BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 11

Evaluación, toma de decisiones y gobernanza

Este capítulo trata de cómo evaluar, gobernar y hacer evolucionar las arquitecturas a lo largo del tiempo. Abarca la evaluación formal con el ATAM, las revisiones ligeras como el risk-storming y los pre-mortems, las funciones de aptitud y la arquitectura evolutiva, la gobernanza en organizaciones ágiles, Team Topologies y el diseño sociotécnico, la modernización de sistemas heredados con el strangler fig, el liderazgo y la influencia, y la justificación económica y ética del trabajo de arquitectura. Reúne todos los capítulos anteriores, usa la plataforma de reparto de comida como ejemplo recurrente y está escrito para desarrolladores con experiencia, líderes técnicos y arquitectos que toman o guían decisiones significativas.

bewiseapps.com/es/software-architecture-wise/evaluation-and-governance/ · Versión 2026-11-03. Solo información educativa, no asesoramiento profesional.

DE UN VISTAZO

Aprenderás a

- Explicar cuándo y por qué evaluar una arquitectura, y aplicar los nueve pasos del ATAM para obtener riesgos, no riesgos, puntos de sensibilidad, puntos de compromiso y temas de riesgo.
- Facilitar revisiones ligeras como el risk-storming y los pre-mortems, y convertir sus resultados en un registro de riesgos con responsables.
- Diseñar funciones de aptitud que conviertan los escenarios de atributos de calidad en comprobaciones automatizadas de la estructura, el rendimiento, la seguridad, la resiliencia y los datos.
- Redactar principios de arquitectura y elegir entre guardrails, compuertas y el proceso de consejo para gobernar distintos tipos de decisión.
- Aplicar los tipos de equipo, los modos de interacción y la carga cognitiva de Team Topologies para alinear los equipos con la arquitectura prevista.
- Planificar una modernización incremental de un sistema heredado con el strangler fig, branch by abstraction y ejecuciones en paralelo.
- Liderar e influir en las decisiones arquitectónicas sin autoridad formal, también al comunicarse con la dirección ejecutiva.
- Calcular el valor actual neto y el periodo de recuperación de una justificación económica, y revisar una arquitectura en busca de inquietudes éticas y de sostenibilidad.

Antes de empezar

Capítulos 1 a 4, en especial las decisiones y los compromisos (Sección 1.6), la deuda técnica (Sección 1.7), los escenarios de atributos de calidad y los árboles de utilidad (Secciones 2.3 y 2.8), la ley de Conway (Sección 3.7) y los registros de decisiones (Sección 4.5). Los Capítulos 5 a 10 ayudan, pero no son imprescindibles.

Tiempo

Unas 3½ horas de lectura; 45 minutos para la prueba.

Términos clave

ATAM · Árbol de utilidad · Punto de sensibilidad · Punto de compromiso · Risk-storming · Pre-mortem · Función de aptitud (fitness function) · Arquitectura evolutiva · Proceso de consejo (advice process) · Radar tecnológico · Team Topologies · Strangler fig (higuera estranguladora) · Valor actual neto · Software Carbon Intensity

Evaluar, gobernar y hacer evolucionar la arquitectura

Los capítulos anteriores enseñaron a describir lo que un sistema debe lograr, a estructurarlo y a hacerlo rápido, fiable, seguro y económico. Este capítulo trata del trabajo que rodea esas decisiones durante toda la vida de un sistema: comprobar que son sólidas antes de comprometer el dinero, mantenerlas vigentes mientras llegan cientos de cambios cada mes, decidir quién puede tomar qué decisiones en una organización con muchos equipos,

reorganizar los equipos y los sistemas heredados, persuadir a las personas y justificar el costo del trabajo de arquitectura en términos económicos y éticos. Es donde el curso se une, así que remite a menudo a lo anterior: a los escenarios de atributos de calidad y los árboles de utilidad del Capítulo 2, a los principios y las métricas del Capítulo 3, a los registros de decisiones del Capítulo 4, a la ley de Conway de la Sección 3.7 y a la deuda técnica de la Sección 1.7.

El nivel es avanzado, pero cada sección empieza con una explicación breve para quienes se inician, y el ejemplo recurrente es la plataforma de reparto de comida que se usa desde el Capítulo 2: clientes, restaurantes y repartidores, servicios de pedidos, despacho y notificaciones, un intermediario de eventos y un proveedor de pagos externo. Verás cómo se evalúa su arquitectura, cómo se le aplica risk-storming, cómo se protege con funciones de aptitud, cómo se gobierna mediante principios y un radar tecnológico, cómo se reorganiza en torno a los equipos y cómo se moderniza en parte.

Evaluar y proteger las decisiones: Secciones 11.1 a 11.3

La Sección 11.1 trata la evaluación de la arquitectura formal y el ATAM del SEI: por qué y cuándo evaluar, sus nueve pasos repartidos en dos reuniones, quién participa y sus resultados característicos: riesgos, no riesgos con sus supuestos, puntos de sensibilidad, puntos de compromiso y temas de riesgo. El árbol de utilidad de la Sección 2.8 vuelve a su contexto original, y el CBAM muestra cómo sopesar el beneficio de cada corrección frente a su costo. Un ATAM resumido de la plataforma de reparto de comida muestra las tablas que produce realmente una evaluación.

La Sección 11.2 explica por qué las juntas de revisión mensuales fracasan tan a menudo y qué funciona en su lugar: el risk-storming de Simon Brown, el pre-mortem de Gary Klein, las katas de arquitectura de Ted Neward, el TARA de Eoin Woods para sistemas existentes, las revisiones de diseño por escrito y las listas de comprobación. Termina con un formato paso a paso para una revisión de 90 minutos y una guía para elegir entre las técnicas.

La Sección 11.3 convierte la evaluación puntual en una protección continua. Una función de aptitud (fitness function) es una comprobación objetiva de una característica arquitectónica, idealmente automatizada: una regla de dependencia, una prueba de carga con umbrales de percentil, un análisis de vulnerabilidades, un experimento de caos o una comprobación de compatibilidad de esquemas. Junto con cambios pequeños, frecuentes y seguros, las funciones de aptitud hacen posible una arquitectura evolutiva. Tres escenarios del Capítulo 2 se convierten en comprobaciones que funcionan, con código.

Gobernar, organizar y modernizar: Secciones 11.4 a 11.6

La Sección 11.4 se pregunta cómo una organización con muchos equipos autónomos mantiene coherente su arquitectura. Distingue entre principios, estándares, guardrails (salvaguardas) y compuertas; muestra cómo redactar principios con un nombre, un enunciado, una justificación y unas implicaciones; y describe la toma de decisiones descentralizada mediante el proceso de consejo (advice process) de Andrew Harmel-Law, los foros asesores de arquitectura y los radares tecnológicos, con el cumplimiento automatizado mediante funciones de aptitud y política como código.

La Sección 11.5 se toma en serio la ley de Conway como herramienta de diseño. Presenta Team Topologies, de Matthew Skelton y Manuel Pais: cuatro tipos de equipo, tres modos de interacción, la carga cognitiva del equipo, las API de equipo y los planos de fractura, y muestra cómo rediseñar los equipos de la plataforma de reparto de comida en torno a flujos de cambio. La Sección 11.6 se ocupa de los sistemas que ya existen: por qué las reescrituras big-bang fracasan tan a menudo, el abanico de estrategias de migración y las técnicas incrementales que funcionan, como el strangler fig, branch by abstraction, las ejecuciones en paralelo, la captura de datos de cambios y las capas anticorrupción, aplicadas paso a paso extrayendo un módulo de pedidos de un monolito con el strangler fig.

Liderar, justificar y actuar con responsabilidad: Secciones 11.7 y 11.8

La Sección 11.7 trata la parte humana del rol: liderar sin autoridad formal, generar confianza, facilitar decisiones por consentimiento o por consenso, gestionar el conflicto y la política organizativa, comunicarse con la dirección ejecutiva, la mentoría, las trayectorias de ingeniería staff-plus y las trampas de decisión en las que caen las personas con experiencia, con una decisión disputada sobre un bus de eventos resuelta paso a paso.

La Sección 11.8 cierra el capítulo con los dos argumentos que un arquitecto más necesita saber defender bien. La justificación económica: costos, beneficios, riesgos y opciones, con el valor actual neto y el periodo de recuperación calculados en un ejemplo práctico, y la deuda técnica expresada en dinero y en tiempo. Y la justificación ética: los códigos de conducta profesionales, los daños previsibles como la pérdida de privacidad, el sesgo, la falta de accesibilidad y el diseño manipulador, y la sostenibilidad medida con la especificación Software Carbon Intensity.

La idea que debes llevarte

La arquitectura no está terminada cuando se dibuja el diagrama. Sus decisiones tienen que ponerse a prueba frente a lo que necesita el negocio, protegerse mientras el sistema cambia, tomarlas las personas adecuadas con el consejo adecuado, llevarlas a cabo equipos cuya

estructura encaje con ellas y justificarse en términos que los patrocinadores y la sociedad puedan aceptar. Los arquitectos más eficaces hacen que este trabajo sea ligero y continuo en lugar de pesado y ocasional: revisiones de riesgos breves en lugar de auditorías anuales, guardrails automatizados en lugar de colas de aprobación, consejo en lugar de permiso, migración incremental en lugar de reescrituras. La disciplina que lo une todo es la misma con la que empezó este curso: nombra la cualidad que necesitas, hazla medible, decide de forma deliberada, anota por qué y sigue comprobando.

SECCIÓN 11.1 DE 11.8 · AVANZADO · 28 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Evaluación de la arquitectura y el ATAM

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.6 Decisiones y compromisos
- 2.3 Escenarios de calidad
- 2.8 Elicitar y priorizar

Términos anteriores que utiliza: Escenario de atributo de calidad, Árbol de utilidad, Atributo de calidad, Parte interesada

LA IDEA EN BREVE

Antes de que una ciudad construya un puente nuevo, ingenieros independientes revisan el diseño: preguntan qué pasa en una inundación, con los camiones más pesados o en una tormenta, y señalan dónde una decisión que ahorra dinero agrava otro peligro. No rediseñan el puente ni le ponen una nota del uno al diez; dicen a los propietarios dónde están los riesgos reales, para que puedan decidir qué hacer mientras los cambios todavía son baratos. La evaluación de la arquitectura hace lo mismo con el software. Su método más conocido, el ATAM, reúne a los propietarios, los constructores y los usuarios del sistema para poner a prueba el diseño frente a las situaciones que más importan.

LO QUE NECESITAS SABER

- La evaluación de la arquitectura comprueba si las decisiones significativas de un sistema le permitirán cumplir sus objetivos de negocio y sus requisitos de calidad, y es más valiosa antes de compromisos importantes o irreversibles.
- El ATAM (Kazman, Klein y Clements, SEI, 2000) sigue nueve pasos repartidos entre la preparación, dos reuniones de evaluación y el seguimiento, con un equipo de evaluación independiente, los responsables de las decisiones del proyecto y un grupo más amplio de partes interesadas.
- Sus resultados son riesgos, no riesgos (con sus supuestos), puntos de sensibilidad, puntos de compromiso y temas de riesgo vinculados a los impulsores de negocio a los que amenazan; no otorga una nota de aprobado.
- El árbol de utilidad que construyen los responsables de las decisiones en el paso 5 se compara con los escenarios que las partes interesadas generan y votan en el paso 7, y las diferencias entre ambos revelan puntos ciegos.

- El CBAM se basa en el ATAM: pondera la utilidad que cada estrategia arquitectónica añade a los escenarios priorizados y la divide por su costo para comparar el valor obtenido por costo, aunque las restricciones pueden seguir imponiéndose a esa proporción.
- Un ATAM completo cuesta decenas de días-persona, así que el método debe ajustarse a lo que está en juego, y sus hallazgos deben convertirse en registros de decisiones, elementos del backlog y funciones de aptitud con responsables.

Por qué evaluar una arquitectura

Toda arquitectura es una apuesta. Cuando un equipo elige un intermediario de eventos, una base de datos por servicio o una sola región de nube, está prediciendo que esas decisiones darán el rendimiento, la disponibilidad, la seguridad y el costo que necesita el negocio. La Sección 1.3 mostró por qué importan esas apuestas: el costo de cambiar una decisión estructural crece con todo lo que se construye encima, y algunos de los fallos más caros del software fueron fallos de organización de las piezas más que de líneas de código concretas. La evaluación de la arquitectura es el acto deliberado de comprobar la apuesta antes de gastar el dinero, o mientras se gasta.

Una evaluación plantea una pregunta concreta y práctica: dado lo que este sistema debe lograr, ¿cuáles de sus decisiones arquitectónicas ponen en riesgo esos objetivos? No es una revisión de código, ni un concurso de belleza entre estilos, ni una auditoría de la calidad de la documentación. Trabaja sobre las decisiones y sus consecuencias para los atributos de calidad, así que puede hacerse mucho antes de que exista código, sobre un diseño en una pizarra o un conjunto de registros de decisiones, y puede repetirse en un sistema que lleva años funcionando.

La evaluación también tiene beneficios más allá de los hallazgos. Las partes interesadas que rara vez se reúnen (responsables de producto, operaciones, seguridad, finanzas, desarrolladores) tienen que ponerse de acuerdo sobre lo que más importa, lo que a menudo saca a la luz supuestos contradictorios. El arquitecto tiene que explicar el diseño con la claridad suficiente para que personas externas puedan examinarlo, lo que mejora la documentación. Y la organización obtiene un registro de los riesgos conocidos que ha decidido aceptar, algo muy distinto de los riesgos que nadie advirtió.

Cuándo evaluar

El mejor momento para evaluar es cuando ya se ha decidido lo suficiente para que la evaluación tenga algo que examinar, pero antes de que esas decisiones resulten caras de revertir. En la práctica hay varios momentos naturales, y las organizaciones maduras aplican a cada uno una evaluación de distinto peso.

Momentos para evaluar

Desencadenante	Qué se comprueba	Peso habitual
Existe una arquitectura candidata, antes de construir a gran escala	Si los estilos y las tácticas principales pueden cumplir los escenarios de calidad priorizados	ATAM completo o reducido para sistemas grandes o de larga vida; si no, una revisión ligera
Antes de un compromiso importante	Un contrato con un proveedor, una migración a la nube, una compuerta de financiación o una decisión de cambio de plataforma que sería costoso deshacer	Formal, a menudo con evaluadores externos, porque la decisión es una puerta de un solo sentido
Cambio significativo en los impulsores de negocio	Nuevos mercados, una fusión, un objetivo de crecimiento por diez, nueva normativa	Rehacer el árbol de utilidad y analizar los escenarios que cambiaron
Cambios acumulados y deriva	Si el sistema tal como está construido sigue correspondiendo a la arquitectura prevista y a sus razones	Revisión ligera periódica más funciones de aptitud automatizadas
Adquisición o due diligence	Si un sistema existente puede sostener los planes del comprador	Evaluación con tiempo limitado de un sistema existente, a menudo combinada con análisis de código
Después de un incidente grave	Si el incidente reveló una debilidad arquitectónica y no un defecto aislado	Análisis centrado en los escenarios afectados

Evaluar demasiado pronto desperdicia esfuerzo en decisiones que todavía no son reales; evaluar demasiado tarde convierte los hallazgos en lamentos. Aquí también sirve una regla útil del modelo dirigido por riesgos de Fairbanks (Sección 1.8): invierte en evaluación en proporción al riesgo. Una herramienta interna de dos equipos no necesita un taller de dos días; una plataforma nacional de pagos, sí.

SI ESTO ES NUEVO PARA TI

Si eres nuevo en esto: qué necesita una evaluación como punto de partida

Una evaluación es tan buena como lo que se le da. Como mínimo necesita los objetivos de negocio, una lista corta de escenarios de atributos de calidad priorizados (Sección 2.3) y una descripción de la arquitectura: un diagrama de contexto y de contenedores (Sección 4.3), los principales flujos en tiempo de ejecución y las decisiones significativas. Si no existen, elaborarlos es la primera parte de la evaluación, y a menudo la más valiosa.

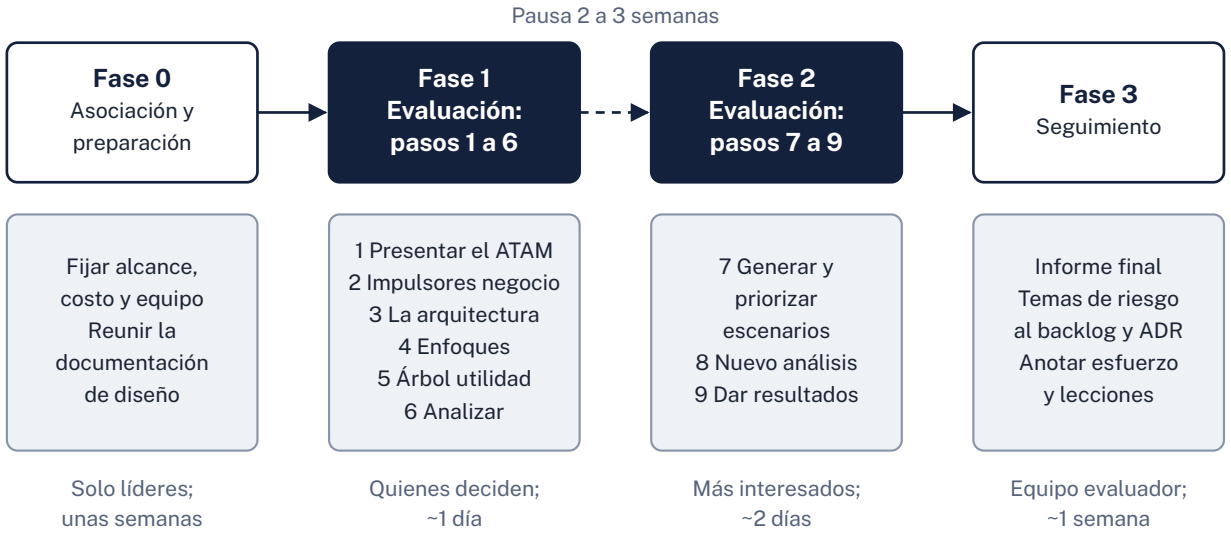
El ATAM: orígenes, propósito y participantes

El Architecture Tradeoff Analysis Method (ATAM) es el método de evaluación más conocido. Rick Kazman, Mark Klein y Paul Clements, del Software Engineering Institute (SEI) de la Carnegie Mellon University, lo describieron en el informe técnico *ATAM: Method for Architecture Evaluation* (CMU/SEI-2000-TR-004, agosto de 2000). Surgió del anterior Software Architecture Analysis Method (SAAM), que Kazman y sus colegas publicaron en 1994 y que se centraba sobre todo en la modificabilidad. El ATAM también se describe en *Software Architecture in Practice*, de Bass, Clements y Kazman, que va ya por su cuarta edición (2021).

Su propósito se malinterpreta a menudo. El ATAM no da a una arquitectura una puntuación ni una nota de aprobado, y no demuestra que un sistema vaya a cumplir sus requisitos. Identifica dónde las decisiones arquitectónicas ponen en riesgo los objetivos de negocio y dónde obligan a hacer compromisos entre cualidades, para que las personas responsables puedan decidir qué hacer. El nombre del método destaca los compromisos porque la mayoría de los riesgos interesantes están donde una cualidad se consigue a costa de otra.

Participan tres grupos. El **equipo de evaluación**, normalmente de tres a cinco personas ajenas al proyecto, dirige el método: un jefe de equipo, un líder de evaluación que facilita, secretarios que recogen los escenarios y el desarrollo de las sesiones, y preguntadores expertos en cualidades concretas. Los **responsables de las decisiones del proyecto** son el arquitecto, el jefe de proyecto o de producto y el patrocinador o cliente que puede actuar sobre los hallazgos. Las **partes interesadas en la arquitectura** son todos los demás que tienen algo en juego: desarrolladores, testers, operaciones, seguridad, integradores, soporte y representantes de los usuarios. La independencia importa: los evaluadores que diseñaron el sistema, o que dependen del arquitecto, encuentran menos riesgos.

Las fases del ATAM



Las dos reuniones de evaluación están separadas por una pausa en la que el equipo asimila los primeros hallazgos y se reúne al grupo más amplio de partes interesadas.

Los nueve pasos

1. **1. Presentar el ATAM**
   El líder de evaluación explica el método, los roles de los participantes y los resultados, para que nadie confunda la sesión con una auditoría o con una revisión de diseño de su trabajo personal.

2. **2. Presentar los impulsores de negocio**
   Un representante sénior del negocio explica los objetivos del sistema, las restricciones, las partes interesadas y las cualidades que más importan, en términos de negocio. Estos impulsores de negocio (business drivers)
   son la vara de medir de todo lo que viene después.

3. **3. Presentar la arquitectura**
   El arquitecto describe la arquitectura con el nivel de detalle adecuado: contexto, estructuras principales, comportamiento en tiempo de ejecución, despliegue y cómo responde a los impulsores. Un límite de tiempo habitual de aproximadamente una hora obliga a centrarse.

4. **4. Identificar los enfoques arquitectónicos**
   El arquitecto nombra los
   enfoques arquitectónicos
   en uso (estilos, patrones y tácticas como un broker, la replicación o la caché) y el equipo de evaluación los cataloga, sin analizarlos todavía.

5 **5. Generar el árbol de utilidad de atributos de calidad**
Los responsables de las decisiones construyen o refinan un árbol de utilidad y valoran cada escenario hoja según su importancia para el negocio y su dificultad técnica.

6 **6. Analizar los enfoques arquitectónicos**
Para los escenarios de mayor prioridad, los evaluadores preguntan qué enfoques responden a cada uno y los examinan con preguntas específicas de cada cualidad, registrando riesgos, no riesgos, puntos de sensibilidad y puntos de compromiso.

7 **7. Generar y priorizar escenarios**
En la fase 2, el grupo más amplio de partes interesadas propone sus propios escenarios y los vota. El resultado se compara con el árbol de utilidad; los escenarios importantes que faltan en el árbol son un hallazgo en sí mismos.

8 **8. Volver a analizar los enfoques arquitectónicos**
Los escenarios nuevos mejor clasificados se analizan del mismo modo que en el paso 6, ahora con el conocimiento de las partes interesadas presentes en la sala.

9 **9. Presentar los resultados**
El equipo de evaluación presenta los enfoques, los escenarios, el árbol de utilidad, los riesgos, los no riesgos, los puntos de sensibilidad, los puntos de compromiso y los temas de riesgo, y relaciona cada tema con los impulsores de negocio a los que amenaza.

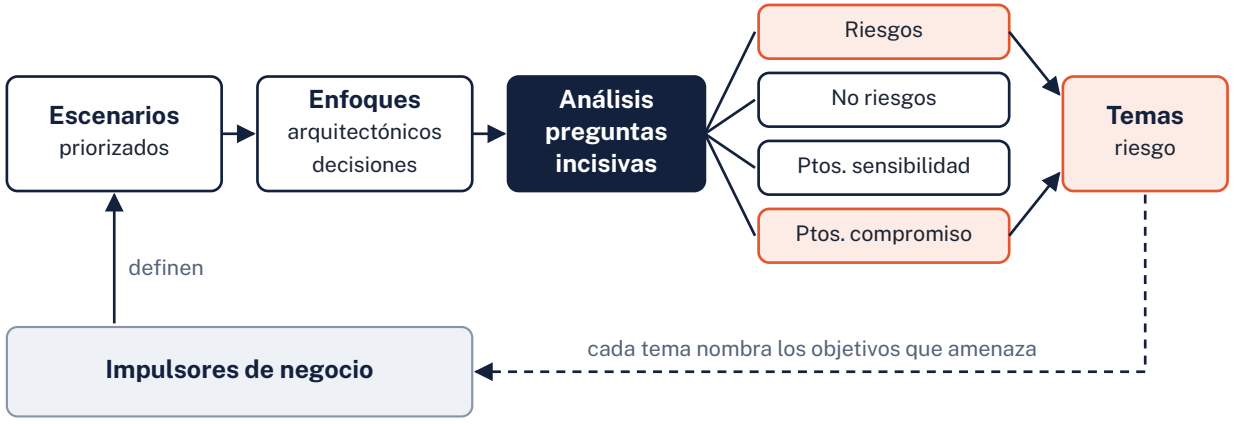
El vocabulario del análisis

El núcleo del método son los pasos 6 y 8. Para cada escenario importante, los evaluadores piden al arquitecto que explique paso a paso cómo responde la arquitectura, y la examinan con preguntas extraídas de cada cualidad: ¿cuántas llamadas síncronas hay en esta ruta?, ¿qué ocurre con las solicitudes en curso cuando falla ese nodo?, ¿dónde se guarda esta credencial?, ¿qué limita la cola? Cada respuesta se clasifica con cinco conceptos que se han convertido en vocabulario habitual mucho más allá del propio ATAM.

Los resultados del análisis

Resultado	Significado	Ejemplo de una plataforma de reparto de comida
Riesgo	Una decisión, o la falta de una decisión, que puede impedir que se cumpla un escenario importante o un objetivo de negocio	La autorización de la tarjeta está dentro de la ruta síncrona del pedido, así que un proveedor de pagos lento ralentiza todos los pedidos
No riesgo	Una decisión que se considera sólida para el escenario, registrada junto con los supuestos de los que depende	La bandeja de salida transaccional y los consumidores idempotentes cumplen el escenario de no perder pedidos, suponiendo que el broker replique cada partición tres veces
Punto de sensibilidad	Una propiedad de los componentes o las conexiones que es crítica para la respuesta de una cualidad	La latencia de los pedidos es sensible al número de saltos síncronos y al tiempo de respuesta p99 del proveedor
Punto de compromiso	Una propiedad que es un punto de sensibilidad para más de una cualidad, de modo que mejorar una empeora otra	El intervalo de actualización de la ubicación del repartidor: acortarlo mejora la actualidad del mapa, pero agota las baterías de los teléfonos y aumenta el costo de los datos móviles y de la ingesta
Tema de riesgo	Un grupo de riesgos con una causa de fondo común, vinculado a los impulsores de negocio a los que amenaza	Los servicios de terceros están en rutas críticas síncronas, lo que amenaza el objetivo de duplicar el volumen de pedidos sin perder conversiones

Cómo produce el análisis sus resultados



El análisis relaciona los escenarios priorizados con las decisiones que responden a ellos; los riesgos resultantes se agrupan en temas que remiten a los objetivos de negocio a los que amenazan.

Los no riesgos merecen más atención de la que suelen recibir. Registrar que una decisión es sólida *gracias* a un supuesto concreto (tres réplicas, un tiempo de respuesta contractual de un proveedor, un tráfico por debajo de cierto nivel) convierte una vaga tranquilidad en algo que se puede comprobar más adelante. Cuando el supuesto cambia, el no riesgo pasa a ser un riesgo candidato. Los puntos de sensibilidad y de compromiso son igual de valiosos después de la evaluación: indican a los futuros mantenedores qué parámetros deben cambiar con cuidado, y son lugares naturales para las comprobaciones automatizadas descritas en la Sección 11.3.

Otra mirada al árbol de utilidad

La Sección 2.8 presentó el árbol de utilidad como herramienta para priorizar los requisitos de calidad. Nació en el ATAM, y dentro del método desempeña dos papeles distintos. En el paso 5, los responsables de las decisiones construyen el árbol de arriba abajo: la utilidad, luego los atributos de calidad, luego los refinamientos y luego los escenarios concretos valorados (importancia para el negocio, dificultad técnica). Las hojas (H,H) se convierten en el orden del día del paso 6, porque son las que más importan y las más difíciles de lograr, así que es ahí donde es más probable que se escondan los riesgos.

En el paso 7, el grupo más amplio de partes interesadas genera escenarios de abajo arriba y los vota, de forma muy parecida a un Quality Attribute Workshop. Comparar las dos visiones es uno de los momentos más reveladores de una evaluación. Si los escenarios principales de las partes interesadas ya aparecen en el árbol, los responsables de las decisiones entienden el propósito de su sistema. Si falta un escenario que recibe muchos votos, por ejemplo un escenario de operaciones sobre tabletas de restaurantes que se quedan sin conexión, los responsables de las decisiones tienen un punto ciego, y puede que la arquitectura también. Esos escenarios se añaden al árbol y se analizan en el paso 8.

