las consultas y comandos** y la función adapta el modelo para la comunicación con el frontend.

El trigger se sitúa en el lado conducido porque reacciona a los cambios de estado dentro del servicio. Consume eventos de cambio internos del stream CDC del datastore. **Adapta los datos de los eventos de cambio al modelo de domain event y publica los resultados.**

Los niveles nano y micro trabajan juntos para mantener nuestra arquitectura limpia y desacoplada. Habilitan el patrón Command, Publish, Consume, Query (CPCQ) que utilizamos para conectar servicios entre sí y llevar a cabo el trabajo de un subsistema autónomo.

- El trabajo comienza upstream. Un usuario realiza una acción y una función command actualiza el estado del modelo de dominio.
- Luego, una función trigger publica un domain event para registrar el hecho de que ocurrió la acción y desencadenar el procesamiento posterior.
- Downstream, una o más funciones listener asumen la responsabilidad de consumir el domain event y almacenan en caché los datos necesarios para que sus usuarios puedan consultar la información que necesitan para realizar la siguiente acción.
- Repetimos este simple patrón tantas veces como sea necesario para entregar la funcionalidad deseada de un subsistema autónomo.

Librerías compartidas

Tenderemos a evitar el uso de librerías compartidas para reutilizar lógica de negocio entre servicios. Esto nos ayudará a evitar la falsa reutilización que genera acoplamiento y complejidad. Utilizaremos librerías de código abierto para las preocupaciones transversales.

Con el tiempo, la lógica duplicada puede refactorizarse en librerías cuando se demuestre que no es falsa reutilización.

Gobernar sin obstaculizar

Como arquitectos, una vez que hemos definido los límites arquitectónicos del sistema, necesitamos soltar el control y apartarnos del camino, a menos que queramos convertirnos en un impedimento para la innovación.

El gobierno tiene una reputación comprensible de obstaculizar el progreso y la innovación. Aunque tiene buenas intenciones, el enfoque manual tradicional del gobierno en realidad aumenta el riesgo en lugar de reducirlo, porque incrementa el lead time, lo que disminuye la capacidad de una organización para reaccionar ante los desafíos en un entorno dinámico moderno.

Definimos límites arquitectónicos que limitan el alcance de cualquier cambio dado, y reforzamos estos límites para controlar el radio de impacto cuando ocurren errores humanos honestos.

En lugar de obstaculizar las innovaciones, debemos empoderar a los equipos con una cultura y una plataforma que abrazan el gobierno continuo. Esto es una red de seguridad que da confianza a los equipos y a la dirección para avanzar, sabiendo que podemos detectar errores y hacer correcciones en tiempo real.

Automatización y preocupaciones transversales

Un objetivo principal del gobierno es garantizar que un sistema cumpla con las regulaciones y las mejores prácticas. Estas incluyen los atributos de calidad típicos (las «-ilidades»), como la escalabilidad y la confiabilidad, y por supuesto la seguridad, junto con regulaciones como NIST, PCI, GDPR e HIPAA.

Afortunadamente, ahora tenemos una mejor opción. **Nuestros despliegues están completamente automatizados por nuestros pipelines CI/CD.** Esto ya es una mejora significativa en la calidad porque Infrastructure as Code reduce el error humano y nos permite fallar adelante rápidamente. Aún tenemos algunas puertas manuales para cada despliegue.

- La primera puerta es la revisión de código y la aprobación de un pull request. Ejecutamos esta puerta rápidamente porque cada rama de tarea tiene un tamaño de lote pequeño.
- La segunda puerta es la certificación de un despliegue canario regional. Desplegamos en una región para realizar pruebas de humo continuas antes de desplegar en otras regiones.
- También contamos con observabilidad, que proporciona información oportuna y accionable para saber cuándo intervenir y recuperarnos rápidamente.
- Llevaremos la automatización más lejos y reforzaremos nuestros procesos de build añadiendo auditoría continua y asegurando el perímetro de nuestros subsistemas y cuentas de nube.

Sin embargo, todas estas son preocupaciones transversales, y **no queremos que los equipos reinventen estas capacidades para cada subsistema autónomo.** Necesitamos un equipo dedicado con el conocimiento y las habilidades especializadas para gestionar una suite integrada de herramientas SaaS, provisionar cuentas con un conjunto estándar de capacidades y mantener estas preocupaciones transversales para uso en todas las cuentas.

Fomento de una cultura de robustez

Nuestro objetivo de aumentar el ritmo de innovación nos lleva a un ciclo de retroalimentación rápido con tamaños de lote pequeños y lead times cortos. **Estamos desplegando código con mucha mayor frecuencia y estos despliegues deben resultar en cero tiempo de inactividad.**

> **El principio de robustez establece: sé conservador en lo que envías, sé liberal en lo que recibes.**

Este principio es muy adecuado para el despliegue continuo, donde podemos realizar una secuencia sucesiva de despliegues:

1. Para hacer un cambio conforme en un lado del contrato,
2. seguido de una actualización en el otro lado
3. y luego otra en el primer lado para eliminar el código antiguo.

El truco es desarrollar una cultura de robustez donde este baile de tres pasos quede grabado en la memoria muscular del equipo y se convierta en algo natural.

En mi experiencia, los equipos autónomos están ansiosos por adoptar una cultura de robustez, especialmente una vez que perciben cuánto más productivos y efectivos pueden llegar a ser. **Pero esto es un cambio de paradigma, y resulta poco familiar desde una perspectiva de gobierno tradicional.**

Aprovechamiento de las cuatro métricas clave de equipo

Los equipos autónomos son responsables de aprovechar las métricas de observabilidad de sus aplicaciones y servicios como herramienta de autogobierno y mejora continua.

Nicole Forsgren, Gene Kim y Jez Humble propusieron cuatro métricas que podemos aprovechar para ayudarnos a identificar qué equipos pueden necesitar más asistencia y mentoría:

- **Lead time:** ¿Cuánto tiempo le toma a un equipo completar una tarea y llevar el cambio a producción?
- **Tasa de despliegue:** ¿Cuántas veces al día despliega un equipo cambios a producción?
- **Tasa de fallos:** ¿Con qué frecuencia un despliegue resulta en un fallo que impacta una funcionalidad disponible de forma general?
- **Tiempo medio de recuperación (MTTR):** Cuando ocurre un fallo, ¿cuánto tiempo le toma al equipo fallar adelante con una corrección?

Los equipos que tienen dificultades con estas métricas suelen estar atravesando su propia transformación digital y están ansiosos por recibir mentoría y coaching.

Podemos recopilar métricas de nuestro software de seguimiento de incidencias y de nuestra herramienta CI/CD, y hacer seguimiento junto con todas las demás en nuestra herramienta de observabilidad.