EJEMPLO PRÁCTICO

Ejemplo práctico: un ATAM resumido de la plataforma de reparto de comida

La plataforma de los Capítulos 2 y 4 conecta a los clientes, 4000 restaurantes y 9000 repartidores. Los impulsores de negocio son duplicar el volumen de pedidos en 18 meses (de 50 a 100 pedidos por segundo en el pico del viernes) sin perder conversiones, y captar cadenas nacionales de restaurantes que exigen una disponibilidad contractual. La arquitectura: microservicios por contexto delimitado (pedidos, restaurantes, despacho, notificaciones), cada uno con su propia base de datos PostgreSQL; un API gateway; REST síncrono para hacer pedidos, incluida la autorización de la tarjeta con un proveedor de pagos externo; un broker Kafka con una bandeja de salida transaccional para los eventos de pedidos; las posiciones de los repartidores se envían cada 4 segundos y se guardan en Redis; una sola región de nube con tres zonas de disponibilidad, con copias de seguridad nocturnas copiadas en una segunda región. El árbol de utilidad de la fase 1 reutilizó los seis escenarios de la Sección 2.8.

Latencia de los pedidos (H,H)	p95 por debajo de 1.5 s y p99 por debajo de 3 s con 100 pedidos/s. Enfoques: gateway, cuatro saltos síncronos, autorización de la tarjeta dentro de la solicitud. Latencia medida del proveedor: p95 de 900 ms, p99 de 2.5 s
Análisis	El proveedor por sí solo consume la mayor parte del presupuesto de p99 y deja 500 ms para cuatro saltos y las escrituras en la base de datos. La ley de Little usa el tiempo medio, no el p99 (Sección 2.4), pero el riesgo es claro: si una ralentización del proveedor lleva la media a 2.5 s, 100 pedidos/s suponen 250 solicitudes en curso y el pool de 200 workers del servicio de pedidos se satura. **Riesgo R1**; **punto de sensibilidad S1**: número de saltos síncronos y p99 del proveedor
Disponibilidad del pago (H,H)	El proveedor de pagos agota el tiempo de espera en el pico: pedidos aceptados y puestos en cola, ninguno perdido, cliente informado en 10 s. El diseño actual hace fallar el pedido cuando la autorización agota el tiempo de espera. **Riesgo R2**
Opción planteada	Autorizar de forma asíncrona: aceptar el pedido como pendiente y autorizarlo desde una cola. Mejora la latencia y la disponibilidad, pero algunos pedidos aceptados fallan después en el pago y los restaurantes no deben cocinar pedidos sin pagar. **Punto de compromiso T1**: en qué punto del flujo del pedido se confirma el pago
No perder pedidos (parte de H,H)	Outbox en la misma transacción local que el pedido, Kafka con tres réplicas, consumidores idempotentes. **No riesgo N1**, suponiendo un factor de replicación de 3 y confirmaciones del productor de todas las réplicas sincronizadas
Fallo de región (H,M)	RTO de 15 min, RPO de 1 min. Las copias de seguridad nocturnas dan un RPO de hasta 24 horas y una restauración que se mide en horas. **Riesgo R3**

Actualidad del seguimiento (M,M)	La posición en el mapa nunca tiene más de 5 s de antigüedad. 9000 repartidores cada 4 s son 2250 escrituras/s, algo cómodo para Redis. **Punto de compromiso T2**: el intervalo de actualización (actualidad frente a batería, datos y costo)
Robo de cuentas (H,M)	La limitación de tasa (rate limiting) por dirección IP en el gateway no detecta el relleno de credenciales (credential stuffing) repartido entre miles de direcciones. **Riesgo R4**
Generación de escenarios de la fase 2	Las partes interesadas de restaurantes y de operaciones votaron mucho un escenario que faltaba en el árbol: una tableta de restaurante sin conexión durante 10 minutos en el pico debe recibir todos los pedidos al reconectarse. Las notificaciones push se envían una sola vez y sin confirmación de recepción. **Riesgo R5**, y un punto ciego en el árbol
Temas de riesgo	**Tema A** (R1, R2, R4): las dependencias de terceros y del perímetro están en rutas críticas síncronas sin aislamiento; amenaza duplicar el volumen sin perder conversiones. **Tema B** (R3, R5): las garantías de recuperación y de entrega van por detrás de la disponibilidad que se promete a las cadenas de restaurantes; amenaza los contratos con las cadenas
Qué pasó después	Registros de decisiones para la autorización asíncrona con un estado pendiente (aceptando T1, y avisando a los restaurantes solo después de la autorización), una región de reserva en caliente (warm standby), la entrega de pedidos con confirmación de recepción a las tabletas y la detección del relleno de credenciales en el proveedor de identidad. Los supuestos de N1 se convirtieron en una comprobación automatizada de la configuración

Los hallazgos tratan de decisiones, no de personas ni de calidad del código, y cada uno puede rastrearse hasta un escenario y un impulsor de negocio.

Dos de los cinco riesgos salieron de escenarios que el arquitecto no había priorizado o no había tenido en cuenta, y por eso existe el paso de las partes interesadas.

Los puntos de compromiso no son problemas que haya que eliminar: T1 se aceptó deliberadamente y quedó registrado, con sus consecuencias para los restaurantes y los clientes.

CBAM: poner el dinero sobre la mesa

El ATAM te dice dónde están los riesgos; no te dice qué correcciones merece la pena pagar. El Cost Benefit Analysis Method (CBAM), desarrollado en el SEI por Rick Kazman, Jai Asundi y Mark Klein (publicado por primera vez en 2001), se basa en los escenarios de un ATAM para construir esa justificación económica.

A grandes rasgos, las partes interesadas toman los escenarios priorizados y asignan una utilidad (en una escala de 0 a 100) a varios niveles de respuesta de cada uno: el peor caso, el actual, el deseado y el mejor caso. Los arquitectos proponen estrategias arquitectónicas y estiman la respuesta que lograría cada una. La utilidad que gana cada estrategia se lee en la curva de utilidad-respuesta del escenario y se pondera por la prioridad del escenario; sumada en todos los escenarios, es el beneficio de la estrategia. Al dividir el beneficio por el costo se obtiene una relación valor-costo (VFC), y las estrategias se eligen por orden de VFC dentro del presupuesto disponible y luego se contrastan con la intuición y las restricciones.

Un CBAM en miniatura para el riesgo de fallo de región

Estrategia	Respuesta esperada	Ganancia de utilidad x peso	Costo anual	Valor por costo
Actual: copias de seguridad nocturnas	RTO de unas 6 h, RPO de hasta 24 h	utilidad 10 (referencia)	ya pagado	n/a
A: región de reserva en caliente, replicación asíncrona de la base de datos	RTO de 10 min, RPO por debajo de 1 min	(90 - 10) x 0.3 = 24	£180k	24 / 180 = 0.13
B: pilot light (datos replicados, cómputo que se arranca bajo demanda)	RTO de 40 min, RPO por debajo de 1 min	(60 - 10) x 0.3 = 15	£70k	15 / 70 = 0.21

La estrategia B da más beneficio por libra, pero no cumple el tiempo de recuperación de 15 minutos que exigen los contratos con los restaurantes. Los números informan la decisión; no la toman. El verdadero valor del CBAM es que obliga a hacer estimaciones explícitas y comparables del beneficio y del costo, y deja ver cuándo debe decidir una restricción y no una proporción. Su debilidad es que las utilidades y los costos son juicios de valor; los equipos deberían variarlos y ver si cambia la clasificación, exactamente como recomienda la Sección 1.6 para las matrices de decisión ponderadas. La Sección 11.8 muestra cómo convertir esas estimaciones en una justificación económica.

Costos y aspectos prácticos

Un ATAM completo es caro en tiempo de las personas. A modo de ilustración aproximada: un equipo de evaluación de cuatro personas que dedica dos días cada una a la preparación (8 días-persona), una fase 1 de un día con tres responsables de las decisiones (7), una fase 2 de dos días a la que se suman diez partes interesadas (34) y una semana de seguimiento a cargo de

dos personas (10) suman unos 60 días-persona, antes de que nadie corrija nada. Para un sistema cuyo fallo costaría millones, o que va a funcionar durante una década, es un seguro barato. Para un equipo de producto pequeño, es desproporcionado.

Varias prácticas hacen que las evaluaciones merezcan la pena. Ajusta el método: Bass, Clements y Kazman describen una evaluación ligera de la arquitectura, un ATAM reducido para equipos internos que puede completarse en un día o menos, y algunas organizaciones solo hacen los pasos 2 a 6 con un puñado de personas. Prepárate bien: la mayor parte del costo de una mala evaluación se va en descubrir, ya en la sala, que nadie sabe describir la arquitectura. Mantén a los evaluadores independientes pero informados. Acuerda de antemano quién es responsable de los hallazgos y cómo se les dará seguimiento, idealmente como registros de decisiones y elementos del backlog con responsables. Y repite: los hallazgos son una instantánea, y los riesgos cambian a medida que cambian el sistema y el negocio.

Los profesionales no se ponen de acuerdo sobre el lugar que ocupa hoy el método. Sus críticos lo ven como un artefacto pesado de los grandes proyectos con diseño previo, que encaja mal con equipos que despliegan muchas veces al día. Sus defensores responden que los conceptos (escenarios, puntos de sensibilidad y de compromiso, temas de riesgo) son justo lo que les falta a los equipos ágiles cuando se saltan la evaluación, y que el método se puede reducir. Ambos tienen parte de razón: la ceremonia completa conviene a sistemas de alto riesgo y larga vida y a compromisos importantes, mientras que el vocabulario y el hábito de preguntar a partir de escenarios son útiles en todas partes. La Sección 11.2 trata el extremo más ligero.

ATENCIÓN

Errores frecuentes en las evaluaciones de arquitectura

- Tratar la evaluación como un examen de aprobado o suspenso, de modo que el arquitecto se defiende en lugar de explorar y los riesgos se ocultan.
- Evaluar sin impulsores de negocio, lo que produce una lista de preocupaciones técnicas sin forma de clasificarlas.
- Dejar que el análisis se desvíe hacia el estilo del código, las preferencias de herramientas o los gustos personales en lugar de preguntas basadas en escenarios.
- Registrar los no riesgos sin sus supuestos, de modo que nadie se da cuenta cuando un supuesto deja de cumplirse.
- Producir una larga lista de riesgos y ningún tema de riesgo, lo que impide a los responsables ver las pocas causas de fondo.
- Hacer la evaluación después de firmar el contrato o de comprar la plataforma, cuando de los hallazgos solo cabe lamentarse.
- No dar seguimiento: los hallazgos que no se convierten en registros de decisiones, elementos del backlog y responsables se olvidan en un trimestre.

Preguntas frecuentes

¿Se puede aplicar el ATAM a un sistema que ya existe?
Sí. Los pasos son los mismos; la arquitectura se reconstruye a partir de la documentación, el código y las personas que operan el sistema, y las mediciones del sistema en producción sustituyen a las estimaciones. Evaluar un sistema existente es habitual antes de una ampliación importante, una migración o una adquisición.

¿Un ATAM sin hallazgos graves significa que el sistema cumplirá sus requisitos?
No. El ATAM examina una muestra de escenarios al nivel de las decisiones de diseño. Reduce la probabilidad de sorpresas desagradables, pero no puede sustituir a la medición, las pruebas y la experiencia operativa. Sus hallazgos deberían convertirse en pruebas y funciones de aptitud siempre que sea posible.

¿Quién debería presentar la arquitectura?
El arquitecto o el ingeniero principal que mejor la conoce. El trabajo del equipo de evaluación es hacer preguntas, no presentar, y el trabajo del arquitecto es explicar y explorar, no defenderse.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Qué diferencia hay entre un punto de sensibilidad y un punto de compromiso?**

Un punto de sensibilidad es una propiedad crítica para la respuesta de una cualidad. Un punto de compromiso es una propiedad que es un punto de sensibilidad para varias cualidades a la vez, de modo que cambiarla favorece a una y perjudica a otra.

▼ **¿Por qué registra el ATAM los no riesgos y qué debe registrarse junto con ellos?**

Para mostrar qué decisiones se examinaron y se consideraron sólidas. También deben registrarse los supuestos de los que dependen, para poder volver a examinar el no riesgo si un supuesto cambia.

▼ **¿Qué significa que las partes interesadas voten mucho en la fase 2 un escenario que falta en el árbol de utilidad?**

Que los responsables de las decisiones tienen un punto ciego sobre lo que importa, y puede que la arquitectura no responda a ese escenario. Se añade al árbol y se analiza en el paso 8.

▼ **En el CBAM, ¿cómo se calcula el valor por costo de una estrategia?**

Su beneficio, la utilidad ponderada por prioridad que añade en los escenarios a los que afecta, se divide por su costo.

▼ **Inténtalo: Clasifica estos hallazgos de ATAM: el throughput de pedidos depende directamente del número de particiones del broker; el cifrado de un enlace mejora la seguridad pero añade latencia; un solo proveedor de pagos, sin alternativa, en la ruta de pedidos; una zona caída se gestiona con un failover probado.**

El número de particiones es un punto de sensibilidad: una propiedad afecta fuertemente a un atributo de calidad. El cifrado es un punto de compromiso, porque la misma decisión ayuda a la seguridad y perjudica al rendimiento. El único proveedor de pagos es un riesgo, que amenaza el impulsor de negocio de la disponibilidad de los pedidos. La conmutación por error probada es un no riesgo, bajo supuestos declarados como disponer de capacidad suficiente en las otras zonas. Agrupa los riesgos en temas y convierte cada hallazgo en un registro de decisión o un elemento del backlog con un responsable.

FUENTES

- SEI: ATAM: Method for Architecture Evaluation (CMU/SEI-2000-TR-004): https://www.sei.cmu.edu/library/atam-method-for-architecture-evaluation/
- SEI: Making Architecture Design Decisions: An Economic Approach (the CBAM): https://www.sei.cmu.edu/library/making-architecture-design-decisions-an-economic-approach

SECCIÓN 11.2 DE 11.8 · AVANZADO · 27 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Revisiones ligeras y risk-storming

ANTES DE LEER ESTO

Esta sección se basa en:

- 4.3 El modelo C4
- 4.8 Comunicar
- 11.1 Evaluación con ATAM

Términos anteriores que utiliza: Riesgo (en la evaluación de arquitectura), Modelo C4, Diagrama de contexto del sistema, Diagrama de contenedores, Registro de decisiones de arquitectura (ADR)

LA IDEA EN BREVE

Antes de una excursión larga, un grupo sensato extiende el mapa sobre la mesa y cada persona, en silencio y por su cuenta, marca los lugares que le preocupan: el cruce del río, la pedrera empinada, el tramo sin cobertura de teléfono. Después comparan. Los lugares que marcan muchas personas reciben un plan; un lugar que marca una sola persona recibe una pregunta, porque esa persona puede saber algo que los demás no saben. Las revisiones ligeras de arquitectura funcionan así. En lugar de que un comité lejano apruebe un plan terminado, las personas que conocen el sistema miran juntas sus diagramas, pronto y a menudo, y acuerdan qué riesgos abordar primero.

LO QUE NECESITAS SABER

- Las juntas de revisión que se reúnen en momentos fijos suelen ver los diseños demasiado tarde, carecen de contexto, invitan a presentaciones a la defensiva y añaden tiempo de espera en cola; la revisión independiente funciona mejor cuando es frecuente, temprana, colaborativa y centrada en los riesgos.
- El risk-storming (Simon Brown) tiene cuatro pasos: dibujar diagramas, identificar los riesgos de forma individual y en silencio, hacerlos converger sobre los diagramas, y después revisar y resumir, puntuando la probabilidad por el impacto de 1 a 9.
- Un pre-mortem (Gary Klein, Harvard Business Review, 2007) pide a las personas que imaginen que el proyecto ya ha fracasado y que anoten por qué, lo que saca a la luz dudas sobre los planes, las competencias y los plazos que las revisiones de diagramas pasan por alto.

- Las katas de arquitectura (Ted Neward) desarrollan las habilidades de diseño y de revisión, mientras que TARA (Eoin Woods) revisa sistemas existentes combinando el juicio experto con el análisis automatizado del código y de los datos en tiempo de ejecución.
- Las revisiones de diseño y de RFC por escrito funcionan mejor con una decisión explícita, un plazo para los comentarios, revisores obligatorios y comentarios etiquetados como preguntas, sugerencias o inquietudes bloqueantes.
- Una buena revisión produce un registro de riesgos con responsables, fechas, riesgos aceptados y no riesgos con sus supuestos, publicado en pocos días y seguido en el backlog habitual.

Por qué fracasan a menudo las juntas de revisión pesadas

Muchas organizaciones gobiernan la arquitectura mediante una junta de revisión de arquitectura: un panel de arquitectos sénior que se reúne cada mes, recibe una presentación de diapositivas o un documento largo, y aprueba o rechaza diseños. La intención es buena: ojos independientes y con experiencia sobre las decisiones significativas. Los resultados suelen ser pobres, por razones estructurales más que personales.

El momento es el primer problema. Una junta que se reúne en momentos fijos ve un diseño o demasiado pronto (todavía no hay nada decidido) o, mucho más a menudo, demasiado tarde, cuando el equipo ya ha construido un prototipo, se ha comprometido con un proveedor y ha prometido una fecha. Entonces el rechazo es tan costoso que la junta aprueba con condiciones a las que nadie da seguimiento. El contexto es el segundo: los revisores que dedican una hora a un sistema que nunca han visto hacen preguntas genéricas y pasan por alto los riesgos concretos que los propios desarrolladores, el personal de operaciones y los ingenieros de seguridad del equipo podrían nombrar en minutos. Los incentivos son el tercero: cuando una revisión es un examen, los equipos presentan el diseño que quieren que se apruebe, no las dudas que tienen, y la junta aprende poco. La cola es el cuarto: una reunión mensual con el orden del día lleno añade semanas de espera a cada decisión significativa, lo que empuja a los equipos a esquivar la junta etiquetando las decisiones como menores.

Nada de esto significa que la revisión independiente no valga nada. Los sectores regulados (banca, sanidad, aviación) suelen necesitar una aprobación documentada, y algunas decisiones realmente necesitan la autoridad de un grupo sénior. La lección es que la revisión debe ser *frecuente, temprana, colaborativa y centrada en los riesgos*, reservando la aprobación formal para las pocas decisiones que de verdad la requieren. La Sección 11.4 vuelve sobre cómo puede funcionar la gobernanza sin compuertas lentas.

Lo que te aporta una buena revisión ligera

Una revisión ligera es una sesión breve y estructurada, normalmente de una o dos horas, en la que las personas que mejor conocen un diseño y unas pocas que lo ven por primera vez buscan riesgos juntas. Persigue cuatro cosas: descubrir los riesgos mientras todavía es barato abordarlos, construir una comprensión compartida del diseño en todo el equipo, ofrecer un cuestionamiento independiente de los supuestos del arquitecto y dejar constancia de los riesgos aceptados, mitigados o descartados. Su resultado es una lista de riesgos priorizada con responsables, no un veredicto.

Varias técnicas encajan en esta descripción. Se diferencian en lo que examinan (un diagrama, un plan, un documento o un sistema en funcionamiento) y en aquello para lo que mejor sirven, y se combinan bien.

Comparación de técnicas de revisión ligera

Técnica	Examina	Más adecuada para	Tiempo habitual
Risk-storming (Simon Brown)	Diagramas de arquitectura	Encontrar riesgos técnicos rápidamente con todo el equipo	De 45 a 90 minutos
Pre-mortem (Gary Klein)	Un plan o una decisión	Sacar a la luz dudas sobre todo el proyecto, incluidas las personas y los plazos	De 30 a 60 minutos
Kata de arquitectura (Ted Neward)	Un enunciado breve, inventado o real	Desarrollar la habilidad de diseño y explorar alternativas	De 1 a 3 horas
TARA (Eoin Woods)	Un sistema existente: personas, documentos, código y datos en tiempo de ejecución	Evaluar la deriva y los riesgos de un sistema en funcionamiento	Unos días de trabajo de expertos
Revisión de diseño o de RFC	Una propuesta por escrito	Comentarios asíncronos e inclusivos antes de construir	Un plazo de comentarios de varios días, más una reunión breve
Revisión con lista de comprobación	Cualquier diseño	Asegurarse de que se han tenido en cuenta los puntos ciegos habituales	Parte de cualquiera de las anteriores

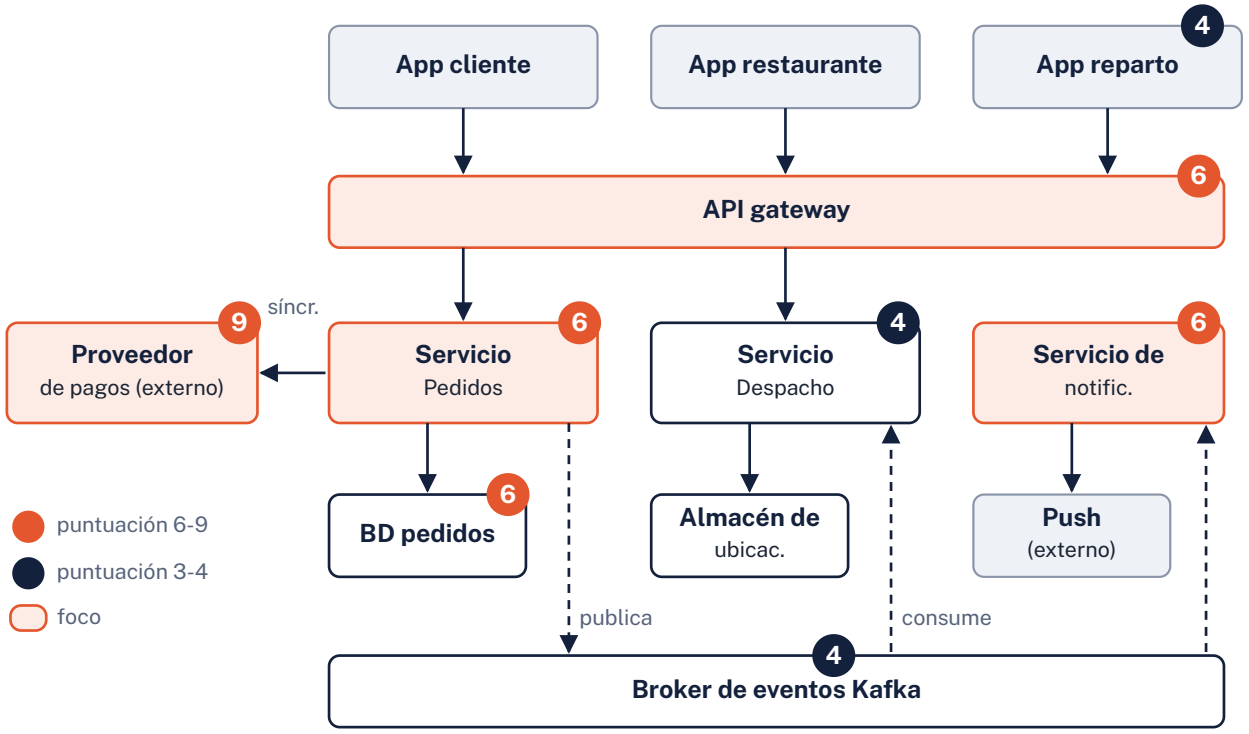
Risk-storming

El risk-storming es una técnica visual y colaborativa creada por Simon Brown, el autor del modelo C4 (Sección 4.3). Se basa en una observación sencilla: personas distintas ven riesgos distintos, y la mejor forma de recogerlos es poner a todos delante de los mismos diagramas y pedirles que miren por su cuenta antes de debatir. Tiene cuatro pasos.

1. Dibuja la arquitectura. Muestra diagramas de lo que piensas construir o cambiar, idealmente en más de un nivel: un diagrama de contexto del sistema y un diagrama de contenedores suelen bastar, con un diagrama de componentes para el contenedor más arriesgado. **2. Identifica los riesgos de forma individual.** Cada participante escribe en silencio un riesgo por nota adhesiva, con una descripción breve y una prioridad, durante unos diez minutos. El silencio importa: evita que la primera persona en hablar, o la más sénior, condicione a todas las demás, que es la principal forma en que los debates en grupo pierden información. **3. Haz converger los riesgos sobre los diagramas.** Cada persona coloca sus notas en la parte del diagrama a la que se refiere el riesgo. Los grupos aparecen de inmediato, y el diagrama se convierte en un mapa de calor. **4. Revisa y resume.** El grupo comenta los grupos de notas y después presta especial atención a dos tipos de nota: los riesgos que ha planteado una sola persona (a menudo alguien con conocimientos que les faltan a los demás, como el ingeniero de soporte) y los riesgos a los que distintas personas han dado prioridades muy diferentes (señal de un malentendido). Los riesgos acordados pasan a un registro de riesgos con responsables; también se registran los elementos que se acuerda que no son riesgos.

La prioridad suele puntuarse como probabilidad por impacto, cada uno valorado como bajo (1), medio (2) o alto (3), lo que da puntuaciones de 1 a 9. La guía de Brown las agrupa en alta (6 o 9), media (3 o 4) y baja (1 o 2), que por convención se muestran con notas rosas o rojas, amarillas o ámbar, y verdes. Una lista de sugerencias ayuda a las personas a pensar más allá de su especialidad: escalabilidad, disponibilidad, rendimiento, seguridad, consistencia de los datos, dependencias de terceros, costo, madurez de la tecnología, competencias del equipo, y entrega y operación.

Un mapa de calor de risk-storming sobre un diagrama de contenedores



Tras la convergencia, las puntuaciones se concentran en la ruta de pedidos y pagos, el gateway compartido y las notificaciones a los restaurantes; esos contenedores son los focos que hay que comentar primero.

EJEMPLO PRÁCTICO

Ejemplo práctico: risk-storming del diagrama de contenedores de la plataforma de reparto de comida

Seis personas (el arquitecto, dos desarrolladores, un ingeniero de operaciones, un ingeniero de seguridad y un ingeniero de soporte que atiende las quejas de los restaurantes) dedicaron 75 minutos al diagrama de contenedores anterior, con el diagrama de contexto del sistema al lado. Cada una escribió riesgos en silencio durante diez minutos; se colocaron 31 notas, que se fusionaron en 8 riesgos distintos.

Proveedor de pagos en la ruta síncrona del pedido (9)	Lo plantearon cuatro personas. Las ralentizaciones del proveedor en el pico bloquean la realización de pedidos. Acción: registro de decisión sobre la autorización asíncrona con un estado de pedido pendiente; responsable: el arquitecto; plazo: dos semanas
Notificaciones a los restaurantes sin confirmación de recepción (6)	Lo planteó solo el ingeniero de soporte, que ve quejas por pedidos perdidos cada viernes. Nadie más lo sabía. Acción: entrega con confirmación de recepción, reenvío y una alerta cuando una tableta deja de confirmar; responsable: el equipo de notificaciones
Configuración compartida del gateway (6)	Un solo cambio de ruta defectuoso puede tumbar las tres apps a la vez. Acción: configuración del gateway por app y despliegue canario de los cambios del gateway
Conmutación por error de la base de datos de pedidos nunca probada (6)	La réplica existe, pero la promoción nunca se ha ensayado. Acción: simulacro trimestral de conmutación por error, medido frente al escenario de recuperación
Competencias en Kafka (puntuado con 9 por una persona y con 2 por las demás)	El único ingeniero que había operado Kafka antes explicó el reequilibrio de particiones durante los despliegues, el retraso de los consumidores y los errores de retención. Tras el debate, el grupo acordó un 4: real pero manejable. Acción: servicio de Kafka gestionado y un runbook
Tráfico de ubicaciones de los repartidores a través del gateway (4)	Unas 2250 actualizaciones por segundo comparten el gateway con el tráfico de pedidos. Acción: medirlo en la próxima prueba de carga; endpoint de ingesta separado si la latencia de los pedidos se resiente
Retraso de los consumidores de despacho en el pico (4)	Acción: alertar del retraso de los consumidores; añadir una función de aptitud a la prueba de carga
El almacén de ubicaciones pierde datos al reiniciarse (2)	Se acordó que no es un riesgo: las posiciones se actualizan cada 4 segundos. Se registró así, con el supuesto

El hallazgo de mayor valor (las notificaciones a los restaurantes) vino de la persona con menos probabilidades de hablar primero en un debate abierto, y esa es precisamente la razón de identificar los riesgos en silencio y de forma individual.

El desacuerdo sobre Kafka se resolvió con conocimiento, no promediando las puntuaciones.

Cada riesgo acordado salió de la sala con un responsable y una acción; el registro se publicó el mismo día.

Pre-mortems

El risk-storming examina una estructura. Un pre-mortem examina un plan. Gary Klein lo describió en la *Harvard Business Review* en septiembre de 2007. Un post-mortem explica por qué murió un paciente, lo que ayuda a todos menos al paciente; un pre-mortem se hace antes de que empiece el proyecto, cuando las respuestas todavía pueden cambiar su rumbo.

El procedimiento es breve. El líder explica al grupo el plan o la decisión y después anuncia que ahora es dentro de unos meses y que el proyecto ha fracasado, y de mala manera. Cada persona dedica unos minutos a anotar, por su cuenta, todas las razones del fracaso que se le ocurran. Después, el líder recorre la sala recogiendo una razón cada vez hasta agotar las listas, y el responsable del plan las revisa en busca de formas de reforzarlo. Klein cita una investigación de Deborah Mitchell, Jay Russo y Nancy Pennington (1989) que concluyó que imaginar que un acontecimiento ya ha ocurrido (retrospectiva prospectiva) aumenta en torno a un 30% la capacidad de las personas para identificar las razones de resultados futuros. El enfoque también cambia la dinámica social: la pregunta ya no es si el plan podría fracasar, lo que suena desleal, sino por qué fracasó, lo que invita a la creatividad.

En arquitectura, los pre-mortems son especialmente útiles antes de los grandes compromisos: adoptar un estilo dirigido por eventos, migrar a una nube nueva, sustituir un sistema heredado (Sección 11.6) o introducir una plataforma. Entre los resultados habituales que una revisión basada en diagramas pasaría por alto están que el equipo carezca de competencias en la nueva tecnología, que la migración se estanque a medio camino con dos sistemas que operar, que un proveedor clave cambie sus precios y que el negocio pierda la paciencia antes de que lleguen los beneficios.

SI ESTO ES NUEVO PARA TI

Paso a paso: imaginar primero el fracaso

La idea que requiere más atención aquí es por qué funciona un pre-mortem: imaginar que un plan ya ha fracasado saca a la luz dudas que una simple conversación sobre "¿qué podría salir mal?" a menudo deja sin decir. Sigue un plan surgido de la sesión de risk-storming anterior.

- Supongamos que el equipo decide sacar la autorización de la tarjeta de la ruta síncrona del pedido y aceptar los pedidos en estado pendiente. Antes de empezar el trabajo, el equipo y personas de equipos vecinos se reúnen durante media hora.
- El líder explica el plan y después anuncia que han pasado seis meses y que el cambio ha fracasado de mala manera. Cada persona anota, sola y en silencio, todas las razones que se le ocurran.
- El enfoque hace el trabajo. Si se pregunta si el plan podría fracasar, quien tiene dudas parece desleal por decirlo. Si se pregunta por qué fracasó, cada duda se convierte en una respuesta útil.
- El líder recoge una razón cada vez hasta que se agotan las listas: nadie en el equipo de pedidos había construido antes un diseño de este tipo; el cambio se estancó a medio camino y dejó funcionando a la vez las rutas de pago antigua y nueva; el proveedor de pagos cambió sus precios; el negocio perdió la paciencia antes de que llegaran los beneficios.
- Pocas de estas razones aparecerían como notas adhesivas en el diagrama de contenedores: tienen más que ver con los plazos, los proveedores y la paciencia que con cajas y flechas. Después, el responsable del plan lo refuerza, por ejemplo organizando primero una formación y fijando una fecha para desactivar la ruta antigua.
- **La lección:** el risk-storming examina una estructura; un pre-mortem examina un plan. Imaginar que el fracaso ya ha ocurrido hace que las razones sean más fáciles de encontrar y más seguras de decir, así que haz uno antes de un gran compromiso, mientras las respuestas todavía pueden cambiar el plan.

Katas de arquitectura

Las katas arquitectónicas las creó Ted Neward, inspirándose en las katas de código de Dave Thomas, como ejercicios de práctica para arquitectos. Una kata es un enunciado breve sobre un sistema ficticio (unos pocos párrafos de requisitos, usuarios y restricciones, incompletos a propósito). Equipos pequeños de tres a cinco personas trabajan en ella durante un tiempo fijo, haciendo preguntas a un moderador que hace de cliente para descubrir los requisitos que

faltan; después esbozan una arquitectura y la presentan a los demás equipos, que la cuestionan con dureza pero con justicia antes de que la sala vote. Neal Ford mantiene una colección pública de katas, y O'Reilly ha organizado competiciones de katas arquitectónicas.

Las katas no son revisiones de un diseño real, pero contribuyen a la cultura de revisión de dos maneras. Desarrollan la habilidad de explicar y defender un diseño ante las preguntas, y la de hacer buenas preguntas sobre el diseño de otra persona, que es exactamente lo que necesitan las revisiones ligeras. Y una kata interna sobre un problema real que se avecina, con dos o tres equipos diseñando por separado, es una forma barata de explorar el espacio de opciones antes de comprometerse: cuando tres equipos producen tres diseños distintos, las diferencias muestran dónde están los compromisos reales.

TARA y revisiones basadas en listas de comprobación

Las revisiones de sistemas que ya existen se enfrentan a un problema distinto: la arquitectura tal como está construida puede diferir de la arquitectura tal como está descrita, y las pruebas están repartidas entre personas, documentos, código y datos de producción. Eoin Woods, coautor de *Software Systems Architecture*, describió en 2011 el Tiny Architectural Review Approach (TARA) como una alternativa industrial, deliberadamente ligera, a los métodos muy basados en escenarios. Un número reducido de revisores recaba los requisitos clave y el contexto del sistema, establece su estructura real, evalúa su implementación mediante análisis automatizado (dependencias entre módulos, tamaño, métricas de código, cobertura de pruebas) junto con información de tiempo de ejecución como los logs, usa el juicio experto para vincular lo que encuentra con los requisitos, y presenta los hallazgos y las recomendaciones en un formato predefinido. Su punto fuerte es la rapidez y la evidencia; su debilidad, que Woods reconoció, es que depende de la pericia de los revisores.

Las listas de comprobación complementan todas las técnicas. Una lista de comprobación no encuentra riesgos por sí sola, pero evita que una revisión olvide una categoría entera (la restauración de copias de seguridad, los secretos, el costo a escala) porque a nadie de la sala se le ocurrió pensar en ella. Mantén las listas de comprobación cortas, específicas del historial de incidentes de tu organización y formuladas como preguntas que exijan pruebas en lugar de respuestas de sí o no.

Revisiones Well-Architected

Los grandes proveedores de nube publican marcos que pueden servir como listas de comprobación de revisión ya preparadas. AWS y Google Cloud describen seis pilares cada uno y Azure cinco, que abarcan operaciones, seguridad, fiabilidad, rendimiento y costos, con la sostenibilidad añadida por los dos primeros. Una revisión recorre una carga de trabajo mediante preguntas estructuradas. La AWS Well-Architected Tool no tiene cargo adicional,

registra tus respuestas y señala los problemas de alto riesgo, y las lentes personalizadas pueden añadir tus propias reglas; Azure ofrece una Well-Architected Review en línea. Para la carga de trabajo de pedidos de la plataforma de reparto de comida, una sesión de media jornada da una cobertura amplia y una lista de riesgos que corregir o aceptar. Los límites son reales: las preguntas expresan la visión de un proveedor sobre las buenas prácticas, y no sopesan entre sí tus propios escenarios de calidad como hace la Sección 11.1. Comprueba la versión vigente, porque los pilares y las preguntas cambian.

Documentos de diseño y revisiones de RFC

La Sección 4.8 describió los documentos de diseño y los RFC como una forma de proponer un cambio por escrito antes de construirlo. Revisarlos bien es una habilidad en sí misma. Una revisión por escrito tiene ventajas reales sobre una reunión: los revisores comentan cuando tienen tiempo para pensar, los compañeros en remoto y los más callados contribuyen en igualdad de condiciones, y los comentarios quedan como registro. Sus fallos típicos son documentos que nadie lee, hilos de comentarios que nunca se cierran y revisiones que se convierten en una compuerta extraoficial.

Una buena práctica es indicar al principio la decisión que se busca y el plazo para los comentarios, nombrar a los revisores cuya opinión es obligatoria (seguridad, operaciones, los responsables de los servicios afectados) e invitar a cualquier otra persona; pedir a los revisores que etiqueten cada comentario como pregunta, sugerencia o inquietud bloqueante, para que el autor sepa cuáles debe resolver; revisar frente a los escenarios de calidad y la lista de comprobación de más abajo, no frente a los gustos personales; cerrar con una reunión breve solo para las inquietudes sin resolver; y registrar las decisiones significativas como ADR. En las organizaciones que usan el proceso de consejo (Sección 11.4), el RFC es el vehículo para pedir consejo, y la decisión sigue siendo de su autor.

Cómo dirigir una revisión de arquitectura de 90 minutos, paso a paso

1. **Antes: preparar (unos días antes)**
   El arquitecto distribuye un resumen de una página, los diagramas de contexto y de contenedores, los tres a cinco escenarios de calidad principales con sus cifras y las preguntas abiertas. Invita a entre cinco y ocho personas: el equipo, operaciones, seguridad, alguien de cada equipo vecino y una persona externa con experiencia. Elige a un facilitador que no sea el arquitecto.

2. **Minutos 0 a 10: objetivos y reglas**
   El facilitador expone el propósito (encontrar riesgos, no aprobar ni rechazar), el tiempo asignado y el resultado esperado. Un representante del negocio o el responsable de producto repasa los impulsores de negocio en dos minutos.

3. **Minutos 10 a 25: recorrer la arquitectura con escenarios**
   El arquitecto recorre dos o tres de los escenarios más importantes sobre los diagramas, por ejemplo un pedido en el pico y una caída del proveedor de pagos. Solo preguntas de aclaración.

4. **Minutos 25 a 35: identificación de riesgos en silencio**
   Cada persona escribe riesgos en notas, uno por nota, con una probabilidad y un impacto. Sin debate.

5. **Minutos 35 a 50: converger y agrupar**
   Las notas se colocan en los diagramas. El facilitador fusiona en voz alta las duplicadas y destaca los grupos, las notas aisladas y los desacuerdos de prioridad.

6. **Minutos 50 a 75: debatir los riesgos principales**
   Empieza por los grupos con mayor puntuación. Para cada uno, acuerda con el grupo qué es realmente el riesgo, su puntuación y si es un riesgo, un no riesgo con un supuesto o una pregunta que necesita un spike.

7. **Minutos 75 a 85: acciones y responsables**
   Cada riesgo alto o medio acordado recibe una mitigación, un responsable y una fecha. Se enumeran las decisiones que deben registrarse como ADR.

8. **Minutos 85 a 90: cierre**
   Repasa en voz alta los riesgos, los riesgos aceptados y los no riesgos con sus supuestos. Acuerda cuándo se volverá a examinar el diseño.

9. **Después: publicar en un plazo de dos días**
   Publica el registro de riesgos, las fotografías de los diagramas anotados y la lista de acciones donde el equipo guarda su documentación de arquitectura, y haz el seguimiento de las acciones en el backlog habitual.

GUÍA DE DECISIÓN

Guía de decisión: ¿qué tipo de revisión?

Empieza por arriba; las técnicas se pueden combinar.

Q1 **¿Es inminente un compromiso caro y difícil de revertir, como un contrato de varios años, la compra de una plataforma o una compuerta de financiación, con muchas partes interesadas?**

- **Sí** Usa un ATAM o un ATAM reducido con evaluadores independientes (Sección 11.1), y añade un pre-mortem sobre el plan.
- **No** Continúa.

Q2 **¿Hay una propuesta por escrito para un cambio significativo?**

- **Sí** Haz una revisión de diseño o de RFC por escrito con un plazo para los comentarios y, después, una reunión breve para las inquietudes bloqueantes.
- **No** Continúa.

Q3 **¿Existen diagramas de la arquitectura propuesta o existente?**

- **Sí** Aplica risk-storming sobre ellos, con el formato de 90 minutos.
- **No** Dibuja primero un diagrama de contexto y uno de contenedores; dibujarlos a menudo saca a la luz riesgos por sí solo.

Q4 **¿La principal preocupación es el éxito de todo el proyecto (competencias, plazos, migración, adopción) más que la estructura?**

- **Sí** Añade un pre-mortem.
- **No** Continúa.

Q5 **¿El sistema ya está en funcionamiento, con sospechas de deriva o de deuda creciente?**

- **Sí** Usa una revisión al estilo de TARA que combine entrevistas con análisis automatizado del código y del tiempo de ejecución, y convierte los hallazgos en funciones de aptitud.
- **No** Confía en las funciones de aptitud entre revisiones y vuelve a revisar en la siguiente decisión significativa.

Lista de comprobación: preguntas para cualquier revisión ligera de arquitectura

Pide pruebas, no tranquilidad. Adapta la lista al historial de incidentes de tu propia organización.

- [ ] **¿Qué tres a cinco escenarios de calidad importan más, y cuáles son sus cifras?**
  Sin impulsores medibles, la revisión no puede clasificar los riesgos (Sección 2.3).
- [ ] **¿Qué hay en la ruta síncrona del recorrido más importante?**
  Cuenta los saltos, enumera los timeouts y los reintentos, e identifica a cada tercero que haya en la ruta.
- [ ] **¿Qué ocurre cuando cada dependencia está lenta o caída?**
  Lento suele ser peor que caído; pregunta por ambos casos y por cómo se informa a los usuarios.
- [ ] **¿Quién es el propietario de cada dato importante, y se ha probado una restauración?**
  Escritor único, expectativas de consistencia, retención y pruebas de una restauración reciente con éxito (Capítulo 7).
- [ ] **¿Dónde están los límites de confianza, las identidades y los secretos?**
  Busca un modelo de amenazas actualizado y secretos en el código o en la configuración (Capítulo 8).
- [ ] **¿Qué se rompe primero con el doble del pico esperado?**
  Pide el modelo de capacidad o los resultados de las pruebas de carga, no la intuición.
- [ ] **¿Cómo se observa, se despliega y se revierte el sistema?**
  Paneles, alertas ligadas a los SLO, estrategia de despliegue y el tiempo que tarda un rollback.
- [ ] **¿Qué es probable que cambie el próximo año, y a qué afectaría ese cambio?**
  Comprueba que los cambios probables se quedan dentro de un solo módulo o servicio.
- [ ] **¿Cuánto costará operarlo con el volumen esperado?**
  Un costo por pedido, por usuario o por transacción, con sus principales impulsores.
- [ ] **¿Están registradas las decisiones significativas con sus alternativas y consecuencias?**
  ADR o un documento de diseño, incluidos los compromisos aceptados.

- [ ] **¿La estructura del equipo se corresponde con el diseño?**
  Cada parte tiene un equipo propietario con las competencias para operarla (Sección 3.7 y Sección 11.5).

ATENCIÓN

Cómo fallan las revisiones ligeras

- Debate abierto antes de la identificación en silencio, de modo que la primera persona en hablar condiciona al grupo.
- Eliminar los desacuerdos promediándolos en lugar de preguntar por qué una persona puntuó un riesgo de forma tan distinta.
- Revisar la notación del diagrama en lugar de los riesgos de la arquitectura.
- Sin responsables ni fechas, de modo que el registro de riesgos se convierte en una lista de buenas intenciones.
- El arquitecto como facilitador, lo que dificulta recibir críticas y dirigir la sesión al mismo tiempo.
- Hacer revisiones solo cuando algo ya está en apuros; las revisiones periódicas y rutinarias son más baratas y menos defensivas.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué el risk-storming pide a las personas que identifiquen los riesgos en silencio antes de cualquier debate?

Para evitar que la primera persona en hablar, o la más sénior, condicione al grupo, de modo que el conocimiento independiente de cada persona llegue al diagrama.

▼ En el risk-storming, ¿qué dos tipos de nota merecen especial atención durante el paso de revisión?

Los riesgos que plantea una sola persona, que puede saber algo que los demás no saben, y los riesgos a los que se dan prioridades muy distintas, que indican un malentendido.

▼ ¿Qué examina un pre-mortem que normalmente no examina una sesión de risk-storming sobre un diagrama de contenedores?

El plan y su contexto en conjunto: competencias, plazos, migración, proveedores y paciencia de la organización, más que la estructura técnica.

▼ **Nombra dos razones estructurales por las que las juntas de revisión de arquitectura mensuales suelen fracasar.**

Dos cualesquiera de estas: ven los diseños demasiado tarde, los revisores carecen de contexto, los equipos se presentan a la defensiva y la cola añade semanas de retraso que fomentan que se evite la junta.

▼ **Inténtalo: Un risk-storming valora (probabilidad, impacto): caída del proveedor de pagos (3, 3); mala configuración del broker (2, 3); índice de búsqueda desactualizado (3, 1); incendio en el centro de datos (1, 3). Puntúalos y ordénalos. ¿Por qué las notas se escribieron primero en silencio?**

Proveedor de pagos 9, alto. Broker 6, alto. Índice de búsqueda 3 e incendio 3, ambos medios, aunque el incendio merece una línea que indique sus supuestos, porque tiene un impacto alto. Aborda primero los dos principales, con responsables y fechas en el backlog normal. La escritura en silencio evita que la primera voz o la más veterana dirija al grupo. Cuando cuatro personas plantean de forma independiente el mismo riesgo, eso es una evidencia útil, y los desacuerdos muestran supuestos distintos.

FUENTES

- Simon Brown: Risk-storming: https://riskstorming.com/
- Gary Klein: Performing a Project Premortem, Harvard Business Review (2007): https://hbr.org/2007/09/performing-a-project-premortem
- Neal Ford: Architectural Katas: https://nealford.com/katas/about.html
- Experiences with ALMA: Architecture-Level Modifiability Analysis: https://research.vu.nl/en/publications/experiences-with-alma-architecture-level-modifiability-analysis/
- The pillars of the AWS Well-Architected Framework: https://docs.aws.amazon.com/wellarchitected/latest/framework/the-pillars-of-the-framework.html
- What is AWS Well-Architected Tool?: https://docs.aws.amazon.com/wellarchitected/latest/userguide/intro.html
- Workloads (AWS Well-Architected Tool): https://docs.aws.amazon.com/wellarchitected/latest/userguide/workloads.html
- AWS Well-Architected Tool pricing: https://aws.amazon.com/well-architected-tool/pricing/
- Azure Well-Architected Framework pillars (Microsoft): https://learn.microsoft.com/en-us/azure/well-architected/pillars
- Google Cloud Well-Architected Framework: https://docs.cloud.google.com/architecture/framework

SECCIÓN 11.3 DE 11.8 · AVANZADO · 27 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Funciones de aptitud y arquitectura evolutiva

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.3 Escenarios de calidad
- 3.8 Simplicidad y medición
- 10.5 CI/CD y despliegue

Términos anteriores que utiliza: Escenario de atributo de calidad, Medida de la respuesta, Registro de decisiones de arquitectura (ADR), Árbol de utilidad, Último momento responsable

LA IDEA EN BREVE

El tablero de un automóvil tiene luces de advertencia para las cosas que nunca deben fallar sin que nadie lo note: la presión del aceite, la temperatura del motor, los frenos. No esperas a la revisión anual para descubrir que el motor se sobrecalentó en marzo. Una función de aptitud (fitness function) es una luz de advertencia para una arquitectura de software. El equipo decide qué cualidades importan de verdad, como la rapidez con que se confirma un pedido o qué partes del código pueden comunicarse entre sí, escribe una comprobación automática para cada una y ejecuta esas comprobaciones cada vez que cambia el software. Cuando una comprobación falla, el cambio se detiene o se avisa a alguien, mientras el problema todavía es pequeño.

LO QUE NECESITAS SABER

- Una función de aptitud arquitectónica es cualquier mecanismo que ofrece una evaluación objetiva de una característica de la arquitectura, idealmente automatizada y vinculada a un escenario de atributo de calidad cuya medida de la respuesta se convierte en su umbral.
- Las funciones de aptitud se clasifican en atómicas u holísticas; activadas, continuas o temporales; estáticas o dinámicas; automatizadas o manuales; e intencionales o emergentes; un conjunto sano las combina.
- Ejemplos típicos son las reglas de dependencia para la estructura, las pruebas de carga y los presupuestos de latencia para el rendimiento, los análisis de vulnerabilidades para la seguridad, los experimentos de caos para la resiliencia y las comprobaciones de compatibilidad de esquemas para los datos.

- La arquitectura evolutiva (Ford, Parsons, Kua y Sadalage) admite un cambio guiado e incremental en múltiples dimensiones; las funciones de aptitud aportan la guía, y las unidades desplegables pequeñas y los pipelines aportan los incrementos.
- Cada función de aptitud necesita un responsable, una justificación registrada y un umbral, y un fallo obliga a decidir: corregir el código o cambiar la regla y registrar por qué.
- Empieza con unas pocas comprobaciones fiables para las características de mayor prioridad, usa líneas base para ajustar progresivamente los sistemas existentes y deja las comprobaciones holísticas lentas fuera de cada compilación.

De la evaluación puntual a la comprobación continua

Una evaluación como el ATAM (Sección 11.1) es una instantánea. Al día siguiente de terminar, los desarrolladores empiezan a cambiar el sistema, y cada cambio puede deshacer en silencio una decisión en la que se apoyaba la evaluación: una importación cómoda a través del límite de un módulo, una llamada síncrona adicional en la ruta del pedido, una biblioteca con una vulnerabilidad conocida. La Sección 1.7 describió el resultado: la erosión, cuando se viola la arquitectura, y la deriva, cuando se ignora. Las revisiones periódicas detectan parte de esto, con meses de retraso. La idea de esta sección es comprobar las decisiones arquitectónicas más importantes de forma automática y continua, del mismo modo que las pruebas unitarias comprueban el comportamiento.

El término procede del libro de Neal Ford, Rebecca Parsons y Patrick Kua *Building Evolutionary Architectures* (O'Reilly, 2017), cuya segunda edición, con Pramod Sadalage, apareció en noviembre de 2022. Lo tomaron de la computación evolutiva, donde una función de aptitud puntúa lo cerca que está una solución candidata del objetivo para que el algoritmo pueda quedarse con las mejores variantes. En arquitectura, el objetivo es el conjunto de características que el sistema debe conservar mientras todo lo demás cambia.

Qué es una función de aptitud

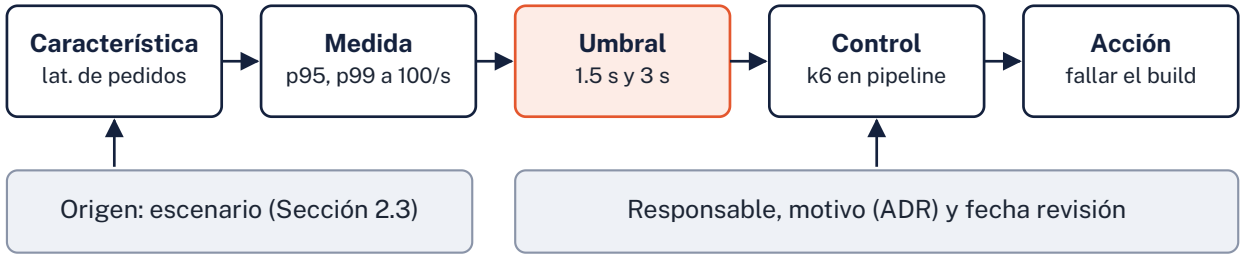
La segunda edición define una función de aptitud (fitness function) arquitectónica como cualquier mecanismo que proporciona una evaluación objetiva de la integridad de alguna característica o características de la arquitectura. Tres expresiones de esa definición hacen el trabajo. *Cualquier mecanismo*: una prueba unitaria, una regla de análisis estático, una prueba de carga, un monitor, un experimento de caos o incluso una revisión manual programada cumplen los requisitos. *Objetiva*: el resultado no debe depender de quién la ejecute; "el código parece bien estructurado" no es una función de aptitud, "ningún paquete de la capa de dominio importa el framework web" sí lo es. *Característica de la arquitectura*: la comprobación protege

una cualidad que la arquitectura existe para ofrecer (rendimiento, disponibilidad, modificabilidad, seguridad, compatibilidad de datos), no una regla de negocio, que es tarea de las pruebas ordinarias.

En la práctica, una función de aptitud útil tiene algunas propiedades más. Está vinculada a un requisito, idealmente un escenario de atributo de calidad cuya medida de la respuesta se convierte en su umbral (Sección 2.3). Tiene un responsable que decide qué ocurre cuando falla. Falla de forma ruidosa, rompiendo una compilación o avisando a alguien de guardia, en lugar de alimentar un panel que nadie lee. Y su umbral y el motivo de este quedan registrados, normalmente en un ADR, para que un equipo posterior pueda distinguir entre una regla que protege algo importante y un número arbitrario.

La Sección 3.8 trazó la distinción importante con las métricas: una métrica describe, una función de aptitud decide. Una métrica de acoplamiento en un panel te dice que las cosas empeoran; una función de aptitud codifica una decisión que el equipo ya ha tomado, de modo que incumplirla obliga a una elección consciente: corregir el código, o cambiar la regla y registrar por qué.

Anatomía de una función de aptitud



Una función de aptitud convierte la medida de la respuesta de un escenario en un umbral, un control automatizado y una acción, con un responsable y un motivo registrado.

Categorías de funciones de aptitud

Ford y sus coautores clasifican las funciones de aptitud según varias dimensiones independientes. La clasificación es útil menos como taxonomía que como estímulo para el diseño: preguntarse "¿debería ser atómica u holística, activada o continua?" a menudo revela una comprobación mejor que la primera que vino a la mente.

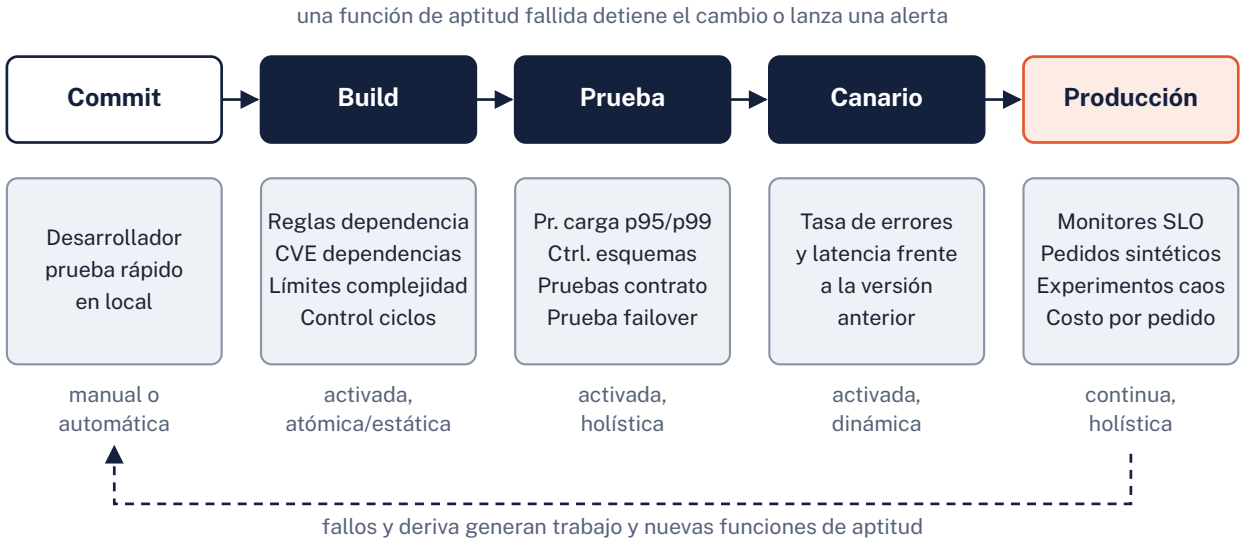
Dimensiones de las funciones de aptitud

Dimensión	Opciones	Qué significa	Ejemplo
Alcance	Atómica u holística	La atómica comprueba una característica en un contexto; la holística comprueba varias características que interactúan en un contexto compartido	Atómica: ningún ciclo entre paquetes. Holística: latencia de los pedidos medida con el cifrado, el registro de auditoría y la caché activados a la vez
Cuándo se ejecuta	Activada, continua o temporal	La activada se ejecuta ante un evento como un commit, una compilación o un despliegue; la continua se ejecuta todo el tiempo contra el sistema en funcionamiento; la temporal se dispara al pasar un tiempo, como recordatorio para revisar algo	Activada: análisis de dependencias en cada compilación. Continua: alerta de tasa de consumo del SLO en producción. Temporal: hacer fallar la compilación cuando una versión fijada de un framework tiene más de 12 meses
Tipo de resultado	Estática o dinámica	La estática compara con un umbral fijo; la dinámica ajusta el umbral al contexto, por ejemplo a la escala	Estática: p95 por debajo de 1.5 s. Dinámica: el p95 puede subir a 2 s por encima de 150 pedidos/s, según lo acordado para los picos extremos
Cómo se ejecuta	Automatizada o manual	Se prefiere la automatizada; algunas características, como los requisitos legales o de usabilidad, siguen necesitando personas, idealmente con un calendario fijo	Manual: revisión trimestral de accesibilidad según WCAG 2.2 AA
Planificación	Intencional o emergente	Las intencionales se diseñan de antemano a partir de requisitos conocidos; las emergentes se añaden a medida que el equipo aprende qué importa	Emergente: una comprobación del retraso de los consumidores (consumer lag) añadida tras un incidente

Las comprobaciones holísticas merecen atención porque las cualidades interactúan: los controles de seguridad añaden latencia, la caché complica la frescura de los datos, y una prueba atómica de cada una por separado puede pasar mientras la combinación falla. Las comprobaciones continuas importan porque algunas características solo existen en producción: la latencia real de los usuarios, los modos de fallo reales, el costo real. Una cartera

sana combina comprobaciones rápidas, atómicas y activadas en la compilación con un número menor de comprobaciones holísticas más lentas antes del lanzamiento y comprobaciones continuas en producción.

Dónde se ejecutan las funciones de aptitud



Los controles estructurales baratos se ejecutan en cada compilación, los holísticos antes del lanzamiento y los continuos en producción, y los fallos vuelven como trabajo nuevo.

SI ESTO ES NUEVO PARA TI

Paso a paso: cuando todas las comprobaciones pequeñas pasan

El punto en el que más gente tropieza aquí es que unas comprobaciones que pasan cada una por su cuenta pueden no detectar un fallo que solo aparece cuando todo se ejecuta a la vez, y por eso un equipo necesita funciones de aptitud tanto atómicas como holísticas. Sigue el escenario de latencia de los pedidos a través de ambas.

- El escenario de la plataforma de reparto de comida dice que, en el pico del viernes de 100 pedidos por segundo, el 95% de las confirmaciones de pedido deben devolverse en menos de 1.5 segundos.
- Supón que, a lo largo de unos meses, el equipo activa el cifrado y el registro de auditoría en la ruta del pedido. Cada cambio tiene su propia comprobación atómica, que confirma que funciona y que, medido por separado, solo añade un poco de tiempo. Todas las comprobaciones pasan.
- Pero el tiempo se acumula. Con ambos activados en la carga máxima, sus costos se suman a todo lo demás que hay en la ruta y, por ejemplo, el 6% de las confirmaciones tardan ahora más de 1.5 segundos, más que el 5% que permite el escenario. Ninguna comprobación estaba mal; ninguna miraba el conjunto.
- Una comprobación holística lo detecta: una prueba de carga que envía 100 pedidos por segundo a un entorno similar al de producción con el cifrado, el registro de auditoría y la caché activados a la vez, y que falla si el percentil 95 supera los 1.5 segundos.
- Las comprobaciones holísticas tienen costos. Una prueba de carga de diez minutos es demasiado lenta para cada compilación, así que se ejecuta cada noche y antes de cada lanzamiento. Y un fallo dice que el conjunto es demasiado lento, no qué parte lo causó; el ejemplo práctico más adelante en esta sección muestra una forma de resolverlo.
- **La lección:** las comprobaciones atómicas rápidas en cada compilación detectan la mayoría de los errores con poco costo, pero no pueden ver cómo interactúan las cualidades. Añade menos comprobaciones holísticas, más lentas, antes del lanzamiento, y comprobaciones continuas en producción, donde se puede medir la experiencia de los usuarios reales.

Funciones de aptitud para cada cualidad

La mayoría de los equipos empiezan por la estructura, porque las comprobaciones estructurales son baratas, rápidas y deterministas, y después añaden comprobaciones de rendimiento, seguridad, resiliencia y datos a medida que madura el pipeline. La tabla ofrece ejemplos típicos; las herramientas cambian, los patrones no.

Ejemplos por cualidad

Cualidad	Función de aptitud (fitness function)	Clase	Herramientas típicas
Estructura	Reglas de capas y de dependencias, ningún ciclo, los módulos se usan entre sí solo a través de API públicas (Sección 3.4)	Atómica, activada, estática	ArchUnit, import-linter, dependency-cruiser, verificación de Spring Modulith
Rendimiento	Prueba de carga de extremo a extremo con umbrales de percentiles; presupuestos de latencia por servicio en el pipeline propio de cada servicio	Holística, activada	k6, Gatling, JMeter, Locust; umbrales que hacen fallar el trabajo del pipeline
Seguridad	Hacer fallar la compilación si hay dependencias con vulnerabilidades conocidas altas o críticas; ningún secreto en el código; imágenes de contenedor solo del registro aprobado	Atómica, activada	pip-audit, npm audit, OWASP Dependency-Check, Trivy, escáneres de secretos; política como código (Sección 11.4)
Resiliencia	Experimentos de caos que eliminan instancias o ralentizan dependencias y comprueban que se mantiene el estado estable (Sección 9.8)	Holística, activada o continua	Toxiproxy, Chaos Toolkit, servicios de inyección de fallos de la nube
Datos	Comprobaciones de compatibilidad de esquemas en eventos y API; migraciones ensayadas con datos del tamaño de producción (Sección 7.7)	Atómica, activada	Comprobaciones de compatibilidad del registro de esquemas, buf breaking, pruebas de contrato dirigidas por el consumidor
Operabilidad y costo	Todos los servicios exponen comprobaciones de estado (health checks) y métricas estándar; el costo por pedido se mantiene por debajo de un objetivo	Atómica o continua	Comprobaciones en el pipeline sobre las plantillas de servicio; paneles de costos con alertas

La arquitectura evolutiva y el cambio incremental

Las funciones de aptitud son una mitad de lo que Ford, Parsons, Kua y Sadalage llaman arquitectura evolutiva: una arquitectura que admite un cambio guiado e incremental en múltiples dimensiones. Cada parte de la frase es un requisito de diseño.

El *cambio incremental* tiene una vertiente de desarrollo y otra operativa. Los desarrolladores deben poder hacer cambios pequeños con seguridad, lo que depende de la modularidad y de un acoplamiento adecuado (Capítulo 3) y de mantener cada parte desplegable de forma independiente, el cuanto de arquitectura de la Sección 5.1, lo bastante pequeña para cambiarla sin coordinarse con otros. Las operaciones deben poder lanzar cambios pequeños con frecuencia y con seguridad, lo que depende de un pipeline de despliegue, un despliegue progresivo y un rollback rápido (Sección 10.5). *Guiado* es donde entran las funciones de aptitud: definen qué significa "apto" e impiden que los incrementos lo degraden. *Múltiples dimensiones* recuerda que la arquitectura no es solo la estructura del código: los datos, la seguridad, las operaciones, la estructura de los equipos y el costo evolucionan, y todos necesitan protección.

El enfoque cambia la forma en que los arquitectos piensan en el futuro. En lugar de predecir cada requisito y construir de antemano flexibilidad para él, algo que la Sección 3.8 mostró que es caro y a menudo erróneo, el equipo mantiene el sistema fácil de cambiar y protege solo las características que importan, aplazando las demás decisiones hasta el último momento responsable (Sección 1.6). Los críticos señalan límites reales. No todas las cualidades se pueden comprobar de forma automática o barata; los umbrales mal elegidos generan ruido que los equipos aprenden a ignorar; y algunas decisiones, como una base de datos o un proveedor de nube, siguen siendo puertas de un solo sentido por bueno que sea el pipeline. La arquitectura evolutiva es una disciplina para mantener reversibles las partes reversibles y protegidas las partes importantes, no una promesa de que todo se podrá cambiar más adelante.

EJEMPLO PRÁCTICO

Ejemplo práctico: tres escenarios del Capítulo 2 como funciones de aptitud

La Sección 2.8 construyó un árbol de utilidad para la plataforma de reparto de comida. El equipo convierte tres de sus hojas en funciones de aptitud, una activada y holística, otra un experimento de caos y otra estructural, después de que el ATAM de la Sección 11.1 llevara a una autorización de pago asíncrona.

Escenario 1: latencia de los pedidos (H,H)	En el pico del viernes de 100 pedidos/s, el 95% de las confirmaciones de pedido llegan en menos de 1.5 s y el 99% en menos de 3 s
Función de aptitud 1	Holística, activada: una prueba de carga con k6 contra un entorno similar al de producción, cada noche y antes de cada lanzamiento, que falla si el p95 o el p99 superan el umbral o si falla más del 0.1% de las solicitudes. Respaldada por una monitorización continua en producción de los mismos percentiles a partir del tráfico real
Presupuestos de latencia	Los 1.5 s se reparten en presupuestos: red móvil y TLS 400 ms, gateway 50 ms, pedidos 250 ms, precios 100 ms, escrituras en la base de datos 150 ms, lo que deja 550 ms de margen ahora que la autorización de la tarjeta está fuera de la ruta. El pipeline de cada servicio ejecuta una prueba de carga atómica del componente contra su propio presupuesto, de modo que una regresión se detecta en el servicio que la causó
Escenario 2: disponibilidad del pago (H,H)	Cuando el proveedor de pagos agota el tiempo de espera durante el pico, los pedidos se aceptan y se ponen en cola, no se pierde ningún pedido confirmado y se informa a los clientes en menos de 10 s
Función de aptitud 2	Experimento de caos holístico, semanal en staging y trimestral como game day en producción con un radio de impacto pequeño: se hace que el proveedor se quede colgado; los pedidos deben aceptarse como pendientes en menos de 10 s y todos deben liquidarse tras la recuperación. Responsable: equipo de pedidos
Escenario 3: nuevo proveedor de POS (M,L)	Un nuevo proveedor de punto de venta (POS) para restaurantes se integra en 5 días-persona añadiendo un adaptador
Función de aptitud 3	Atómica, activada, estática: un contrato de import-linter prohíbe que los módulos de dominio y de API de restaurantes y el paquete de pedidos importen el SDK de un proveedor, y un segundo contrato impide que los adaptadores se importen entre sí, para que el cambio quede localizado. Se complementa con una comprobación manual: se registra el esfuerzo de cada integración real con un proveedor y se compara con el objetivo de 5 días
Se registra con cada una	Escenario de origen, umbral, responsable, el ADR que lo explica y una fecha de revisión

Cada medida de la respuesta se convirtió en un umbral sin reinterpretarla, que es la recompensa de haber escrito bien los escenarios desde el principio.

El escenario de modificabilidad no se puede automatizar del todo: la regla estructural protege las condiciones para un cambio de 5 días, y la medición manual comprueba el resultado.

Los presupuestos de latencia convierten un único número holístico en comprobaciones atómicas que avisan al equipo correcto de una regresión.

```
import http from 'k6/http';
import { check } from 'k6';

export const options = {
  scenarios: {
    friday_peak: { executor: 'constant-arrival-rate', rate: 100, timeUnit: '1s',
                   duration: '10m', preAllocatedVUs: 300 },   // 100 orders per second
  },
  thresholds: {   // the scenario's response measure, as pass/fail rules
    'http_req_duration{name:place_order}': ['p(95)<1500', 'p(99)<3000'],
    'http_req_failed{name:place_order}': ['rate<0.001'],
  },
};

const order = { restaurantId: 'r-42', items: [{ sku: 'pad-thai', qty: 1 }] };

export default function () {
  const res = http.post(`${__ENV.BASE_URL}/orders`, JSON.stringify(order), {
    headers: { 'Content-Type': 'application/json' }, tags: { name: 'place_order' } });
  check(res, { 'order accepted': (r) => r.status === 201 || r.status === 202 });
}
```

Función de aptitud 1: una prueba de carga con k6 cuyos umbrales son la medida de la respuesta del escenario. k6 termina con un error cuando un umbral falla, lo que hace fallar el trabajo del pipeline.

```
# Fitness function: checkout keeps accepting orders when payments hang (illustrative)
# Runs weekly in staging; toxiproxy, api and ledger are test fixtures.
def test_orders_survive_payment_provider_outage(toxiproxy, api, ledger):
    toxiproxy.add_latency("payment-provider", latency_ms=30_000)   # provider hangs
    placed = [api.place_order(sample_order()) for _ in range(200)]

    assert all(r.status in (201, 202) for r in placed)           # still accepted
    assert all(r.json()["state"] == "PENDING_PAYMENT" for r in placed)
    assert max(r.elapsed_seconds for r in placed) < 10           # told within 10 s

    toxiproxy.remove_toxics("payment-provider")                   # provider recovers
    ledger.wait_until_settled(timeout_seconds=300)
    assert ledger.lost_orders(placed) == []                       # none lost
```

Función de aptitud 2: un experimento de caos escrito como prueba (ilustrativo; los fixtures envuelven un proxy de inyección de fallos, la API y el libro de pagos).

```
# .importlinter: only the POS adapters may use vendor SDKs
[importlinter]
root_package = fooddelivery
include_external_packages = True

[importlinter:contract:pos-vendors-isolated]
name = Vendor POS SDKs are used only inside the POS adapters
type = forbidden
source_modules =
    fooddelivery.restaurant.domain
    fooddelivery.restaurant.api
    fooddelivery.ordering
forbidden_modules =
    tillpro_sdk
    kitchenlink

[importlinter:contract:adapters-independent]
name = POS adapters do not import each other
type = independence
modules =
    fooddelivery.restaurant.pos_adapters.tillpro
    fooddelivery.restaurant.pos_adapters.kitchenlink
```

Función de aptitud 3: contratos de import-linter que mantienen el código específico de cada proveedor dentro de los adaptadores, ejecutados con lint-imports en cada compilación.

La compatibilidad de los datos se protege de la misma manera. Antes de que pueda fusionarse un cambio en el evento `OrderPlaced`, el pipeline pregunta al registro de esquemas si el nuevo esquema es compatible con la última versión registrada según la regla de compatibilidad del sujeto, y falla si no lo es:

```
# Data fitness function: block the merge if the new event schema would break consumers
curl -s -X POST -H "Content-Type: application/vnd.schemaregistry.v1+json" \
  --data @build/order-placed-v7.request.json \
  "$SCHEMA_REGISTRY_URL/compatibility/subjects/order-placed-value/versions/latest" \
  | jq -e '.is_compatible == true'
```

Una función de aptitud de datos que usa el endpoint de compatibilidad de un registro de esquemas; jq -e devuelve un código de salida distinto de cero, lo que hace fallar el trabajo, cuando el esquema es incompatible.

Diseñar funciones de aptitud que perduren

Empieza por poco y por lo importante. Elige las tres a cinco características que el árbol de utilidad valora más alto, escribe una función de aptitud para cada una y asegúrate de que sean rápidas y fiables antes de añadir más. Una docena de comprobaciones de confianza valen más que cien que fallan al azar.

Usa trinquetes (ratchets) en los sistemas existentes. Cuando una base de código ya viola una regla, registra las violaciones actuales como línea base y falla solo ante las nuevas; endurece la línea base a medida que el equipo salda la deuda. Lo mismo sirve para el rendimiento: falla ante regresiones superiores a un margen acordado respecto al último lanzamiento mientras avanzas hacia el objetivo del escenario.

Haz que los umbrales sean explícitos y tengan responsable. Guarda los umbrales en una configuración bajo control de versiones junto a una referencia al ADR que los justifica, con un responsable con nombre. Cuando una función de aptitud falla, el responsable decide entre corregir el código y cambiar la regla, y la decisión se registra. Trata una función de aptitud que se anula con frecuencia como un hallazgo: o la regla está mal o la arquitectura está sometida a una presión real.

Mantén el pipeline rápido. Las comprobaciones atómicas van en cada compilación; las comprobaciones holísticas pesadas, como las pruebas de carga de diez minutos, se ejecutan cada noche, antes del lanzamiento o sobre la versión candidata. Las comprobaciones continuas en producción necesitan el mismo cuidado que cualquier alerta: vinculadas al impacto en los usuarios, con un runbook y en silencio cuando no pasa nada.

ATENCIÓN

Antipatrones de las funciones de aptitud

- Comprobaciones inestables: una prueba de carga ejecutada en una infraestructura compartida y ruidosa falla al azar, así que los equipos aprenden a repetirla hasta que pasa y dejan de creer en ella.
- Hacer trampa con la medida: cuando un número se convierte en objetivo, la gente optimiza el número (ley de Goodhart, Sección 3.8); prefiere comprobaciones vinculadas a escenarios reales antes que métricas abstractas.
- Umbrales sin motivos: nadie sabe por qué el límite es 15, así que nadie se atreve a cambiarlo o todo el mundo lo ignora.
- Comprobaciones que solo informan: un panel que nadie mira no es una función de aptitud.
- Automatizar lo fácil e ignorar lo importante: decenas de reglas de estilo, pero ninguna comprobación de la latencia de la ruta del pedido ni de la alternativa de pago.
- Funciones de aptitud como sustituto de pensar: protegen decisiones ya tomadas; no te dicen qué decisiones tomar.

Preguntas frecuentes

¿Las funciones de aptitud son solo pruebas?
Muchas se implementan como pruebas, pero su objeto es distinto. Las pruebas ordinarias comprueban que las funcionalidades se comportan correctamente; las funciones de aptitud comprueban características de la arquitectura como la estructura, la latencia, la resiliencia o la compatibilidad. Algunas, como los monitores de producción y las revisiones manuales, no son pruebas en absoluto.

¿Quién escribe las funciones de aptitud: los arquitectos o los desarrolladores?
Ambos. Los arquitectos suelen identificar las características y los umbrales, y a menudo escriben las primeras comprobaciones, lo que es una buena forma de seguir con las manos en el código (Sección 1.4). Los equipos son propietarios de ellas y las mantienen como cualquier otro código.

¿Qué relación tienen las funciones de aptitud con los SLO?
Un SLO con una alerta sobre su presupuesto de errores es una función de aptitud continua y holística para la disponibilidad o la latencia en producción (Sección 9.5). Las funciones de aptitud del pipeline intentan detectar las mismas regresiones antes de que lleguen a los usuarios.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Cuál es la diferencia entre una métrica y una función de aptitud?

Una métrica describe una propiedad; una función de aptitud codifica una decisión con un umbral y una acción, de modo que incumplirla hace fallar una compilación o lanza una alerta y obliga a una elección consciente.

▼ Da un ejemplo de una función de aptitud atómica y otro de una holística.

Atómica: una regla que prohíbe importaciones desde la capa web hacia el dominio. Holística: una prueba de carga que mide la latencia de los pedidos con la seguridad, el logging y la caché activos a la vez.

▼ ¿Qué aporta un presupuesto de latencia a una función de aptitud de latencia de extremo a extremo?

Divide el objetivo de extremo a extremo en límites por servicio que cada servicio comprueba en su propio pipeline, de modo que una regresión se detecta en el servicio que la causó.

▼ ¿Cuáles son las tres partes de la definición de arquitectura evolutiva?

Admite un cambio guiado (por funciones de aptitud) e incremental (cambios y lanzamientos pequeños y seguros) en múltiples dimensiones (código, datos, seguridad, operaciones, equipos y más).

▼ Inténtalo: Convierte cada uno en una función de aptitud, con disparador y umbral: "El 95% de las confirmaciones de pedido en 1.5 s a 100 pedidos/s" y "Pedidos nunca debe importar las partes internas de Pagos". ¿Dónde se ejecutaría cada una y qué ocurre cuando una falla?

Latencia: una prueba de carga holística y dinámica a 100 pedidos por segundo, que falla si el p95 supera 1.5 s, ejecutada cada noche o antes del lanzamiento porque es lenta. Regla de dependencia: una comprobación atómica y estática con ArchUnit o import-linter, que falla ante cualquier importación prohibida, ejecutada en cada compilación. Un fallo obliga a tomar una decisión: corregir el código, o cambiar la regla y anotar por qué. Cada una necesita un responsable y un enlace a su escenario.

FUENTES

- Building Evolutionary Architectures, 2nd edition (O'Reilly): https://oreilly.com/library/view/building-evolutionary-architectures/9781492097532
- Evolutionary Architecture: book site with fitness function katas: https://evolutionaryarchitecture.com/

- Grafana k6 documentation: Thresholds: https://grafana.com/docs/k6/latest/using-k6/thresholds/
- import-linter documentation: Forbidden contracts: https://import-linter.readthedocs.io/en/stable/contract_types/forbidden/

SECCIÓN 11.4 DE 11.8 · AVANZADO · 28 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Gobernanza de la arquitectura en organizaciones ágiles

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.6 Decisiones y compromisos
- 4.5 Registros de decisiones
- 11.3 Funciones de aptitud

Términos anteriores que utiliza: Registro de decisiones de arquitectura (ADR), Función de aptitud (fitness function), Arquitectura empresarial

LA IDEA EN BREVE

Una ciudad no puede pedir al alcalde que apruebe cada cocina nueva. En su lugar, fija unos pocos principios (que las calles se puedan recorrer a pie), algunas normas de construcción que los inspectores comprueban automáticamente y un pequeño número de decisiones que de verdad necesitan un comité, como un puente nuevo. Los vecinos pueden hacer la mayoría de las cosas por sí mismos, pero para todo lo que afecte a los vecinos de al lado se espera que les pregunten primero. La gobernanza de la arquitectura en las organizaciones ágiles funciona igual: los equipos deciden la mayoría de las cosas por sí mismos, las comprobaciones automáticas hacen cumplir las reglas que una máquina puede comprobar, y las decisiones más grandes se toman después de consultar a las personas a las que afectan.

LO QUE NECESITAS SABER

- La gobernanza de la arquitectura garantiza que las decisiones significativas se tomen bien, de forma coherente con los objetivos de la organización, y que de verdad se sigan; sus propósitos incluyen la alineación, el riesgo y el cumplimiento normativo, limitar la diversidad innecesaria, proteger las cualidades compartidas y difundir el aprendizaje.
- Los principios dan una dirección duradera, los estándares hacen obligatorias elecciones concretas, los guardrails mantienen el trabajo dentro de unos límites de forma automática y las compuertas detienen el trabajo hasta su aprobación; prefiere los guardrails cuando las reglas son objetivas y mantén las compuertas tempranas y escasas.
- Los principios al estilo de TOGAF tienen nombre, enunciado, justificación e implicaciones; un buen principio podría elegirse razonablemente en sentido contrario y resuelve decisiones reales.

- En el proceso de consejo (advice process) (Andrew Harmel-Law, 2021), cualquiera puede tomar una decisión arquitectónica después de pedir consejo a los afectados y a los expertos, y la registra en un ADR; un foro asesor da visibilidad, pero no decide.
- Un radar tecnológico sitúa las tecnologías en anillos como Adopt, Trial, Assess y Hold, con razones por escrito; un radar interno recoge la experiencia propia de la organización y orienta las nuevas elecciones.
- La política como código (por ejemplo, Open Policy Agent con Rego) y las funciones de aptitud automatizan el cumplimiento normativo y producen pruebas para la auditoría como subproducto.

Para qué sirve la gobernanza de la arquitectura

La gobernanza de la arquitectura es el conjunto de prácticas con las que una organización se asegura de que sus decisiones técnicas significativas se toman bien, son coherentes con sus objetivos y se siguen de verdad. Responde a cuatro preguntas: quién puede tomar qué decisiones, cómo deben tomarlas, qué dirección deben seguir y cómo sabe nadie si el resultado se ajusta a lo previsto.

Los propósitos son prácticos. La gobernanza alinea la tecnología con la estrategia, para que cincuenta equipos no elijan cincuenta formas de almacenar los datos de los clientes. Gestiona el riesgo y el cumplimiento normativo: la protección de datos, la seguridad, la accesibilidad y la regulación sectorial necesitan pruebas de que se siguen las reglas. Limita la diversidad innecesaria, porque cada base de datos, lenguaje o producto de mensajería adicional necesita conocimientos, parches y soporte de guardia. Protege cualidades que ningún equipo posee por sí solo, como la latencia de extremo a extremo, la seguridad y el costo. Y difunde el aprendizaje, para que una lección que pagó un equipo no la vuelva a pagar el siguiente.

Las organizaciones ágiles sienten aquí una tensión. El principio del Manifiesto Ágil según el cual las mejores arquitecturas surgen de equipos autoorganizados (Sección 1.8) empuja hacia la autonomía; la escala, la regulación y la infraestructura compartida empujan hacia la alineación. Las descripciones de Henrik Kniberg de la cultura de ingeniería de Spotify popularizaron la expresión *autonomía alineada (aligned autonomy)* para el objetivo: equipos libres de decidir el cómo, dentro de un entendimiento compartido del qué y el porqué. La gobernanza es la maquinaria que lo hace posible. Mal hecha, se convierte en el lento proceso de aprobación de la Sección 11.2; bien hecha, los equipos apenas la notan, porque lo correcto es también lo fácil.

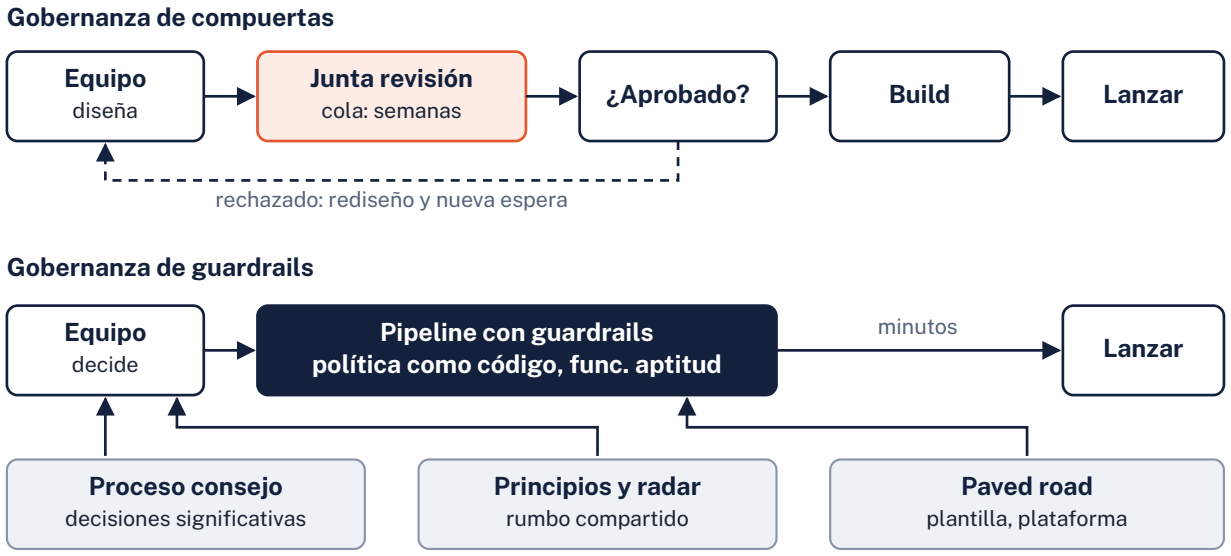
Principios, estándares, guardrails y compuertas

La gobernanza usa cuatro tipos de instrumento, que difieren en cuánto restringen y en cuándo actúan. Los principios expresan una dirección duradera y dejan margen al criterio. Los estándares son elecciones concretas y obligatorias (todos los servicios exponen trazas de OpenTelemetry; todas las API públicas se describen en OpenAPI). Los guardrails (salvaguardas) mantienen a los equipos dentro de límites seguros mientras trabajan, normalmente de forma automática y continua. Las compuertas (gates) detienen el trabajo hasta que alguien lo aprueba.

Instrumentos de gobernanza

Instrumento	Ejemplo	Fortaleza	Riesgo
Principio	Cada dato de negocio tiene exactamente un servicio propietario	Guía muchas decisiones, incluidas las imprevistas	Demasiado vago para actuar en consecuencia, o ignorado en la práctica
Estándar	Todos los servicios emiten trazas de OpenTelemetry y logs JSON estructurados	Coherencia, herramientas compartidas, incorporación sencilla	Se queda obsoleto; bloquea opciones mejores; las excepciones se acumulan
Guardrail (salvaguarda)	El pipeline rechaza imágenes de contenedor de registros no aprobados	Continuo, rápido, objetivo, sin cola	Solo sirve para reglas que una máquina puede comprobar; se puede eludir si nadie es su responsable
Compuerta	Visto bueno de seguridad antes de tratar una nueva categoría de datos personales	Criterio humano con rendición de cuentas para decisiones de alto riesgo e irreversibles	Retrasos, retroalimentación tardía, evasión y aprobaciones de trámite

Compuertas frente a guardrails



Una compuerta hace esperar la aprobación de cada diseño significativo; los guardrails comprueban continuamente las reglas objetivas y dejan a las personas libres para las decisiones que requieren criterio.

El consenso actual, aunque no universal, es preferir los guardrails a las compuertas siempre que una regla pueda formularse de manera objetiva, reservar las compuertas para las pocas decisiones en las que de verdad hace falta un criterio humano con rendición de cuentas, y celebrar esas compuertas pronto y con un tiempo limitado. Un camino pavimentado (paved road) (término de Netflix; Spotify lo llama camino dorado, golden path) complementa a ambos: cuando la forma admitida de construir un servicio ya incluye imágenes aprobadas, logging, análisis de seguridad y despliegue, la mayoría de los equipos cumplen sin que se les diga, y el esfuerzo de gobernanza puede concentrarse en las desviaciones deliberadas. La Sección 10.7 trata las plataformas que ofrecen esos caminos.

SI ESTO ES NUEVO PARA TI

Paso a paso: una regla como compuerta y como guardrail

La idea más difícil de esta sección es elegir el instrumento adecuado para cada regla: por qué una regla que una máquina puede comprobar debe ir en un guardrail, y por qué unas pocas decisiones siguen necesitando una compuerta. Sigue dos reglas de la tabla anterior.

- Toma la regla de que toda imagen de contenedor debe proceder del registro analizado de la organización. Supón que se hace cumplir con una compuerta: cada equipo enumera sus imágenes para que la junta de revisión mensual las apruebe.
- El equipo espera semanas a que haya un hueco. La junta revisa la lista una sola vez, así que una imagen cambiada al mes siguiente nunca se comprueba. Y como la cola es lenta, los equipos aprenden a calificar los cambios de menores para evitarla, como describió la Sección 11.2.
- Ahora haz cumplir la misma regla como guardrail: el pipeline rechaza cualquier despliegue cuya imagen proceda de otro lugar. Comprueba cada lanzamiento, en minutos, sin cola, y su registro sirve de prueba para un auditor.
- Esto funciona porque la regla es objetiva: una máquina puede responder sí o no. El guardrail sigue necesitando un responsable y una vía para las excepciones, o la gente encontrará formas de esquivarlo.
- Toma una segunda regla: el visto bueno de seguridad antes de que la plataforma trate una nueva categoría de datos personales. Ninguna máquina puede juzgar si la finalidad y los riesgos son aceptables, así que debe hacerlo una persona que rinda cuentas, y una compuerta es lo adecuado. Celébrala pronto, limítala en el tiempo e indica de antemano qué pruebas necesita.
- **La lección:** si una máquina puede comprobar una regla, conviértela en un guardrail; reserva las compuertas para las pocas decisiones de alto riesgo que necesitan un criterio humano con rendición de cuentas, y celébralas pronto. Las decisiones que necesitan criterio pero no aprobación pasan por el proceso de consejo, que se trata más adelante en esta sección.

Redactar principios de arquitectura

TOGAF, el marco de arquitectura empresarial de The Open Group (el TOGAF Standard, 10th Edition, 2022), ofrece una plantilla muy utilizada para los principios, con cuatro partes. El **nombre** es breve y fácil de recordar. El **enunciado** expresa la regla en una o dos frases inequívocas. La **justificación** explica el beneficio para el negocio, en términos que un patrocinador no técnico podría aceptar. Las **implicaciones** dicen qué exige el principio en la

práctica: qué hay que hacer, cuánto cuesta y qué descarta. TOGAF también enumera criterios para un buen conjunto de principios: comprensible, robusto (lo bastante preciso para decidir casos difíciles), completo, coherente y estable.

Dos pruebas detectan la mayoría de los principios débiles. Primero, ¿podría una organización sensata elegir lo contrario? "Los sistemas deben ser seguros" no la supera; nadie elegiría la inseguridad. "Preferir comprar a construir para las capacidades genéricas" la supera, porque algunas organizaciones prefieren, con razón, construir. Segundo, ¿cambia el principio una decisión real? Si nadie puede nombrar una elección que resolvería, es un eslogan. Andrew Harmel-Law pide un conjunto que no sea ni demasiado pequeño para cubrir las necesidades clave ni demasiado grande para que los equipos lo recuerden, que proceda de los propios equipos y esté vinculado a la estrategia de negocio, en lugar de máximas o prácticas generales de programación. Los principios más escasos y precisos se recuerdan y se usan; las listas largas, no.

EJEMPLO PRÁCTICO

Ejemplo práctico: tres principios para la plataforma de reparto de comida

Redactados por los arquitectos con los líderes de equipo, debatidos en el foro asesor y publicados junto al radar. Cada uno sigue la estructura de TOGAF.

Nombre de P1	**Un único propietario para cada dato**
Enunciado	Cada tipo de dato de negocio (pedidos, cartas, posiciones de los repartidores, pagos) tiene exactamente un servicio propietario, que es el único que lo escribe; los demás usan su API o sus eventos.
Justificación	Los datos compartidos con escritura acoplan los lanzamientos de los equipos y hacen que los errores no se puedan rastrear (Sección 7.1). Una propiedad clara permite a los equipos cambiar sus esquemas y desplegar de forma independiente, que es como el negocio lanza cada semana.
Implicaciones	Ningún servicio lee ni escribe las tablas de otro. Los informes usan eventos publicados o un almacén de datos. Los datos nuevos necesitan un propietario con nombre antes de almacenarse. Una función de aptitud comprueba las credenciales de base de datos por servicio.
Nombre de P2	**Asíncrono entre dominios**
Enunciado	Los dominios se integran por defecto mediante eventos publicados; las llamadas síncronas entre dominios se usan solo cuando el usuario está esperando la respuesta.
Justificación	Las cadenas síncronas multiplican la indisponibilidad y añaden latencia (Sección 6.1); el pico del viernes y las caídas de los socios son los mayores riesgos operativos del negocio.
Implicaciones	Los equipos diseñan para la consistencia eventual y los consumidores idempotentes. Cada llamada síncrona entre dominios tiene un timeout, una alternativa y un presupuesto de latencia. Más inversión operativa en el broker y en el trazado.
Nombre de P3	**Comprar lo genérico, construir lo central, seguir siendo portables**
Enunciado	Las capacidades genéricas (identidad, pagos, correo electrónico, monitorización) se compran o se usan como servicios gestionados; las capacidades diferenciadoras (optimización del despacho, herramientas para restaurantes) se construyen. La lógica de negocio queda detrás de puertos, no dentro de los servicios propietarios de un proveedor.
Justificación	El tiempo de ingeniería es el recurso más escaso y debe ir a donde compite el negocio (dominios centrales, Sección 5.7), evitando a la vez una dependencia del proveedor (lock-in) que debilitaría el poder de negociación (Sección 10.8).

Implicaciones	Construir o comprar es una pregunta explícita en cada ADR pertinente. Se accede a los servicios gestionados a través de adaptadores. Algunas funcionalidades gestionadas con una dependencia profunda del proveedor se rechazan aunque sean cómodas.

Cada principio resuelve elecciones reales y recurrentes, y una organización sensata podría elegir otra cosa.

Las implicaciones incluyen los costos, no solo los beneficios.

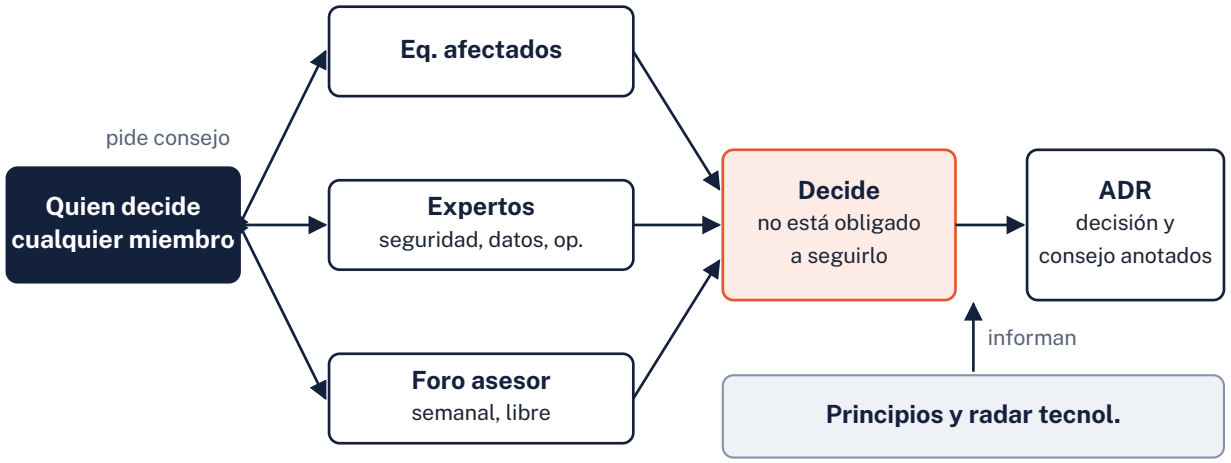
P1 y P2 se pueden hacer cumplir en parte como guardrails; P3 necesita criterio humano, así que se aplica mediante el proceso de consejo.

Toma de decisiones descentralizada: el proceso de consejo

Si las decisiones ya no se canalizan a través de una junta, ¿quién las toma? El artículo de Andrew Harmel-Law *Scaling the Practice of Architecture, Conversationally* (martinfowler.com, diciembre de 2021) propuso una respuesta tomada de las organizaciones autogestionadas: el proceso de consejo (advice process). Su única regla es que cualquiera puede tomar una decisión arquitectónica, siempre que antes de decidir pida consejo a todas las personas a las que afectará de forma significativa y a personas con conocimientos relevantes. Quien toma la decisión debe escuchar y registrar el consejo, pero no está obligado a seguirlo; el objetivo es una aportación amplia, no el consenso. La responsabilidad sigue siendo de la persona o el equipo que decide, y eso es lo que hace funcionar la regla: quien sabe que carga con las consecuencias pide consejo en serio.

Harmel-Law describe cuatro elementos de apoyo. Los registros de decisiones (ADR, Sección 4.5) son la herramienta para pensar y registrar, e incluyen el consejo recibido, de quién y cuándo. Un foro asesor de arquitectura, normalmente semanal, de alrededor de una hora y abierto a cualquiera, da visibilidad a los nuevos spikes, a las decisiones propuestas y a las que se revisan; y, lo que es fundamental, ofrece consejo, pero no decide. Los principios arquitectónicos que proceden de los equipos dan una dirección compartida. Y un radar tecnológico interno muestra el panorama tecnológico actual de la organización y su postura.

El proceso de consejo



El consejo llega de los afectados y de los expertos; la decisión y el consejo se registran juntos, y la responsabilidad sigue siendo de quien toma la decisión.

El proceso de consejo tiene modos de fallo que conviene conocer. Quienes deciden pueden consultar solo a las personas que probablemente estarán de acuerdo; ayuda una lista publicada de a quién hay que consultar para cada tipo de decisión (a seguridad para cualquier cosa que toque datos personales, al equipo de plataforma para la nueva infraestructura). El consejo puede pedirse demasiado tarde, cuando el código ya está escrito; ayuda hacer del ADR propuesto el comienzo del trabajo, no el final. Y a algunas personas les resulta incómodo al principio que el foro no decida. En entornos regulados, el proceso puede convivir con un pequeño número de compuertas formales, aplicadas solo donde la ley o el apetito de riesgo de la organización lo exijan.

Radares tecnológicos

Thoughtworks publica su Technology Radar dos veces al año desde 2010. Cada edición sitúa un centenar de blips (técnicas, herramientas, plataformas, y lenguajes y frameworks: los cuatro cuadrantes) en anillos que expresan una recomendación. Adopt significa probado y recomendado; Trial, que merece la pena aplicarlo en un proyecto que pueda asumir el riesgo; Assess, que merece la pena explorarlo para entender su efecto; y el anillo exterior, llamado Hold durante la mayor parte de la historia del radar y Caution en sus ediciones más recientes, significa avanzar con cuidado. Cada blip tiene una breve justificación por escrito, que es la parte más valiosa.

Muchas organizaciones crean su propio radar, para sí mismas y no para el sector. Thoughtworks ofrece una herramienta de código abierto, Build Your Own Radar, que genera un radar a partir de una hoja de cálculo o un archivo CSV con las columnas name, ring, quadrant, isNew y

description. El valor está menos en la imagen que en el proceso: los equipos proponen blips, un grupo pequeño con representantes de muchos equipos los debate, los anillos se asignan con una razón por escrito y el radar se actualiza cada trimestre o cada semestre. Un anillo Hold es una herramienta de gobernanza suave pero eficaz: dice "ningún uso nuevo" sin prohibir lo que ya existe. Un radar se diferencia de una lista de tecnologías aprobadas en que es explícitamente provisional y razonado, y en que recoge lo que la organización ha aprendido, incluidos sus errores.

EJEMPLO PRÁCTICO

Ejemplo práctico: una entrada del radar y el registro de decisión que la respalda

El equipo de pedidos quiere sustituir su relé de outbox por sondeo (Sección 6.7) por una captura de datos de cambios con Debezium, que lee el registro de escritura anticipada (write-ahead log) de la base de datos y publica los cambios en Kafka. Ningún equipo ha usado Debezium antes, así que la decisión afecta al equipo de plataforma, que gestiona Kafka Connect, y al equipo de datos, responsable de las operaciones de la base de datos.

1. Borrador	El equipo redacta el ADR-0031 con estado Proposed (propuesto) y lo presenta en el foro asesor semanal
2. Pedir consejo	Los equipos de plataforma, datos y seguridad comentan en el pull request; el foro sugiere considerar la CDC gestionada del proveedor de nube
3. Decidir	El equipo de pedidos acepta la mayor parte del consejo, rechaza la opción gestionada por el principio P3 (portabilidad) y registra por qué
4. Proteger	Se añaden una función de aptitud sobre el retraso de los eventos (p99 por debajo de 2 s) y una alerta sobre el retraso del slot de replicación
5. Radar	Debezium entra en el radar en Trial, limitado a pedidos; los relés por sondeo pasan a Hold
6. Revisión	Al cabo de un trimestre, el grupo del radar revisa la experiencia; si es positiva, Debezium pasa a Adopt y se cierra la nota de revisión del ADR

La entrada del radar es el breve resumen público; el ADR contiene el razonamiento y el consejo. Cada uno remite al otro.

Trial tiene un alcance limitado, lo que mantiene la decisión reversible mientras la organización aprende.

Nada esperó en una cola: la decisión llevó alrededor de una semana, dedicada en su mayor parte a reunir consejos realmente útiles.

```
name,ring,quadrant,isNew,description
Debezium for outbox relay,Trial,Tools,TRUE,"Change data capture from the order database's
write-ahead log to Kafka, replacing polling relays. Trial in ordering only; see ADR-0031.
Owner: ordering team. Review: Q3."
Polling outbox relay,Hold,Techniques,FALSE,"Adds load and latency at peak; no new uses.
Existing relays move when Debezium leaves Trial."
```

Las entradas del radar en el formato CSV de Build Your Own Radar (name, ring, quadrant, isNew, description).

```
# ADR-0031: Use Debezium change data capture for the ordering outbox relay

Status: Accepted (2026-05-14)   Decision-taker: Priya N., ordering team
Context: The polling relay queries the outbox table every 500 ms. At 100 orders/s
it adds measurable database load and up to 1 s of event delay to dispatch.
Options: (1) tune polling; (2) Debezium CDC on the write-ahead log; (3) managed CDC
from the cloud provider.
Decision: Option 2, for the ordering service only, as a Trial on the radar.
Advice received:
- Platform team (2026-05-06): supports; will run Kafka Connect, needs runbook.
- Data team (2026-05-07): concern about replication slot growth if Connect stops;
  asks for an alert on slot lag. Accepted.
- Security (2026-05-08): CDC user must have replication rights only. Accepted.
- Advisory forum (2026-05-09): suggested option 3; rejected for portability (P3).
Consequences: lower load and delay; a new component to operate; slot-lag alert
and a fitness function on event delay (p99 under 2 s) added.
Review: after one quarter, or before any second service adopts it.
```

El ADR-0031 en un formato compacto al estilo de Nygard, con el consejo registrado como exige el proceso de consejo.

El papel actual de la arquitectura empresarial

La arquitectura empresarial (Sección 1.2) ha tenido una reputación incómoda en las organizaciones ágiles: equipos centrales que producían marcos, estándares y procesos de aprobación que los equipos de entrega vivían como obstáculos. El papel ha ido cambiando. En las organizaciones donde funciona bien, los arquitectos empresariales curan los principios y el radar en lugar de dictarlos, mantienen el mapa de capacidades de negocio y sistemas que ningún equipo puede ver por sí solo, toman o facilitan las decisiones de puerta de un solo sentido que de verdad afectan a toda la organización (identidad, datos centrales, estrategia de nube, grandes proveedores), dirigen el foro asesor y dan forma a la plataforma y al camino pavimentado junto con los equipos de plataforma.

La imagen de Gregor Hohpe del arquitecto que viaja en ascensor entre la sala de juntas y la sala de máquinas (Sección 11.7) describe la habilidad que esto requiere: traducir la estrategia en dirección técnica y las restricciones técnicas al lenguaje del negocio. Los profesionales siguen sin ponerse de acuerdo sobre cuánta autoridad central es sana. Algunos sostienen que

una función central fuerte es imprescindible en empresas reguladas, con muchas fusiones o muy integradas; otros sostienen que la mayor parte de su valor puede ofrecerse mediante plataformas, principios y consejo. La respuesta depende del tamaño de la organización, de su regulación y de qué parte de su panorama de sistemas es realmente compartida.

Cumplimiento mediante la automatización

Sea cual sea el modelo organizativo, el cumplimiento más barato es el automatizado. Las funciones de aptitud (Sección 11.3) comprueban las características de la arquitectura. La política como código hace lo mismo con las reglas de la organización: las reglas se escriben en un lenguaje legible por máquinas, se guardan en el control de versiones, se revisan y se prueban como el código, y se evalúan automáticamente contra definiciones de infraestructura, manifiestos de despliegue o solicitudes a la API. Open Policy Agent, un proyecto graduado de la Cloud Native Computing Foundation, es un motor de uso general muy utilizado con su propio lenguaje de políticas, Rego; Conftest aplica políticas Rego a archivos de configuración en un pipeline, y los controladores de admisión como OPA Gatekeeper o Kyverno hacen cumplir políticas dentro de los clústeres de Kubernetes.

El cumplimiento automatizado tiene otra ventaja que las organizaciones reguladas valoran mucho: las pruebas. Un registro del pipeline que muestra que cada lanzamiento pasó las mismas comprobaciones es una prueba mucho más sólida para un auditor que un formulario firmado, y se produce sin costo adicional.

```
package main

import rego.v1

# Guardrail: images must come from the organisation's scanned registry
deny contains msg if {
    input.kind == "Deployment"
    some c in input.spec.template.spec.containers
    not startswith(c.image, "registry.fooddelivery.example/")
    msg := sprintf("%s: image %s is not from the approved registry",
[input.metadata.name, c.image])
}

# Guardrail: every container declares a memory limit
deny contains msg if {
    input.kind == "Deployment"
    some c in input.spec.template.spec.containers
    not c.resources.limits.memory
    msg := sprintf("%s: container %s has no memory limit", [input.metadata.name, c.name])
}
```

Dos guardrails como políticas Rego, comprobados contra manifiestos de despliegue de Kubernetes con Conftest (conftest test deploy/) en cada ejecución del pipeline.

Equilibrar autonomía y alineación

La pregunta práctica es qué decisiones se gobiernan y cómo. Una regla útil se deriva de la Sección 1.6: cuanto más reversible y local es una decisión, más cerca del equipo debe tomarse; cuanto más irreversible y de mayor alcance, más consejo y, en ocasiones, aprobación necesita. La tabla presenta un reparto típico; cada organización ajusta el suyo.

Quién decide qué

Tipo de decisión	Quién decide	Mecanismo	Ejemplo
Local y reversible	El equipo	Criterio del equipo; ADR si es significativa	Una biblioteca dentro de un servicio; la estructura interna de los módulos
Afecta a otros equipos, reversible con un costo moderado	El equipo, después de pedir consejo	Proceso de consejo, foro asesor, radar	Adoptar Debezium; un nuevo evento publicado para otros
Para toda la organización o difícil de revertir	Un responsable sénior con nombre, después de una consulta amplia	Proceso de consejo con arquitectos empresariales; cambio de principio o de radar	Elección del proveedor de identidad; proveedor de nube principal
Reglas objetivas y comprobables	Nadie, caso por caso	Guardrails: política como código y funciones de aptitud	Registros de imágenes aprobados; ningún bucket de almacenamiento público
Regulada o de alto riesgo	Un aprobador que rinde cuentas	Compuerta temprana y limitada en el tiempo, con pruebas	Tratamiento de datos de salud; funciones relacionadas con la seguridad (safety)

La propia gobernanza debe medirse. Son señales útiles cuánto tardan las decisiones significativas desde la propuesta hasta la aceptación, cuántas excepciones a los estándares se solicitan y se conceden, qué proporción de servicios está en el camino pavimentado, cuántas tecnologías en Hold se siguen extendiendo y si los incidentes se remontan a decisiones que se saltaron el consejo. Si las decisiones son lentas, la gobernanza es demasiado pesada; si los incidentes siguen remontándose a decisiones no consultadas, es demasiado ligera.

GUÍA DE DECISIÓN

Guía de decisión: ¿compuerta, guardrail o consejo?

Úsala al decidir cómo debe gobernarse una regla concreta o un tipo de decisión.

Q1 **¿Puede formularse la regla de manera objetiva y comprobarse con una máquina?**

- **Sí** Conviértela en un guardrail: política como código o una función de aptitud en el pipeline, con un responsable y una vía para las excepciones.
- **No** Continúa.

Q2 **¿Es la decisión local de un equipo y barata de revertir?**

- **Sí** Deja que el equipo decida; registra un ADR si es significativa.
- **No** Continúa.

Q3 **¿Exigen la ley, la regulación o el apetito de riesgo de la organización una aprobación formal de alguien que rinda cuentas?**

- **Sí** Usa una compuerta, pero celébrala pronto, limítala en el tiempo y deja claras las pruebas que necesita.
- **No** Continúa.

Q4 **¿Es una puerta de un solo sentido que afecta a muchos equipos?**

- **Sí** Usa el proceso de consejo con una consulta amplia, que incluya a los arquitectos empresariales, y un responsable sénior de la decisión con nombre.
- **No** Usa el proceso de consejo: consulta a los afectados y a los expertos, y después decide y registra.

ATENCIÓN

Errores habituales de gobernanza

- Principios con los que nadie podría estar en desacuerdo y que, por tanto, no deciden nada.
- Estándares sin una vía para las excepciones, que los equipos acaban ignorando en silencio.
- Un foro asesor que poco a poco se convierte en una junta de aprobación.
- Un radar mantenido por una sola persona, que recoge opiniones en lugar de la experiencia de la organización.
- Compuertas al final de la entrega, donde los únicos resultados realistas son la aprobación o una costosa reelaboración.
- Automatizar reglas de las que nadie es responsable, de modo que los falsos positivos se esquivan en lugar de corregirse.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Cuál es la diferencia entre un guardrail y una compuerta?

Un guardrail mantiene el trabajo dentro de unos límites de forma continua, normalmente mediante comprobaciones automatizadas, sin detenerlo; una compuerta detiene el trabajo hasta que alguien lo aprueba.

▼ ¿Cuáles son las cuatro partes de un principio de arquitectura al estilo de TOGAF?

Nombre, enunciado, justificación e implicaciones.

▼ En el proceso de consejo, ¿quién toma la decisión y debe seguir el consejo?

La persona o el equipo que toma la decisión, que debe pedir y registrar el consejo de los afectados y de los expertos, pero no está obligado a seguirlo.

▼ ¿Qué suele significar para los usos existentes situar una tecnología en el anillo Hold de un radar?

Significa que no deberían iniciarse usos nuevos; los usos existentes no se prohíben, aunque pueden migrarse con el tiempo.

▼ **Inténtalo: Cada nueva base de datos necesita la aprobación de un comité mensual de arquitectura, y las colas llegan a seis semanas. Los equipos ahora evitan preguntar. La seguridad debe seguir garantizada. ¿Cómo rediseñarías esta gobernanza usando guardrails y el proceso de consejo (advice process)?**

Sustituye la aprobación obligatoria por guardrails donde las reglas sean objetivas: la política como código puede comprobar automáticamente el cifrado, las copias de seguridad y los motores aprobados, y dar evidencias de auditoría. Publica un radar que diga qué bases de datos están en Adopt, Trial o Hold. Para cualquier cosa inusual, usa el proceso de consejo: el equipo decide tras consultar a las personas afectadas y a los expertos, y anota un ADR. El foro da visibilidad, pero no aprueba. Mantén una aprobación temprana, si acaso, solo para los cambios realmente de alto riesgo.

FUENTES

- Andrew Harmel-Law: Scaling the Practice of Architecture, Conversationally (martinfowler.com, 2021): https://martinfowler.com/articles/scaling-architecture-conversationally.html
- Thoughtworks Technology Radar: FAQ: https://www.thoughtworks.com/radar/faq
- Thoughtworks: Build Your Own Radar: https://www.thoughtworks.com/radar/byor
- Open Policy Agent documentation: https://www.openpolicyagent.org/docs/latest/

SECCIÓN 11.5 DE 11.8 · AVANZADO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Team Topologies y diseño sociotécnico

ANTES DE LEER ESTO

Esta sección se basa en:

- 3.7 Ley de Conway
- 5.7 Diseño guiado por el dominio
- 10.7 Ingeniería de plataformas

Términos anteriores que utiliza: Ley de Conway, Contexto delimitado (bounded context), Plataforma interna para desarrolladores (IDP)

LA IDEA EN BREVE

Piensa en un grupo de restaurantes. La cocina de cada restaurante atiende a sus propios comensales de principio a fin. Un proveedor central suministra ingredientes y equipamiento a todas las cocinas a partir de un catálogo sencillo, para que los cocineros no tengan que llevar una granja. Un taller especializado de pastelería elabora el único producto demasiado difícil de dominar para todas las cocinas. Y un jefe de cocina visitante pasa unas semanas enseñando a una cocina una técnica nueva y después se va. Team Topologies ofrece a las organizaciones de software los mismos cuatro tipos de equipo y tres formas de trabajar juntos, para que cada equipo se haga cargo de un trabajo que de verdad pueda abarcar mentalmente.

LO QUE NECESITAS SABER

- La arquitectura sociotécnica diseña a la vez la estructura del software y la estructura de los equipos, porque la ley de Conway hace que cada una dé forma a la otra.
- Team Topologies (Skelton y Pais, 2019; segunda edición 2025) define cuatro tipos de equipo: alineado al flujo (el predeterminado), de plataforma, facilitador y de subsistema complicado.
- Los equipos interactúan mediante tres modos deliberados: colaboración (descubrimiento con un plazo acotado), X como servicio (X-as-a-Service: consumir una interfaz clara) y facilitación (ayudar a otro equipo a adquirir una capacidad).
- La carga cognitiva del equipo, adaptada de la teoría de la carga cognitiva de John Sweller (1988), limita cuánto puede tener a su cargo un equipo; reduce la carga extrínseca con plataformas y automatización, y acota la carga intrínseca con límites bien situados.

- Una API del equipo documenta lo que un equipo ofrece y lo que espera, tanto las interfaces técnicas como las formas de trabajar; la plataforma viable más delgada es la plataforma más pequeña que realmente acelera a sus usuarios.
- Los planos de fractura, como los contextos delimitados, el alcance regulatorio, el ritmo de cambio, el riesgo y el aislamiento del rendimiento, muestran por dónde dividir el software y los equipos; la tecnología por sí sola suele ser la juntura más débil.
- Reorganizar los equipos no reestructura el código acoplado, así que los cambios en los equipos y los cambios en la arquitectura deben planificarse y evolucionar juntos.

Diseñar equipos es trabajo de arquitectura

El Capítulo 3 mostró que la ley de Conway no es una curiosidad sino una fuerza: el sistema que construye un grupo de equipos tiende a copiar la forma en que esos equipos se comunican. Si eso es cierto, decidir cómo se forman los equipos, de qué se hace cargo cada uno y cómo interactúan es una decisión arquitectónica, con consecuencias tan duraderas como elegir una base de datos o un estilo de integración. La disciplina de diseñar a la vez la estructura del software y la estructura de los equipos se llama arquitectura sociotécnica.

El marco práctico más influyente para hacerlo es Team Topologies, de Matthew Skelton y Manuel Pais (IT Revolution, 2019; segunda edición 2025). Parte de un objetivo sencillo, el *flujo rápido*: una corriente constante y sostenible de cambios pequeños desde la idea hasta producción, con poca espera entre equipos. Después ofrece un vocabulario reducido para diseñar organizaciones que lo logren: cuatro tipos de equipo, tres modos de interacción, un límite llamado carga cognitiva del equipo y una técnica para encontrar las junturas por las que pueden dividirse el software y los equipos.

Varias premisas recorren el enfoque. La unidad de entrega es el equipo, no el individuo: el trabajo se asigna a equipos, la propiedad pertenece a los equipos y los equipos son estables, de modo que acumulan conocimiento de su dominio y de su código en lugar de formarse para un proyecto y dispersarse después. Los equipos son pequeños, normalmente de cinco a nueve personas, porque la confianza y la comprensión compartida disminuyen rápidamente por encima de ese tamaño (los autores se basan en la investigación de Robin Dunbar sobre el tamaño de los grupos). Y los límites de los equipos deben diseñarse para el flujo del cambio, no para la comodidad de las líneas jerárquicas ni según la historia de quién construyó qué.

Para los arquitectos, esto cambia el trabajo. Un límite de servicio propuesto solo es bueno si algún equipo puede hacerse cargo de él por completo; un equipo propuesto solo es viable si el código del que se hace cargo le permite trabajar sin pedir permiso a otros tres equipos. El arquitecto que dibuja cajas sin preguntar quién se hará cargo de cada una está diseñando la mitad del sistema.

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

Un *flujo de valor* es la secuencia de trabajo que convierte una necesidad en algo de lo que se beneficia un cliente o un usuario, por ejemplo, todo lo necesario para que los clientes programen un pedido para más tarde. *Multifuncional* significa que el equipo reúne todas las competencias que requiere ese trabajo (diseño, frontend, backend, pruebas, operaciones) en lugar de pasar trabajo a otros equipos. La Sección 3.7 de este curso explica la ley de Conway y el reflejo entre organización y sistema sobre el que se apoya esta sección.

Los cuatro tipos fundamentales de equipo

Team Topologies sostiene que casi todos los equipos de una organización de software deberían ser de uno de cuatro tipos. El sentido de tener tan pocos tipos es la claridad: cada tipo tiene un propósito distinto, una forma distinta de medir el éxito y una relación distinta con los demás equipos, de modo que conocer el tipo de un equipo indica a todos qué esperar de él.

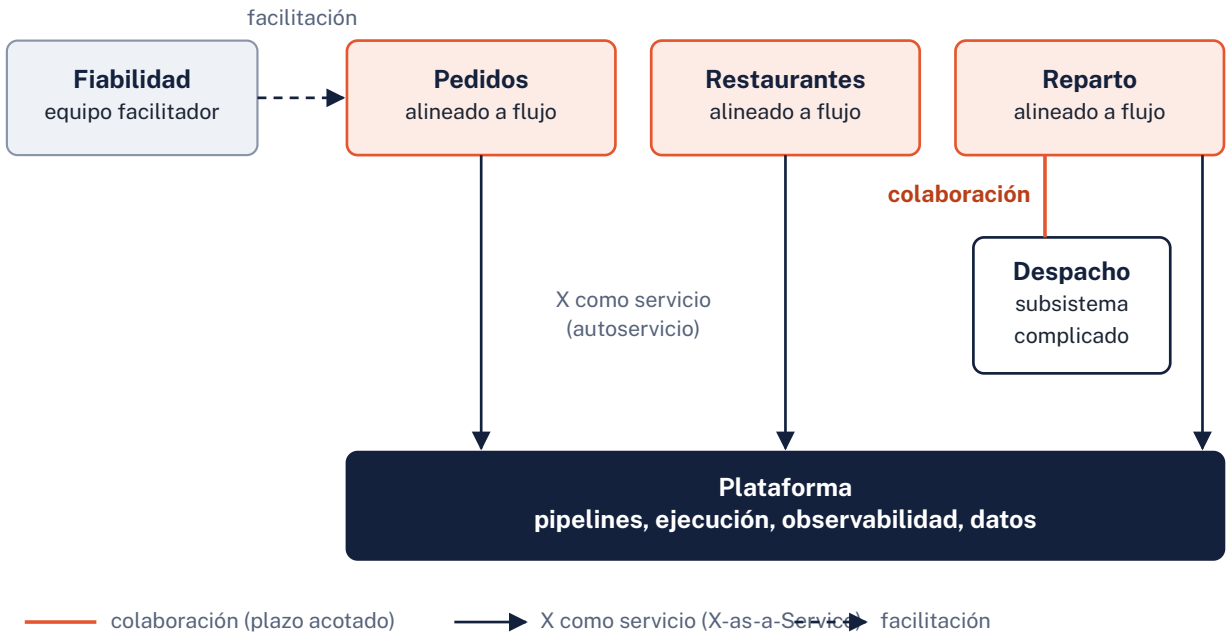
Un equipo alineado al flujo (stream-aligned) está alineado con un flujo de cambio: un producto, un recorrido del cliente, un grupo de usuarios o una capacidad de negocio. Se hace cargo de su parte de extremo a extremo, desde el descubrimiento y la construcción hasta su operación en producción, y debería poder entregar la mayoría de los cambios sin esperar a nadie. Los equipos alineados al flujo son el tipo predeterminado y deberían ser la gran mayoría; los otros tres tipos existen para reducir su carga.

Un equipo de plataforma proporciona servicios internos (pipelines de despliegue, un entorno de ejecución, observabilidad, infraestructura de datos) como un producto que los equipos alineados al flujo consumen mediante autoservicio. Una buena plataforma se juzga por la velocidad de sus usuarios, no por su propia lista de funcionalidades. Un equipo facilitador está formado por especialistas que ayudan a otros equipos a adquirir una capacidad de la que carecen, como la automatización de pruebas, las prácticas de observabilidad o un framework nuevo, y después pasan a otra cosa; su éxito se mide porque el otro equipo deja de necesitarlo. Un equipo de subsistema complicado se hace cargo de una parte que requiere conocimientos poco comunes, como un motor de precios basado en modelos actuariales, un solucionador de optimización de rutas o un códec de video, para que los equipos alineados al flujo puedan usarla sin tener que dominarla. El libro insiste en que este tipo debe usarse con moderación y solo donde la especialidad sea real.

Comparación de los cuatro tipos de equipo

Tipo de equipo	Propósito	Se hace cargo de	Señales de buena salud	Ejemplo de reparto de comida
Alineado al flujo	Entregar un flujo continuo de cambios para una parte del negocio	Las funcionalidades, el código, los datos y la operación de su flujo, de extremo a extremo	Lanza de forma independiente; la mayoría de los cambios no necesitan a ningún otro equipo; la carga de guardias es llevadera	Equipo de pedidos de clientes: cartas que ven los clientes, carrito, proceso de pago, seguimiento de pedidos
De plataforma	Reducir la carga cognitiva de los equipos alineados al flujo mediante productos internos de autoservicio	Los servicios internos de la plataforma, su documentación y su fiabilidad	Los equipos la adoptan voluntariamente; incorporar un servicio nuevo lleva horas, no semanas	Equipo de plataforma de entrega: pipelines, entorno de ejecución de contenedores, logging y métricas, intermediario de eventos
Facilitador	Cubrir carencias de capacidad en otros equipos y después retirarse	Conocimiento práctico, guías, ejemplos; no el código de producción de otros equipos	Las intervenciones tienen un plazo acotado y terminan con el otro equipo siendo autosuficiente	Equipo facilitador de fiabilidad que asesora a los equipos sobre objetivos de nivel de servicio y prácticas ante incidentes
Subsistema complicado	Encapsular una especialidad demasiado profunda para cada equipo alineado al flujo	Un componente especializado detrás de una interfaz clara	Los consumidores lo usan como un servicio sin necesitar a los especialistas presentes	Equipo de optimización del despacho: el algoritmo que asigna repartidores a los pedidos

Tipos de equipo y modos de interacción



La mayoría de los equipos están alineados al flujo; una plataforma les da servicio en modo X como servicio, un equipo facilitador los facilita y un equipo de subsistema complicado colabora estrechamente con un flujo durante un tiempo limitado.

Tres modos de interacción

Los tipos de equipo dicen qué son los equipos; los modos de interacción dicen cómo trabajan juntos. Skelton y Pais sostienen que la interacción ambigua e improvisada entre equipos es una fuente importante de retrasos, porque cada conversación no planificada es una dependencia. Por eso recomiendan que cada par de equipos que deban interactuar elija deliberadamente uno de tres modos, durante un periodo determinado.

En la **colaboración**, dos equipos trabajan estrechamente en un objetivo compartido, con responsabilidades difusas, por ejemplo, mientras descubren cómo debe dividirse una capacidad nueva. Produce un aprendizaje rápido, pero cuesta a ambos equipos capacidad y carga cognitiva, así que debe tener un plazo acotado. En X como servicio (X-as-a-Service), un equipo proporciona algo (una API, una capacidad de plataforma, una biblioteca) que otro consume con poca o ninguna colaboración; escala bien, pero solo funciona una vez que el límite se entiende bien y la interfaz es buena. En la **facilitación**, un equipo, normalmente un equipo facilitador, ayuda a otro a aprender o a eliminar un obstáculo.

Los modos están pensados para cambiar. Una secuencia habitual es descubrir y luego asentar: un equipo de plataforma colabora con un equipo alineado al flujo durante unas semanas para diseñar una capacidad nueva, después la convierte en una oferta de autoservicio y pasa a X

como servicio. Los modos también funcionan como mecanismo de detección. Si un equipo que debería consumir un servicio como servicio sigue necesitando reuniones, probablemente la interfaz o el límite están mal; si dos equipos llevan un año colaborando, quizá en realidad sean un solo equipo, o la división entre ellos esté en el lugar equivocado.

Los tres modos de interacción

Modo	Cómo se manifiesta	Úsalo cuando	Costo o señal de alerta
Colaboración	Dos equipos trabajan como uno solo en un problema compartido durante un tiempo; sesiones de diseño conjuntas, elementos de backlog compartidos	Se está descubriendo un límite, una tecnología o una capacidad nuevos y nadie conoce aún la interfaz adecuada	Carga cognitiva alta en ambos equipos; si se prolonga indefinidamente, probablemente el límite está mal
X como servicio (X-as-a-Service)	Un equipo consume una API, una plataforma o un componente que proporciona otro, mediante documentación y autoservicio	El límite y la interfaz se entienden bien y son estables	Se adapta con lentitud si la interfaz es deficiente; las solicitudes de soporte constantes muestran que aún no es un servicio de verdad
Facilitación	Asesoramiento, trabajo en pareja, ejemplos y talleres breves que ayudan a un equipo a adquirir una capacidad	A un equipo le falta una competencia o lo bloquea un obstáculo que puede aprender a eliminar	Acaba convirtiéndose en hacer el trabajo del otro equipo, lo que crea dependencia en lugar de capacidad

La carga cognitiva como restricción de diseño

La idea de carga cognitiva procede de la psicología de la educación. La teoría de la carga cognitiva de John Sweller, publicada por primera vez en 1988, partía de la observación de que la memoria de trabajo humana solo puede retener y procesar unos pocos elementos nuevos a la vez, por lo que la enseñanza debe diseñarse para no sobrecargarla. La teoría distingue la carga intrínseca (la dificultad inherente del material), la carga extrínseca (la dificultad causada por la forma en que se presenta la tarea) y, en trabajos posteriores, la carga pertinente (el esfuerzo que construye una comprensión duradera). Los investigadores aún debaten si la carga pertinente es una categoría aparte, y Team Topologies usa las tres como heurística práctica, no como medición.

La Sección 3.7 presentó la versión de esta idea aplicada a los equipos: el total que un equipo debe tener en la cabeza para construir, operar y cambiar aquello de lo que se hace cargo. Lo que importa aquí es cómo usarla como restricción de diseño. Cuando una arquitectura asigna a un equipo más de lo que puede comprender, los efectos son previsibles: entregas más lentas,

correcciones superficiales, guardias frágiles, conocimiento concentrado en una o dos personas y, con el tiempo, salidas de la empresa. Son los mismos síntomas que a menudo se atribuyen a malos ingenieros o a mal código.

El libro ofrece reglas prácticas para dimensionar las responsabilidades de un equipo según la complejidad del dominio: asigna cada dominio a un solo equipo; un equipo suele poder llevar dos o tres dominios sencillos; un equipo que se hace cargo de un dominio complejo no debería hacerse cargo de nada más; y ningún equipo debería tener dos dominios complicados. También se puede preguntar directamente a los equipos, por ejemplo con una encuesta breve sobre lo bien que entienden sus sistemas, la seguridad que sienten cuando están de guardia y cuánto tiempo dedican a las herramientas en lugar de al dominio. Las colas de tickets crecientes, los largos tiempos de incorporación y los traspasos frecuentes a la única persona que entiende un componente son señales indirectas útiles.

Las respuestas arquitectónicas reflejan los tres tipos de carga. Reduce la carga extrínseca con plataformas, automatización, caminos pavimentados y buena documentación. Acota la carga intrínseca trazando los límites de servicios y módulos de modo que la parte del dominio de cada equipo sea coherente y de un tamaño manejable, y situando las especialidades profundas detrás de interfaces de subsistema complicado. Protege la carga pertinente dejando tiempo para aprender, que es donde ayudan los equipos facilitadores.

Las API de equipo y la plataforma viable más delgada

Si los equipos interactúan mediante modos deliberados, cada equipo necesita una cara clara hacia el resto de la organización. Team Topologies la llama API del equipo: todo lo que el equipo ofrece y espera. Incluye la API técnica (endpoints, eventos, bibliotecas y sus reglas de versionado y obsolescencia), pero también la interfaz humana: la documentación y las guías de incorporación, la hoja de ruta y las prioridades actuales del equipo, sus prácticas de trabajo, los canales donde se le puede contactar, cómo solicitarle trabajo y qué tiempos de respuesta esperar.

Poner por escrito una API del equipo es un ejercicio barato y revelador. Obliga a un equipo a decir de qué se hace cargo, lo que suele dejar al descubierto servicios sin propietario y datos en disputa, y ofrece a otros equipos una forma de interactuar sin una reunión. Muchas organizaciones la mantienen como una página breve en el repositorio del equipo, revisada cada trimestre junto con la documentación de arquitectura.

La plataforma merece especial cuidado, porque es donde las buenas intenciones fallan con más frecuencia. Una plataforma construida como un gran proyecto interno, con una larga hoja de ruta y adopción obligatoria, puede fácilmente añadir carga cognitiva en lugar de reducirla. Skelton y Pais recomiendan la plataforma viable más delgada (thinnest viable platform): el

conjunto más pequeño de servicios, documentación y estándares que realmente acelera a sus usuarios. Para una organización pequeña puede ser una página bien mantenida que enumere qué servicios del proveedor de nube deben usar los equipos y cómo; para una grande puede ser una plataforma interna para desarrolladores. La ingeniería de plataformas se trata en profundidad en el Capítulo 10; lo esencial aquí es que una plataforma es un producto cuyos clientes son otros equipos, y debe diseñarse y medirse como tal.

Planos de fractura: por dónde dividir el software y los equipos

Cuando una parte del sistema ha crecido demasiado para un solo equipo, hay que dividirla, y la división debe funcionar tanto para el código como para los equipos. Team Topologies llama planos de fractura a las junturas naturales por las que puede dividirse el software. El libro enumera varios, y el primero suele ser el mejor: un contexto delimitado del diseño guiado por el dominio, porque un límite que sigue el lenguaje y las reglas del negocio tiende a mantener juntos los cambios relacionados.

Los demás son útiles cuando coinciden con una diferencia real en cómo deben tratarse las partes del sistema. Dividir solo por un límite tecnológico (por ejemplo, un equipo móvil y un equipo de backend) suele ser la opción más débil, porque entonces la mayoría de los cambios lo cruzan, que es la estructura de equipos por capas sobre la que advierte la ley de Conway.

Planos de fractura con ejemplos de reparto de comida

Plano de fractura	Divide cuando	Ejemplo
Dominio de negocio (contexto delimitado)	Partes del dominio tienen su propio lenguaje, sus propias reglas y sus propios expertos	Pedidos, alta de restaurantes y gestión de repartidores como contextos separados
Cumplimiento normativo	Una parte está sujeta a normas que no deberían recaer sobre el resto	Gestión de pagos con tarjeta mantenida en un alcance pequeño y auditado para PCI DSS
Ritmo de cambio	Algunas partes cambian a diario y otras una vez al año	Las promociones y campañas cambian cada semana; las reglas de facturación cambian rara vez
Ubicación del equipo	Los equipos trabajan en husos horarios con poco solapamiento	Un equipo de operaciones de repartidores en otro país se hace cargo del alta de repartidores
Riesgo	Algunas partes toleran experimentos y otras no	Experimentos de recomendación separados de la aceptación de pedidos
Aislamiento del rendimiento	Una parte tiene necesidades de carga o de latencia muy distintas	Actualizaciones en vivo de la ubicación de los repartidores separadas de la consulta de cartas
Tecnología	Una parte necesita un stack tecnológico realmente distinto	Una integración heredada con las tabletas de los restaurantes que solo funciona en hardware antiguo
Perfiles de usuario	Los distintos grupos de usuarios tienen necesidades y recorridos distintos	Clientes, personal de restaurante y repartidores tienen cada uno su propia app y su propio equipo

La ley de Conway, la maniobra inversa y sus límites

Team Topologies es, en la práctica, un conjunto de herramientas para la maniobra inversa de Conway que describió la Sección 3.7: decide la arquitectura que quieres y después da forma a los equipos y a los modos de interacción para que la ley de Conway empuje el sistema hacia ella. Sus aportaciones son prácticas. Los tipos de equipo dicen qué clase de equipo debe hacerse cargo de cada parte; los modos de interacción dicen qué límites entre equipos deben ser finos (X como servicio) y cuáles son temporalmente gruesos (colaboración); la carga cognitiva dice cuánto puede tener a su cargo cada equipo; los planos de fractura dicen por dónde cortar.

Los límites señalados en la Sección 3.7 siguen vigentes y conviene repetirlos en este contexto. Reorganizar los equipos no reestructura una base de código fuertemente acoplada: durante meses después de una reorganización, los equipos nuevos lucharán contra la arquitectura antigua, y por eso los cambios en los equipos y los cambios en la arquitectura deben planificarse como un único programa. Los límites que se consolidan en equipos demasiado pronto encarecen los errores, por lo que muchas organizaciones mantienen los límites inciertos dentro de un solo equipo o de un solo monolito modular hasta tener pruebas. Y las reorganizaciones frecuentes destruyen el conocimiento acumulado por los equipos estables, del que depende el enfoque.

Los profesionales tampoco se ponen de acuerdo sobre lo literalmente que hay que aplicar el marco. Sus defensores valoran el vocabulario compartido, que convierte quejas vagas (siempre estamos esperando al equipo de datos) en preguntas que se pueden diseñar (¿qué modo de interacción queremos con el equipo de datos y qué lo convertiría en X como servicio?). Sus críticos advierten de que los cuatro tipos a menudo se aplican como etiquetas a una organización que no cambia, de que los equipos reales a veces mezclan tipos (un equipo de plataforma que también se hace cargo de un subsistema complicado) y de que el marco dice poco sobre financiación, estructura de gestión o incentivos. Ambas observaciones son justas: los tipos son una ayuda para el diseño, no un organigrama, y los propios autores los presentan como punto de partida para un ajuste continuo.

DE DÓNDE VIENE

De dónde viene el pensamiento sociotécnico

La expresión sistema sociotécnico procede del Tavistock Institute de Londres. En 1951, Eric Trist y Ken Bamforth publicaron un estudio sobre la minería del carbón británica que mostraba que un método de extracción nuevo y más mecanizado había reducido la productividad porque desarticulaba pequeños grupos de trabajo autoorganizados. Su conclusión, que los sistemas técnicos y sociales deben diseñarse juntos, es la misma idea a la que llegó Conway para el software en 1968 y la que aplica hoy Team Topologies.

EJEMPLO PRÁCTICO

Ejemplo práctico: rediseñar los equipos de la plataforma de reparto de comida en torno a flujos

La plataforma de reparto de comida ha crecido hasta 38 ingenieros en cinco equipos organizados por capa técnica: apps móviles, frontend web, API de backend, base de datos y operaciones. El tiempo de entrega desde la idea hasta producción es de unas seis semanas, los lanzamientos se coordinan cada dos semanas entre todos los equipos, y cada uno de los tres últimos incidentes duró varias horas porque nadie se hacía cargo de un recorrido del cliente completo. El arquitecto y los responsables de ingeniería rediseñan la organización en cinco pasos.

1. Mapear los flujos	Un event storming y una revisión de las últimas 50 funcionalidades muestran tres flujos principales de cambio: pedidos de clientes (cartas, carrito, proceso de pago, seguimiento), restaurantes asociados (alta, cartas, horarios de apertura, liquidaciones) y reparto (alta de repartidores, asignación, seguimiento en vivo). 44 de las 50 funcionalidades correspondían principalmente a uno de ellos.
2. Comprobar los planos de fractura del software	Pedidos, restaurantes y reparto ya son contextos delimitados separados, con sus propios servicios y bases de datos (Sección 11.1), así que el código puede acompañar el cambio. La asignación de repartidores usa un solucionador de optimización que solo entienden tres ingenieros: un candidato a subsistema complicado. Los pagos con tarjeta se quedan en Pedidos, pero en un módulo pequeño auditado por separado (plano de fractura regulatorio).
3. Dimensionar los equipos según la carga cognitiva	Tres equipos alineados al flujo de 7 personas (cada uno con competencias de móvil, web, backend, pruebas y operaciones). Un equipo de subsistema complicado de 5 para la optimización del despacho (los tres especialistas más dos ingenieros que aprenden el dominio). Un equipo de plataforma de 9 que se hace cargo de los pipelines, el entorno de ejecución de contenedores, la observabilidad y el intermediario de eventos. Un pequeño equipo facilitador de 3 para las prácticas de fiabilidad. Total: 38.
4. Elegir los modos de interacción	Los equipos alineados al flujo consumen la plataforma como X como servicio desde el primer día, usando sus pipelines existentes. Reparto y Despacho colaboran durante un trimestre para rediseñar la interfaz de despacho y después pasan a X como servicio. El equipo facilitador facilita a Pedidos durante ocho semanas (objetivos de nivel de servicio, runbooks de guardia) y después pasa a Restaurantes.
5. Publicar las API de equipo y medir	Cada equipo redacta una API del equipo de una página, cada servicio y cada tabla de base de datos tiene exactamente un equipo propietario, y los responsables siguen cada trimestre el tiempo de entrega, la frecuencia de despliegue, los tickets entre equipos por funcionalidad y una encuesta sobre la confianza en las guardias.

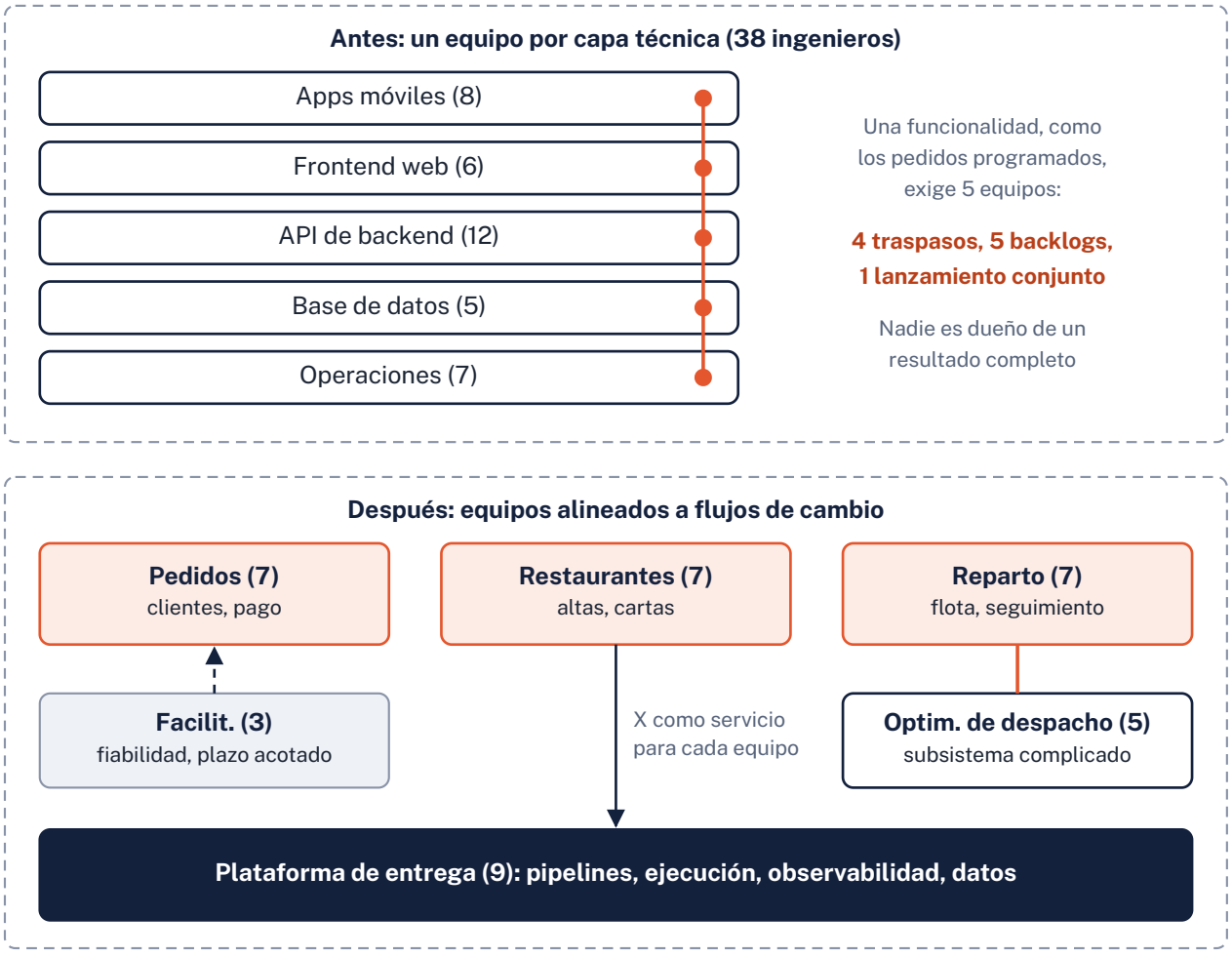
Resultado que buscar	Al cabo de dos trimestres, la mayoría de las funcionalidades deberían entregarse por un solo equipo sin un lanzamiento coordinado; un trabajo entre equipos persistente señalaría un límite que hay que revisar.

El rediseño parte de los flujos de cambio y de los límites existentes del software, no de las líneas jerárquicas actuales.

Los especialistas se concentran solo donde el dominio lo exige (el despacho), y la plataforma existe para reducir la carga de los equipos alineados al flujo, no para controlarlos.

Los modos de interacción vienen con fecha de fin, así que se espera que la organización evolucione en lugar de quedar fijada en un organigrama nuevo.

Antes y después



Los equipos por capa obligan a que cada funcionalidad pase por todos los equipos; los equipos alineados al flujo, apoyados por una plataforma, un equipo facilitador y un equipo especializado, pueden entregar solos la mayoría de las funcionalidades.

ATENCIÓN

Errores comunes

- Renombrar los equipos existentes con las etiquetas de Team Topologies mientras la propiedad, la financiación y la interacción siguen igual.
- Un equipo de plataforma que trabaja mediante tickets y aprobaciones, de modo que los equipos alineados al flujo tienen que esperarlo: eso es un cuello de botella, no una plataforma.
- Un equipo facilitador que acaba haciendo permanentemente el trabajo de otros equipos, lo que crea dependencia en lugar de capacidad.
- Una colaboración sin fecha de fin, que fusiona sin que nadie lo note la carga cognitiva de dos equipos.
- Dividir los equipos según líneas tecnológicas (frontend, backend, base de datos) y esperar un flujo rápido.
- Trazar primero los límites de los servicios y asignar propietarios después, lo que deja servicios huérfanos o de propiedad compartida.

Preguntas frecuentes

¿Los equipos de componente son siempre un error?
No siempre. Un equipo que se hace cargo de un componente compartido puede ser la opción correcta cuando el componente requiere una especialización profunda (un subsistema complicado) o se ofrece como un verdadero producto de autoservicio (parte de una plataforma). Se convierte en un problema cuando la mayoría de las funcionalidades necesitan cambios en el componente y los equipos alineados al flujo tienen que hacer cola para conseguirlos.

¿Dónde encajan los arquitectos en este modelo?
Normalmente no como un equipo aparte que hace de guardián. Los arquitectos suelen actuar como un equipo facilitador, ayudando a los equipos alineados al flujo a tomar buenas decisiones y manteniendo visibles las cualidades de todo el sistema, y participan en el diseño de la propia estructura de equipos. La Sección 11.4 explica cómo puede funcionar la gobernanza sin compuertas.

¿Con qué frecuencia deberían cambiar los límites de los equipos?
Rara vez y de forma deliberada. Se espera que los modos de interacción cambien cada pocos meses; la composición de los equipos y su propiedad deberían mantenerse estables durante mucho más tiempo, porque es en los equipos estables donde se acumula el conocimiento del dominio y del sistema.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **Nombra tres cosas que contiene una API del equipo además de sus interfaces de código.**

Por ejemplo, su documentación y sus guías de incorporación, su hoja de ruta y sus prioridades, sus prácticas de trabajo, los canales donde se le puede contactar y cómo solicitarle trabajo.

▼ **¿Por qué se espera que los modos de interacción cambien con el tiempo?**

Porque la colaboración es adecuada para descubrir un límite nuevo y, una vez que se entienden el límite y la interfaz, los equipos deberían asentarse en X como servicio; los modos siguen el grado de madurez del límite.

▼ **¿Qué es la plataforma viable más delgada y por qué aspirar a ella?**

El conjunto más pequeño de servicios, documentación y estándares que realmente acelera a los equipos que la usan; una plataforma más grande corre el riesgo de añadir carga cognitiva en lugar de reducirla.

▼ **¿Por qué una división en equipo de frontend y equipo de backend suele ser un plano de fractura débil?**

Porque la mayoría de las funcionalidades la cruzan, así que cada cambio necesita a ambos equipos, traspasos y lanzamientos coordinados, que es la estructura por capas que la ley de Conway convierte en un sistema por capas y acoplado.

▼ **Inténtalo: En una empresa de venta de entradas, asigna un tipo de equipo y un modo de interacción a: un equipo dueño del recorrido de compra; un pequeño grupo de especialistas en algoritmos de asignación de asientos; un grupo que proporciona herramientas de CI y despliegue; dos expertos que ayudan a los equipos a aprender sobre pruebas.**

Recorrido de compra: alineado al flujo, el predeterminado. Especialistas en asignación de asientos: subsistema complicado, que ofrece X como servicio al equipo del recorrido, con una colaboración de tiempo limitado en las interfaces nuevas. Grupo de herramientas: plataforma, que ofrece X como servicio y la mantiene como plataforma viable más delgada. Expertos en pruebas: facilitador, que usa el modo de facilitación y aspira a marcharse una vez transferida la habilidad. Comprueba que la carga cognitiva de cada equipo sea manejable, y recuerda que cambiar el nombre de los equipos no cambia por sí solo el código acoplado.

FUENTES

- Team Topologies: key concepts: https://teamtopologies.com/key-concepts
- Martin Fowler: Team Topologies (2023): https://martinfowler.com/bliki/TeamTopologies.html
- Melvin Conway: How Do Committees Invent? (1968): https://melconway.com/Home/Committees_Paper.html

SECCIÓN 11.6 DE 11.8 · AVANZADO · 29 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Modernizar sistemas heredados: strangler fig y migración incremental

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.7 Deuda técnica
- 5.7 Diseño guiado por el dominio
- 7.1 Propiedad de los datos

Términos anteriores que utiliza: Deuda técnica, Sistema de registro (system of record), API gateway (puerta de enlace de API)

LA IDEA EN BREVE

Imagina un pueblo que necesita un puente nuevo. Derribar el antiguo y construir uno de reemplazo dejaría al pueblo aislado durante años. En lugar de eso, los ingenieros construyen el puente nuevo junto al antiguo, lo abren carril a carril, desvían el tráfico poco a poco y solo derriban el puente antiguo cuando ya nada lo usa. Si un carril nuevo tiene un problema, el tráfico simplemente vuelve al puente antiguo durante un tiempo. Modernizar software antiguo funciona mejor de la misma manera: sustitúyelo pieza a pieza mientras sigue funcionando, para que los clientes nunca pierdan el servicio del que dependen.

LO QUE NECESITAS SABER

- Un sistema heredado sigue aportando valor, pero es difícil, arriesgado o caro de cambiar; moderniza por motivos de negocio, como un cambio bloqueado, la pérdida de soporte, la escasez de competencias o el costo, no por moda.
- Las reescrituras big-bang fracasan porque el valor llega tarde, el objetivo se mueve, se pierde comportamiento oculto y la migración de datos y el cambio definitivo (cutover) quedan para el final; el ensayo de Spolsky de 2000 y la migración de TSB de 2018 son advertencias clásicas.
- Las siete R (retirar, retener, realojar, reubicar, recomprar, cambiar de plataforma, y refactorizar o rediseñar la arquitectura) dan a cada aplicación de una cartera una estrategia deliberada; reconstruir es solo una opción.
- El strangler fig (Fowler, 2004) hace pasar el tráfico por una fachada y traslada el comportamiento porción a porción a componentes nuevos hasta que el sistema heredado puede apagarse; branch by abstraction hace lo mismo dentro de una única base de código.

- Las ejecuciones en paralelo comparan en producción los resultados antiguos y nuevos mientras el resultado antiguo sigue siendo el que cuenta; el dark launching ejercita rutas nuevas con carga real sin que los usuarios las vean.
- Los datos necesitan un único sistema de registro en cada etapa; la captura de datos de cambios, la intercepción de eventos y las capas anticorrupción mantienen sincronizados los sistemas, y el histórico se migra por cohortes ensayadas y conciliadas.
- La arquitectura de transición es un costo deliberado y temporal; una migración solo termina cuando se han eliminado el sistema antiguo y el código de transición.

Qué hace que un sistema sea heredado

Un sistema heredado (legacy) no es simplemente un sistema antiguo. Es un sistema que sigue haciendo algo valioso, a menudo algo sin lo que la organización no podría funcionar, pero que se ha vuelto difícil, arriesgado o caro de cambiar. Michael Feathers, en *Working Effectively with Legacy Code* (2004), dio una definición deliberadamente provocadora: el código heredado es código sin pruebas, porque sin pruebas cada cambio es una apuesta. Otras causas habituales son la escasez de competencias (un lenguaje o una plataforma que ya pocos conocen), la tecnología sin soporte (un sistema operativo o una base de datos que han superado el fin de soporte de su proveedor), el conocimiento perdido (las personas que entendían el diseño se han ido) y la deuda técnica acumulada en la propia estructura, el tipo costoso descrito en la Sección 1.7.

Vale la pena decirlo con claridad: la mayoría de los sistemas heredados son sistemas de éxito. Sobrevivieron porque funcionaban y generaban dinero. Ese éxito es también la razón por la que son difíciles de sustituir: años de uso real han incorporado miles de pequeñas decisiones, correcciones y reglas de negocio que no están escritas en ninguna parte salvo en el código y en los hábitos de las personas que lo usan.

Por tanto, la modernización necesita un motivo que pueda expresarse en términos de negocio. Entre los buenos motivos están: el costo o el riesgo del cambio está bloqueando objetivos de negocio importantes; la plataforma está perdiendo el soporte del proveedor o los parches de seguridad; las competencias para mantenerla están dejando de estar disponibles; los costos de operación son desproporcionados respecto al valor; o el sistema no puede satisfacer nuevas exigencias de escala, disponibilidad o regulación. Entre los motivos débiles están la moda, el aburrimiento y la creencia de que la tecnología nueva es automáticamente mejor. La Sección 11.8 muestra cómo convertir un buen motivo en un caso de negocio.

Por qué fracasan las reescrituras

La respuesta obvia a un sistema antiguo problemático es empezar de nuevo. En abril de 2000, Joel Spolsky publicó un ensayo, "Things You Should Never Do, Part I", en el que sostenía que la decisión de reescribir desde cero es uno de los peores errores estratégicos que puede cometer una empresa de software. Su ejemplo era Netscape, cuyo navegador pasó casi tres años entre dos versiones principales mientras se reescribía el código, tiempo durante el cual los competidores se quedaron con el mercado. Su argumento central era sobre el conocimiento: el código antiguo parece desordenado en parte porque contiene años de correcciones de bugs para situaciones que nadie recuerda, y tirarlo supone tirar también esas correcciones.

Hoy se entienden bien las razones por las que fracasa una reescritura big-bang. El valor solo llega al final, así que el negocio espera meses o años y pierde la paciencia. El sistema antiguo sigue cambiando mientras se construye el nuevo, así que el objetivo se mueve y la organización o bien congela el sistema antiguo (y el negocio), o bien paga por hacer cada cambio dos veces. Se pasa por alto comportamiento oculto, porque nadie puede enumerar todas las reglas que impone el sistema antiguo. Las estimaciones son optimistas, porque las funcionalidades visibles son una fracción del trabajo real. La migración de datos y el cambio definitivo se dejan para el final, cuando son más difíciles y arriesgados. Y el equipo se divide en un grupo prestigioso del sistema nuevo y un grupo desmoralizado del sistema antiguo. El efecto del segundo sistema de Fred Brooks, de *The Mythical Man-Month* (1975), añade otro riesgo: los diseñadores que sustituyen un sistema que conocen tienden a cargar el nuevo con todas las ideas que el antiguo no podía acomodar.

Las grandes migraciones que cambian de sistema en un solo paso también pueden fracasar de forma muy pública. En abril de 2018, el banco británico TSB trasladó los datos y servicios de sus clientes a una nueva plataforma informática; la plataforma tuvo graves problemas de inmediato, todas sus sucursales y una proporción significativa de sus 5.2 millones de clientes se vieron afectadas, y el banco no volvió a funcionar con normalidad hasta diciembre de 2018. En diciembre de 2022, la Financial Conduct Authority y la Prudential Regulation Authority impusieron a TSB multas por un total de £48.65 millones, al concluir que no había planificado, organizado ni controlado adecuadamente la migración ni gestionado los riesgos derivados de su proveedor externo.

No todos los profesionales están de acuerdo en que las reescrituras sean siempre un error. Una reescritura puede ser la opción correcta cuando el sistema es pequeño, su comportamiento está bien especificado y cubierto por pruebas, la plataforma está realmente en un callejón sin salida, o el propio negocio ha cambiado tanto que el comportamiento antiguo ya no se quiere. Incluso entonces, las reescrituras más seguras se entregan de forma incremental, con los sistemas antiguo y nuevo funcionando uno junto al otro, que es el tema del resto de esta sección.

Estrategias de migración: las R

Cuando una organización revisa toda una cartera de aplicaciones, por ejemplo antes de abandonar un centro de datos, necesita un vocabulario para decidir qué hacer con cada una. Gartner describió cinco opciones de este tipo a principios de la década de 2010. En 2016, Stephen Orban, de AWS, publicó seis estrategias, y AWS añadió más tarde una séptima, lo que dio lugar a las siete R (Seven Rs) que se usan hoy en su guía de migración. Otros proveedores de nube y consultoras usan listas parecidas con nombres ligeramente distintos. La lista es más útil como recordatorio de que reconstruir es solo una opción entre varias, y normalmente la más cara.

Las siete R (según AWS Prescriptive Guidance)

Estrategia	Qué significa	Encaja cuando	Cambio de arquitectura
Retirar (retire)	Apagar la aplicación y archivar sus datos	Nadie la usa, o su función está duplicada en otro lugar	Ninguno; eliminar las dependencias de ella
Retener (retain)	Dejarla donde está por ahora y revisarla más adelante	No hay un motivo urgente para moverla, u otro trabajo debe ir primero	Ninguno por ahora
Realojar (rehost, lift and shift)	Moverla sin cambios a una infraestructura nueva, como máquinas virtuales en la nube	El principal impulsor es abandonar rápidamente un centro de datos	Ninguno; los problemas se mudan con ella
Reubicar (relocate)	Mover plataformas enteras, como un parque virtualizado, sin cambiar las aplicaciones	Existe una plataforma compatible en el destino	Ninguno
Recomprar (repurchase)	Sustituirla por un producto, a menudo software como servicio	La capacidad es genérica y existe un buen producto	Integración y migración de datos; pérdida de personalización
Cambiar de plataforma (replatform)	Moverla con unas pocas optimizaciones específicas, como una base de datos gestionada	Hay mejoras baratas disponibles sin rediseñar	Pequeño y local
Refactorizar o rediseñar la arquitectura (refactor o re-architect)	Rediseñarla, a menudo de forma incremental, para satisfacer nuevas necesidades de calidad o de negocio	La capacidad es central, cambia a menudo y su estructura bloquea el cambio	Grande; usar los patrones incrementales que se describen a continuación

GUÍA DE DECISIÓN

Guía de decisión: elegir una estrategia para una aplicación

Hazte estas preguntas para cada aplicación de la cartera, en este orden. Las respuestas salen de los datos de uso, del responsable de negocio y de la arquitectura, no de las preferencias tecnológicas.

Q1 ¿Alguien sigue necesitando lo que hace esta aplicación?

- **Sí** Pasa a la siguiente pregunta.
- **No** Retírala, después de archivar los datos que la ley te obligue a conservar.

Q2 ¿Hay un producto que cubra la necesidad, para una capacidad que no diferencia al negocio?

- **Sí** Plantéate recomprar, y presupuesta la integración y la migración de datos.
- **No** Pasa a la siguiente pregunta.

Q3 ¿Hay un motivo urgente para actuar ahora, como el fin del soporte, la salida de un centro de datos, el costo o un cambio que está bloqueado?

- **Sí** Pasa a la siguiente pregunta.
- **No** Retenla y fija una fecha para revisar la decisión.

Q4 ¿El problema principal es dónde se ejecuta y no cómo está construida?

- **Sí** Realójala o reubícala, o cámbiala de plataforma cuando un servicio gestionado ofrezca mejoras baratas.
- **No** Pasa a la siguiente pregunta.

Q5 ¿Es una capacidad central cuya estructura está bloqueando cambios frecuentes y valiosos?

- **Sí** Refactorízala o rediseña su arquitectura de forma incremental, con un strangler fig y branch by abstraction.
- **No** Cámbiala de plataforma y contenla: añade pruebas, corrige los peores hotspots y evita un gran rediseño.

El strangler fig

Martin Fowler bautizó el patrón incremental más usado en honor a las higueras estranguladoras que vio en las selvas tropicales de Queensland en 2001: enredaderas que germinan en las ramas de un árbol hospedador, crecen hacia el suelo y rodean poco a poco al hospedador hasta que este muere y la higuera se sostiene por sí sola. Lo describió por primera

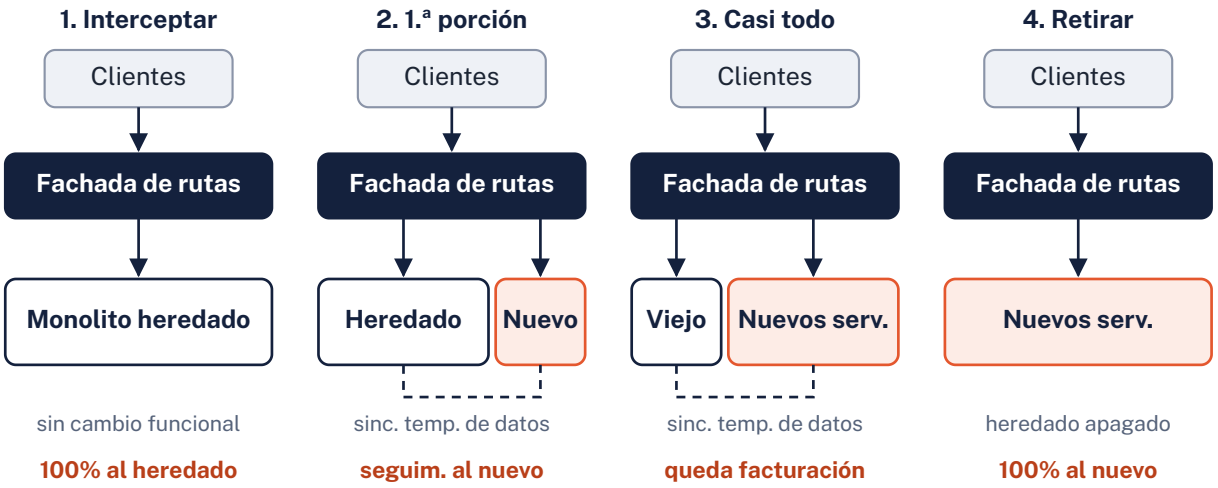
vez como strangler application en 2004, más tarde lo rebautizó como strangler fig application y revisó a fondo el artículo en 2024. La versión de software, normalmente llamada simplemente strangler fig (higuera estranguladora), mantiene en funcionamiento el sistema antiguo mientras crecen a su alrededor componentes nuevos que asumen su comportamiento pieza a pieza, hasta que el sistema antiguo ya no hace nada y puede apagarse.

El mecanismo tiene tres partes. Primero, se coloca una capa de interceptación delante del sistema heredado: un proxy inverso, un API gateway o una fachada de enrutamiento por la que pasan todas las solicitudes. Al principio lo envía todo al sistema antiguo y no cambia nada. Segundo, se elige una porción de comportamiento a lo largo de una juntura (un lugar donde el comportamiento puede redirigirse sin editar mucho código antiguo, en palabras de Feathers), se reconstruye como un componente nuevo y se activa cambiando una regla de enrutamiento, normalmente primero para una pequeña parte del tráfico. Tercero, el proceso se repite hasta que el sistema heredado no gestiona nada, momento en el que se retira. En todo momento, la capa de enrutamiento hace que cada paso sea reversible: si el componente nuevo se comporta mal, se vuelve a cambiar la regla.

El código que permite que lo antiguo y lo nuevo coexistan (la fachada de enrutamiento, los adaptadores, los trabajos de sincronización de datos) es arquitectura de transición: software construido para tirarse. Los equipos a menudo se resisten a escribirlo porque parece un desperdicio. El argumento de Fowler, y la experiencia que lo respalda, es que este costo compra un menor riesgo y un valor mucho más temprano, lo que casi siempre es un buen intercambio. El peligro real es el contrario: una migración que se estanca a medio camino y deja ambos sistemas y todo el código de transición en producción indefinidamente. Por eso cada porción debe eliminar algo del sistema antiguo, y el progreso debe medirse por lo que se ha apagado, no solo por lo que se ha construido.

Ian Cartwright, Rob Horn y James Lewis, de Thoughtworks, han documentado una familia de técnicas relacionadas como *Patterns of Legacy Displacement* en martinfowler.com (desde 2021). Plantean el trabajo como cuatro actividades que no siguen una secuencia fija: entender los resultados que quieres, dividir el problema en partes, entregar esas partes de forma incremental y cambiar la organización para que el sistema nuevo no se deteriore como lo hizo el antiguo. También cuestionan el instinto de reproducir todas las funcionalidades: puede que gran parte del comportamiento de un sistema heredado ya no se quiera, y averiguarlo es más barato que reconstruirlo.

Una migración strangler fig en cuatro etapas



Una fachada de enrutamiento permite trasladar el tráfico porción a porción del sistema heredado a los servicios nuevos, con una sincronización de datos temporal, hasta que el sistema heredado puede apagarse.

Branch by abstraction (ramificación por abstracción)

El strangler fig funciona en el borde de un sistema, donde las solicitudes pueden enrutarse. Algunas sustituciones deben hacerse en lo más profundo de una base de código: cambiar un framework de persistencia, un motor de precios o una capa de mensajería de fabricación propia que usan cientos de puntos de llamada. Branch by abstraction, un nombre que introdujo Paul Hammant mientras defendía el desarrollo basado en trunk y que Martin Fowler describió en 2014, lo hace sin una rama de control de versiones de larga duración. El sistema sigue siendo lanzable en cada paso, así que el cambio puede repartirse a lo largo de semanas o meses mientras continúa otro trabajo en la misma línea principal.

Branch by abstraction, paso a paso

1. **Introduce una abstracción**
   Crea una interfaz delante del componente que se va a sustituir, con una implementación que simplemente delega en el código existente. El comportamiento no cambia en nada.

2. **Mueve los llamadores a la abstracción**
   Cambia los puntos de llamada, poco a poco, para que usen la interfaz en lugar del componente antiguo directamente. Cada cambio es pequeño y se lanza con normalidad.

3. **Construye la implementación nueva**
   Escribe la sustitución detrás de la misma interfaz y pruébala con las mismas pruebas e, idealmente, contra el comportamiento de producción en una ejecución en paralelo.

4 **Haz el cambio de forma gradual**
Usa una feature flag o la configuración para enviar a algunos llamadores o a algunos clientes a la implementación nueva, aumentando la proporción a medida que crece la confianza.

5 **Elimina la implementación antigua**
Cuando nada use ya el código antiguo, bórralo, y elimina también la abstracción si ya no justifica su lugar.

Demostrar la equivalencia: ejecución en paralelo y dark launching

La pregunta más difícil de cualquier migración es si la implementación nueva se comporta realmente como la antigua. Una ejecución en paralelo la responde con evidencia de producción: ambas implementaciones reciben las mismas entradas, la antigua sigue siendo la que cuenta y su respuesta es la que se devuelve al usuario, y los dos resultados se comparan y se registra cualquier diferencia. GitHub publicó su biblioteca Scientist en 2016 exactamente con este fin: ejecuta un control y un candidato uno junto al otro e informa de las discrepancias; existen herramientas parecidas para muchos lenguajes, y la técnica es sencilla de escribir a mano, como muestra el esbozo siguiente.

El dark launching (lanzamiento oculto), que Fowler describe como llamar a un comportamiento de backend nuevo en producción sin que los usuarios vean sus efectos, está estrechamente relacionado. Se usa sobre todo para probar el rendimiento y la estabilidad de la ruta nueva con carga real antes de que ningún usuario dependa de ella, mientras que una ejecución en paralelo se centra en la corrección. Ambos se diferencian de un canary release, en el que una pequeña parte de los usuarios recibe de verdad el comportamiento nuevo.

Ambas técnicas tienen aristas peligrosas. El candidato no debe tener efectos secundarios visibles desde fuera: no debe enviar un segundo correo electrónico, cobrar dos veces una tarjeta ni escribir en la base de datos compartida, así que las ejecuciones en paralelo son más fáciles para lecturas y cálculos, y las escrituras necesitan un sandbox o un almacén aparte. Las comparaciones deben ignorar las diferencias legítimas, como las marcas de tiempo, el orden y el formato de los números de coma flotante, o el informe de discrepancias se ahoga en ruido. Y ejecutar ambas rutas consume capacidad, así que a menudo se limita la proporción del tráfico que se muestrea.

```
def quote_delivery_fee(order):
    old = legacy_pricing.quote(order)          # still authoritative
    try:
        new = new_pricing.quote(order)         # must have no side effects
        if new.amount_pence ≠ old.amount_pence:
            metrics.increment("pricing.mismatch")
            log.info("pricing mismatch", order_id=order.id,
                     old=old.amount_pence, new=new.amount_pence)
        else:
            metrics.increment("pricing.match")
    except Exception:
        metrics.increment("pricing.candidate_error")   # never hurts the user
    return old
```

Una ejecución en paralelo mínima: se sigue devolviendo el resultado heredado, mientras que la respuesta del candidato se compara y se cuentan las discrepancias.

Los datos: la parte más difícil

Enrutar las solicitudes es relativamente fácil; mover los datos y mantenerlos consistentes mientras dos sistemas comparten la responsabilidad sobre ellos es donde más migraciones tropiezan. La primera regla es la del Capítulo 7 sobre la propiedad de los datos: para cada tipo de datos, en cada etapa de la migración, el equipo debe poder decir qué sistema es el sistema de registro, qué copias existen y cuán desactualizada puede estar cada una. El diagrama siguiente muestra el cambio típico: al principio el sistema heredado es dueño de los datos y el componente nuevo lee una copia; más adelante el componente nuevo pasa a ser el dueño y el sistema heredado recibe una copia para los módulos que todavía la necesitan.

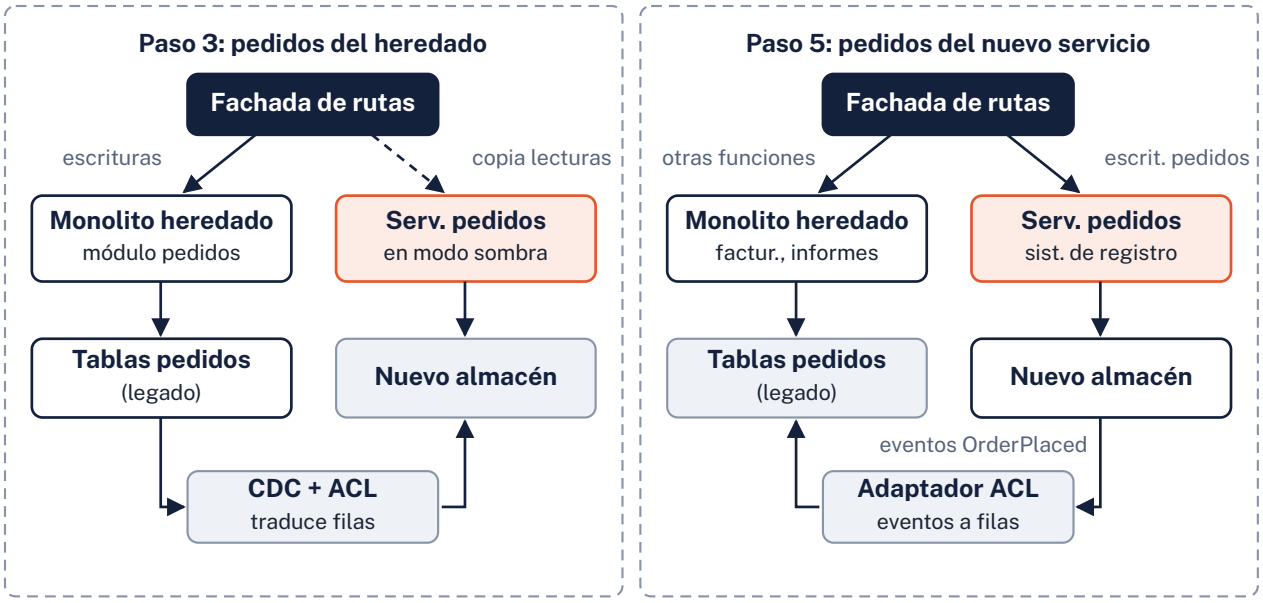
La captura de datos de cambios (CDC) es la forma habitual de copiar datos de una base de datos heredada sin cambiar el código heredado: una herramienta como Debezium lee el registro de transacciones de la base de datos y publica cada cambio como un evento, el mismo mecanismo que la Sección 6.7 usó para retransmitir una bandeja de salida (outbox). La intercepción de eventos, uno de los patrones de desplazamiento de sistemas heredados, actúa en el otro extremo: captura las solicitudes o mensajes que cambian el estado en su camino hacia el sistema heredado y reenvía copias (o, más adelante, los originales) al componente nuevo. En cualquier caso, nunca permitas que ambos sistemas acepten escrituras sobre los mismos datos de forma independiente con una sincronización bidireccional, salvo que no tengas alternativa; en ese caso los conflictos están casi garantizados.

Los datos heredados rara vez coinciden con el modelo nuevo. Los códigos de estado son enteros con significados no documentados, una tabla sirve para tres propósitos y las fechas se guardan en hora local. Una capa anticorrupción, el patrón del diseño guiado por el dominio para

proteger un modelo del de un vecino, traduce entre ambos para que las peculiaridades del sistema heredado no se filtren al código nuevo. Forma parte de la arquitectura de transición y debe desaparecer con el sistema heredado.

La migración final de los datos, mover el histórico y hacer el cambio definitivo, merece su propio plan. Perfila primero los datos heredados para encontrar duplicados, registros huérfanos y valores no válidos. Escribe la correspondencia de forma explícita y pruébala con una copia completa del tamaño de producción. Concilia después de cada ejecución con recuentos de filas, sumas de las columnas de importes y checksums, no con comprobaciones puntuales. Ensaya el cambio definitivo, cronométralo y pon por escrito el rollback. Siempre que sea posible, migra por cohortes (una región, un segmento de clientes, una línea de producto) en lugar de todo a la vez, para que un problema afecte a una fracción de los clientes y te enseñe algo antes de la siguiente cohorte.

El sistema de registro cambia durante la migración



Al principio, la captura de datos de cambios copia las filas heredadas en el almacén nuevo; una vez que el servicio nuevo es dueño de los pedidos, sus eventos se traducen de vuelta a filas heredadas para los módulos que aún no se han migrado.

SI ESTO ES NUEVO PARA TI

Paso a paso: por qué dos sistemas no deben ser a la vez dueños de los pedidos

La idea que requiere más atención aquí es mantener un único sistema de registro (system of record) para cada tipo de datos mientras la propiedad pasa del sistema antiguo al nuevo. El Capítulo 7 enseñó la regla; lo nuevo aquí es que el dueño cambia mientras ambos sistemas siguen teniendo usuarios.

- A mitad de camino, supón que tanto el módulo de pedidos heredado como el nuevo servicio de pedidos aceptan cambios en los pedidos, con un trabajo que copia los cambios en ambos sentidos cada pocos segundos.
- Un cliente cambia en la app la dirección de entrega de un pedido; la fachada de enrutamiento envía este cambio al nuevo servicio de pedidos. Segundos después, un agente de soporte cancela ese mismo pedido en las herramientas de soporte, que siguen usando el módulo heredado.
- Si el trabajo copia pedidos completos, gana la copia que llegue la última: puede que se deshaga la cancelación o que se pierda el cambio de dirección. Con pedidos que cambian todo el día, estos choques están casi garantizados.
- La solución: en cada etapa, solo el sistema de registro acepta cambios en los pedidos, y el otro guarda una copia de solo lectura. Al principio el dueño es el módulo heredado, y la captura de datos de cambios transmite al servicio nuevo cada cambio que hace.
- Más adelante se traspasa la propiedad: el servicio nuevo acepta los cambios, y sus eventos se traducen de vuelta a filas heredadas para los módulos que aún no se han migrado. El traspaso es un paso planificado con un rollback preparado, nunca un periodo en el que ambos mandan; el ejemplo práctico de más abajo lo hace ciudad por ciudad, de modo que cada pedido tiene siempre un único dueño.
- **La lección:** para cada tipo de datos, en cada etapa, nombra el único sistema que es su dueño y en qué sentido fluyen las copias. Permitir que ambos sistemas acepten cambios en los mismos datos es un último recurso.

Lecciones de Kill It with Fire

Kill It with Fire: Manage Aging Computer Systems (and Future Proof Modern Ones) (No Starch Press, 2021), de Marianne Bellotti, se basa en su experiencia modernizando sistemas gubernamentales y comerciales, incluidos parques de mainframes. Varios de sus temas merecen trasladarse a cualquier modernización.

La modernización debe justificarse por el valor que aporta y el riesgo que elimina, no por la antigüedad de la tecnología; una tecnología antigua que se entiende bien y funciona puede ser la parte menos arriesgada del parque de sistemas. Las elecciones tecnológicas van por ciclos,

así que un equipo que sustituye un sistema solo porque su diseño ha pasado de moda probablemente esté construyendo el sistema heredado de la próxima generación. La mejora incremental del sistema existente suele ser más segura que la sustitución, y una reescritura completa necesita una justificación excepcionalmente sólida. Por último, la modernización es un problema organizativo tanto como técnico: necesita éxitos tempranos y visibles para mantener el impulso y la moral, criterios claros para saber cuándo ha terminado y atención a los incentivos que hicieron que el sistema antiguo se deteriorara, o el nuevo seguirá el mismo camino.

EJEMPLO PRÁCTICO

Ejemplo práctico: estrangular el módulo de pedidos para sacarlo del monolito en seis pasos

Este ejemplo repasa cómo obtuvo la plataforma de reparto de comida el servicio de pedidos separado que se evaluó en la Sección 11.1. En una etapa anterior de su vida, el monolito original de la plataforma contenía un módulo de pedidos que gestionaba la realización, la modificación, la cancelación y el seguimiento de los pedidos. Los cambios en los pedidos se habían convertido en el principal cuello de botella: exigían desplegar todo el monolito, y los picos de los viernes por la tarde ralentizaban todas las demás funcionalidades. Otros catorce módulos, entre ellos los de facturación, informes y las herramientas de atención al cliente, leían directamente las tablas de pedidos. El equipo planificó una migración incremental a un servicio de pedidos separado.

1. Fijar los resultados y encontrar la juntura	Resultados: cambios en los pedidos desplegables a diario y de forma independiente; el triple del pico actual de los viernes sin afectar a otras funcionalidades; ningún pedido perdido ni duplicado. El análisis de los logs de consultas confirma que solo el módulo de pedidos escribe en las tablas de pedidos y que todo el tráfico de pedidos entra por los endpoints /api/orders. El equipo graba varias semanas de solicitudes de producción y las convierte en pruebas de caracterización que capturan el comportamiento actual, peculiaridades incluidas.
2. Insertar la fachada de enrutamiento	Se coloca un API gateway delante del monolito y al principio lo enruta todo hacia él. Lanzado por sí solo, no cambia ningún comportamiento; el equipo comprueba la latencia añadida (unos pocos milisegundos en el percentil 95) y practica el cambio de rutas. Rollback: quitar el gateway del camino.
3. Construir el servicio nuevo en la sombra	El nuevo servicio de pedidos tiene su propio almacén. El sistema heredado sigue siendo el sistema de registro: la captura de datos de cambios copia las filas de pedidos a través de una capa anticorrupción que traduce los códigos de estado enteros y las fechas en hora local al modelo nuevo. Los endpoints de lectura funcionan como una ejecución en paralelo frente a las respuestas del sistema heredado; la tasa de discrepancias empieza en torno al 2% (redondeo de comisiones, husos horarios) y se reduce por debajo del 0.01% antes de seguir adelante.
4. Mover las lecturas	Las lecturas de seguimiento de pedidos se enrutan al servicio nuevo para el 5%, después el 25% y después todo el tráfico, vigilando las tasas de error y la latencia en cada etapa. Rollback: una regla de enrutamiento.

5. Mover las escrituras, ciudad por ciudad	El servicio nuevo pasa a ser el sistema de registro de los pedidos hechos en una ciudad piloto y después en todas las ciudades a lo largo de seis semanas. Publica los eventos OrderPlaced y OrderCancelled mediante una bandeja de salida transaccional (transactional outbox); un adaptador de transición los traduce a filas con el formato heredado para que la facturación y los informes sigan funcionando sin cambios. Una conciliación diaria compara los recuentos y los importes totales entre los dos almacenes. Después, esos módulos pasan a los eventos o a la API de pedidos durante los trimestres siguientes.
6. Migrar el histórico y retirar	Tres años de pedidos históricos se copian en lotes limitados y se concilian mediante recuentos y checksums. Cuando los logs de auditoría de la base de datos no muestran ninguna lectura de las tablas de pedidos heredadas durante 30 días, el equipo elimina el adaptador, el pipeline de captura de datos de cambios y las reglas de enrutamiento antiguas, y después borra del monolito el código y las tablas de pedidos. Solo en este punto se completa la migración.

Cada paso entrega algo utilizable o elimina riesgo, y todos los pasos salvo el último tienen un rollback barato.

El sistema de registro de los pedidos no es ambiguo en ninguna etapa, y los datos fluyen en un solo sentido cada vez.

La arquitectura de transición se planifica, se mide y se elimina deliberadamente; el éxito se mide por lo que se ha apagado.

ATENCIÓN

Errores comunes

- Empezar por la parte más difícil y central del sistema en lugar de por una porción que aporte valor y enseñe al equipo a migrar.
- Aspirar a la paridad funcional completa sin preguntar qué funcionalidades antiguas se siguen usando.
- Una sincronización bidireccional en la que ambos sistemas aceptan escrituras sobre los mismos datos.
- Migraciones que se estancan a medio camino y dejan dos sistemas y todo el código de transición funcionando durante años.
- Tratar la migración de datos como una tarea final en lugar de diseñarla, ensayarla y conciliarla desde el principio.
- Congelar todos los cambios en el sistema heredado mientras dura la migración, lo que la convierte en una larga espera para el negocio.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué una reescritura big-bang tiende a perder comportamiento del que dependen los usuarios?**

Porque años de correcciones de bugs y reglas de negocio solo existen en el código antiguo y en los hábitos de los usuarios; nadie puede enumerarlos todos, así que una reescritura a partir de una especificación pasa por alto algunos.

▼ **¿Qué aporta la fachada de enrutamiento a una migración strangler fig?**

Permite trasladar el tráfico porción a porción, y devolverlo si algo sale mal, sin editar el sistema heredado en cada paso.

▼ **¿Qué es la arquitectura de transición y por qué hay que seguir su eliminación?**

Código temporal, como reglas de enrutamiento, adaptadores y trabajos de sincronización, que permite que lo antiguo y lo nuevo coexistan; seguir su eliminación protege frente a una migración que se estanca con ambos sistemas funcionando indefinidamente.

▼ **¿Cuándo puede ser recomprar una opción mejor que refactorizar?**

Cuando la capacidad es genérica en lugar de diferenciadora y existe un buen producto, de modo que reconstruirla supondría invertir esfuerzo donde el negocio no compite.

▼ **Inténtalo: Un sistema de taquilla de 15 años sigue vendiendo entradas, pero los cambios son lentos y arriesgados. El consejo prefiere una reescritura de dos años con un único cambio definitivo (cutover). ¿Qué recomendarías y cómo gestionarías los datos durante la migración?**

Evita el big bang: el valor llega tarde, los requisitos se mueven y el comportamiento oculto se pierde. Usa un strangler fig, encaminando el tráfico a través de una fachada y moviendo una porción cada vez, empezando por algo valioso pero de bajo riesgo, como la navegación por los asientos. Usa ejecuciones en paralelo para comparar resultados mientras el sistema antiguo sigue siendo la autoridad. Cada conjunto de datos necesita un sistema de registro en cada etapa, mantenido en sincronía mediante captura de datos de cambios. Termina eliminando el sistema antiguo.

FUENTES

- Martin Fowler: Strangler Fig Application: https://martinfowler.com/bliki/StranglerFigApplication.html
- Joel Spolsky: Things You Should Never Do, Part I (2000): https://www.joelonsoftware.com/2000/04/06/things-you-should-never-do-part-i/

- Cartwright, Horn and Lewis: Patterns of Legacy Displacement: https://martinfowler.com/articles/patterns-legacy-displacement/
- AWS Prescriptive Guidance: migration strategies (the 7 Rs): https://docs.aws.amazon.com/prescriptive-guidance/latest/large-migration-guide/migration-strategies.html
- Martin Fowler: Branch By Abstraction (2014): https://martinfowler.com/bliki/BranchByAbstraction.html

SECCIÓN 11.7 DE 11.8 · AVANZADO · 28 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Liderazgo, influencia y toma de decisiones

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.4 El papel del arquitecto
- 1.6 Decisiones y compromisos
- 4.8 Comunicar

Términos anteriores que utiliza: Registro de decisiones de arquitectura (ADR), Parte interesada, Matriz poder/interés, Kata de arquitectura

LA IDEA EN BREVE

Piensa en una guía de montaña que conduce a un grupo de escaladores experimentados. No puede ordenar a nadie que suba la montaña. La gente la sigue porque conoce la ruta, explica los riesgos con honestidad, escucha cuando alguien detecta un peligro, admite cuando no está segura y ya ha acertado antes. Cuando el grupo no se pone de acuerdo sobre la ruta, se asegura de que todos sean escuchados, acuerda cómo se tomará la decisión y después ayuda a todos a comprometerse con ella. La mayoría de los arquitectos están en la misma posición: son responsables de la ruta, pero no tienen poder para dar órdenes, así que lideran mediante la confianza, un razonamiento claro y buenas decisiones tomadas en común.

LO QUE NECESITAS SABER

- Los arquitectos suelen tener responsabilidad sin autoridad formal, por lo que su influencia se basa en la experiencia, la información y las relaciones más que en el poder posicional (French y Raven, 1959).
- La confianza crece con la credibilidad, la fiabilidad y la intimidad, y disminuye con la autoorientación (Maister, Green y Galford, 2000); la seguridad psicológica permite a los equipos plantear problemas mientras aún son baratos de resolver.
- Acuerda los derechos de decisión y el método de decisión antes de debatir el fondo; el consentimiento adopta una propuesta salvo que quede una objeción razonada, mientras que el consenso exige el apoyo de todos.
- La negociación basada en principios separa a las personas de los problemas, se centra en los intereses y no en las posiciones, y acuerda criterios objetivos; elige un modo de conflicto de forma deliberada y escala los desacuerdos de forma conjunta.

- El ascensor del arquitecto de Hohpe conecta el ático (la estrategia) con la sala de máquinas (la realidad técnica): sube riesgos y opciones, y baja la estrategia y sus motivos.
- Hacer crecer a otros mediante la mentoría, el coaching, el patrocinio y la delegación, y apoyar las trayectorias profesionales staff-plus (Larson, 2021; Reilly, 2022), evita que el arquitecto se convierta en un cuello de botella.
- Los grupos añaden sus propias trampas de decisión, como la deferencia ante la jerarquía, la escalada del compromiso, el bikeshedding y la decisión por desgaste, y cada una tiene una contramedida práctica.

Liderar sin autoridad

La mayoría de los arquitectos no dirigen a las personas que construyen lo que ellos diseñan. Los equipos dependen de los responsables de ingeniería, las prioridades corresponden a los responsables de producto y los presupuestos, a los directores. Aun así, se espera que el arquitecto se asegure de que las decisiones significativas se tomen bien y se respeten en muchos equipos, que es la responsabilidad central descrita en la Sección 1.4. La distancia entre la responsabilidad y la autoridad formal es la condición que define el puesto, y salvarla es una cuestión de influencia.

El estudio clásico de John French y Bertram Raven sobre el poder social (1959) distinguió las bases de poder que vienen con un puesto (la autoridad legítima, la capacidad de recompensar y la capacidad de castigar) de las que pertenecen a una persona (la experiencia y el poder referente, es decir, el respeto y la identificación que sienten los demás); más tarde Raven añadió el poder informativo, la capacidad de aportar hechos que hacen cambiar de opinión. Los arquitectos rara vez tienen mucho poder posicional y, cuando lo tienen, usarlo sale caro: una decisión impuesta por el rango se acata, pero nadie la hace suya, y las personas que no estaban de acuerdo dejan de contarte los problemas. La influencia arquitectónica duradera se basa casi por completo en la experiencia, la información y las relaciones.

No se trata de una habilidad blanda añadida al trabajo técnico. Un diseño sólido que nadie adopta tiene el mismo efecto que no tener ningún diseño, y un diseño mediocre que los equipos entienden y apoyan a menudo da mejores resultados que uno brillante que esquivan. El resto de esta sección trata la influencia como un conjunto de prácticas que se pueden aprender: generar confianza, facilitar decisiones, gestionar el conflicto, comunicarse hacia arriba, hacer crecer a otras personas y evitar las trampas en las que caen los grupos.

Generar confianza y credibilidad

The Trusted Advisor (2000), de David Maister, Charles Green y Robert Galford, ofrece un modelo útil. La confianza crece con la credibilidad (sabes de lo que hablas), la fiabilidad (haces lo que dices) y la intimidad (las personas se sienten seguras para plantearte cuestiones difíciles), y disminuye con la autoorientación (la sensación de que sirves a tus propios intereses, como tener razón, parecer inteligente o proteger tu diseño). La aportación del modelo para los arquitectos es que la autoorientación está debajo de la línea: divide todo lo demás. No se confía en un arquitecto brillante y fiable que necesita a ojos de todos ganar cada discusión.

En la práctica, la credibilidad se construye de formas pequeñas y visibles. Acierta en las cosas pequeñas antes de pedir que confíen en ti en las grandes. Mantente lo bastante cerca del código para entender su costo real: escribe el spike, revisa el pull request, asiste a una revisión de incidentes. Di lo que no sabes y cómo lo averiguarías. Cumple cada compromiso, sobre todo los aburridos. Reconoce el mérito de los demás en público y asume tus propios errores, también en público. Explica el razonamiento que hay detrás de cada principio para que las personas puedan aplicarlo en situaciones que no previste.

La confianza también depende del clima que te rodea. La investigación de Amy Edmondson sobre la seguridad psicológica, que comenzó con un estudio de 1999 sobre equipos de trabajo, descubrió que los equipos cuyos miembros creen que es seguro admitir errores y cuestionar a los demás aprenden más y rinden mejor. Los arquitectos influyen en ese clima más de lo que creen. Reaccionar a las malas noticias con curiosidad en lugar de buscar culpables, pedir objeciones de forma explícita (¿qué haría que esto fracasara?) y agradecérselas a quien las plantea es lo que permite enterarse de un diseño defectuoso mientras todavía es barato de corregir.

Facilitar decisiones

Muchas discusiones de arquitectura son en realidad discusiones sobre el proceso: quién decide, a quién hay que consultar y cómo se resolverán los desacuerdos. Acordar los derechos de decisión antes de discutir el fondo quita buena parte de la tensión. La Sección 11.4 describió el proceso de consejo (advice process), en el que cualquiera puede tomar una decisión siempre que pida consejo a los afectados y a quienes tienen experiencia; otras organizaciones usan un foro de revisión de arquitectura, un responsable de decisión designado para cada dominio o matrices de roles como RACI. El mecanismo concreto importa menos que el hecho de que se conozca de antemano.

Dentro de los derechos de decisión que se apliquen, el arquitecto suele ser el facilitador: la persona que plantea la pregunta, se asegura de que estén presentes las personas y las pruebas adecuadas y guía al grupo hacia una decisión que perdure. Distintos métodos de

decisión convienen a distintas situaciones, y elegir el método de forma deliberada forma parte del trabajo.

Hay dos métodos que a menudo se confunden. El consenso busca una propuesta que todos apoyen activamente; genera un compromiso fuerte, pero es lento y da a cualquier persona un veto efectivo. La toma de decisiones por consentimiento, desarrollada en la sociocracia de Gerard Endenburg en los Países Bajos a partir de la década de 1970, adopta una propuesta salvo que alguien plantee una objeción razonada de que causaría un daño o haría retroceder al grupo. La pregunta pasa de ¿están todos de acuerdo? a ¿hay alguien que no pueda vivir con esto, y por qué? Una frase de la práctica sociocrática resume bien el umbral: suficientemente bueno por ahora, suficientemente seguro para probarlo. El consentimiento funciona especialmente bien para las decisiones de puerta de doble sentido, que se pueden revisar a medida que llegan pruebas. El consenso aproximado (rough consensus) del IETF, descrito en la Sección 4.8, es un pariente cercano.

Cuando no se puede llegar a un acuerdo, alguien tiene que decidir, y ya debería estar claro quién. Disagree and commit (discrepar y comprometerse), que también se trata en la Sección 4.8, pide a las personas que defiendan su postura a fondo antes de la decisión y que la apoyen por completo después. Solo funciona si el desacuerdo se escuchó de verdad, la discrepancia queda registrada (en el registro de decisiones de arquitectura, con sus motivos) y existe una señal acordada que reabriría la cuestión.

Elegir un método de decisión

Método	Cómo funciona	Úsalo cuando	Principal riesgo
Decidir en solitario	Decide la persona responsable, quizá tras una consulta rápida	Puertas de doble sentido, emergencias, decisiones de alcance limitado	Conocimiento que se pasa por alto; poco compromiso de los demás
Consultar y luego decidir (proceso de consejo)	Quien decide debe pedir consejo a los afectados y a los expertos; después decide y registra por qué	La mayoría de las decisiones de arquitectura en equipos autónomos	Se pide consejo por cumplir y luego se ignora
Consentimiento	Adoptar salvo que quede una objeción razonada y de peso; las objeciones mejoran la propuesta	Decisiones que afectan a un grupo y se pueden revisar	Objeciones usadas indebidamente como vetos por preferencia
Consenso	Continuar hasta que todos apoyen la propuesta	Grupos pequeños, valores o principios de mucho peso que necesitan un compromiso profundo	Lentitud; bloqueo; decisiones tomadas por omisión
Votación	Votación por mayoría o ponderada entre los participantes	Elegir entre opciones que son todas aceptables	Una minoría perdedora sin compromiso; popularidad por encima de las pruebas
Delegar	Confiar la decisión, con restricciones, a un equipo o a una persona más cercana al trabajo	La decisión es local y las restricciones se pueden enunciar	Delegar sin el contexto o sin las restricciones

SI ESTO ES NUEVO PARA TI

Paso a paso: el consentimiento no es consenso

El punto en el que más gente tropieza aquí es la diferencia entre el consenso, en el que todos deben apoyar activamente una propuesta, y el consentimiento, en el que se adopta salvo que alguien plantee una objeción razonada. Sigue una pequeña decisión a través de ambos.

- Tras la sesión de risk-storming de la Sección 11.2, las seis personas que participaron deben acordar cuándo volver a examinar el diseño. El arquitecto propone hacerlo cada tres meses.
- Por consenso: un desarrollador quiere revisiones mensuales; el ingeniero de operaciones, solo antes de los cambios importantes. La discusión continúa hasta que las seis personas apoyen activamente una respuesta, así que cualquiera de ellas puede bloquearla. Puede alargarse hasta estancarse o hasta que gane la voz más insistente.
- Por consentimiento: el facilitador pregunta: "¿Hay alguien que no pueda vivir con cada tres meses, y por qué?". El desarrollador sigue prefiriendo una revisión mensual, pero puede vivir con una trimestral. Una preferencia no es una objeción, así que no bloquea la propuesta.
- El ingeniero de seguridad objeta, con un motivo: el mes que viene está previsto un cambio importante en la ruta de pago, y un intervalo de tres meses lo dejaría sin revisar justo cuando es más nuevo y más arriesgado. Es una afirmación razonada de daño, así que la propuesta se mejora: cada tres meses, más una revisión después del cambio en los pagos.
- Nadie objeta la propuesta mejorada, así que se adopta. No tiene por qué ser perfecta: suficientemente buena por ahora, suficientemente segura para probarla y fácil de revisar más adelante.
- **La lección:** el consentimiento pregunta "¿Hay alguien que no pueda vivir con esto, y por qué?" en lugar de "¿Estamos todos de acuerdo?". Conviene a las decisiones que se pueden revisar. Reserva el consenso para las pocas que necesitan el compromiso profundo de todos, y acuerda el método antes de que empiece el debate.

El conflicto y la política organizativa

La política organizativa es simplemente la forma en que grupos con intereses distintos, recursos limitados y sus propias carreras toman decisiones. No es una corrupción del trabajo de arquitectura, sino su entorno. Ignorarla no hace que un arquitecto tenga más principios, solo que sea menos eficaz. Las herramientas de capítulos anteriores se aplican directamente: el

mapa de partes interesadas y la matriz poder/interés de la Sección 1.5 muestran quién se ve afectado y quién puede bloquear, y los registros de compromisos de la Sección 1.6 hacen visibles las posiciones.

Getting to Yes (1981), de Roger Fisher y William Ury, sigue siendo la guía más útil sobre la negociación basada en principios, y sus cuatro ideas se trasladan directamente. Separa a las personas del problema: la responsable de pagos no es obstruccionista, le preocupan los datos de las tarjetas. Céntrate en los intereses, no en las posiciones: no debemos usar Kafka es una posición; no podemos asumir más carga de guardias es un interés que varios diseños podrían satisfacer. Inventa opciones de beneficio mutuo antes de decidir. Insiste en criterios objetivos, acordados antes de comparar las opciones. Su idea de la mejor alternativa a un acuerdo negociado (BATNA) también es útil: saber qué ocurre si no se llega a un acuerdo indica a todos cuánto vale el acuerdo.

El modelo de modos de conflicto de Kenneth Thomas y Ralph Kilmann (1974) describe cinco formas de responder al conflicto, según el peso que des a tus propias inquietudes y a las de la otra parte. Ninguna es siempre la correcta; la habilidad consiste en elegir de forma deliberada en lugar de recurrir a la costumbre.

Algunos hábitos prácticos ayudan en todos los casos. No sorprendas nunca a una parte interesada en una reunión pública con una propuesta que le afecta; habla antes con ella, en privado (las organizaciones japonesas llaman nemawashi a esta consulta preparatoria). Cuando haga falta escalar, hay que hacerlo juntos: ambas partes presentan el desacuerdo y las opciones a la persona que puede decidir, en lugar de presionar cada una por separado. Y mantén los desacuerdos sobre los diseños separados de los juicios sobre las personas, tanto en lo que dices como en lo que escribes.

Cinco modos de conflicto (según Thomas y Kilmann)

Modo	Qué significa	Útil cuando	Si se abusa de él, lleva a
Competir	Defender con firmeza tu propia posición	Cuestiones de seguridad (safety), de seguridad de la información o legales en las que debes mantenerte firme	Acatamiento sin compromiso; la gente deja de plantear problemas
Colaborar	Trabajar juntos para satisfacer plenamente los intereses de ambas partes	Decisiones importantes en las que las inquietudes de ambas partes son legítimas	Decisiones lentas sobre asuntos que no lo requerían
Transigir	Cada parte renuncia a una parte de lo que quiere	Hay poco tiempo y ambas partes tienen el mismo poder	Diseños que no satisfacen a nadie y son incoherentes
Evitar	Aplazar o esquivar la cuestión	La cuestión es trivial o los ánimos necesitan tiempo para calmarse	Problemas que crecen hasta convertirse en crisis
Complacer	Ceder ante la posición de la otra parte	Estabas equivocado, o la cuestión les importa mucho más a ellos	Pérdida de credibilidad y principios abandonados

Comunicarse con los directivos

La Sección 4.8 trató cómo adaptar un mensaje a su público y empezar por la respuesta. Los directivos merecen algo más de atención, porque tienen los presupuestos y la autoridad de los que depende el trabajo de arquitectura, y porque los arquitectos suelen perderlos por explicar demasiado y pedir demasiado poco.

Plantea cada tema en los términos que los directivos ya gestionan: ingresos, costos, riesgo, plazo de comercialización, experiencia del cliente, regulación y personas. Lleva opciones en lugar de un único plan, cada una con su costo, su riesgo y aquello a lo que renuncia, y di cuál recomiendas y por qué. Cuantifica donde puedas hacerlo con honestidad, y di qué grado de confianza tienen las cifras; un rango con una base explícita es más creíble que una falsa precisión. Deja claro qué decisión o qué apoyo necesitas de ellos y para cuándo. Prepárate para la pregunta ¿qué harías si fuera tu dinero? Y no dejes nunca que un directivo se entere de un riesgo grave primero por otra persona.

A las personas de alto nivel a menudo les interesa menos cómo funciona un sistema que si sus responsables lo tienen bajo control. Mostrar que los riesgos se conocen, tienen responsable, se miden y se están reduciendo (por ejemplo, mediante un registro de riesgos de arquitectura o las funciones de aptitud de la Sección 11.3) genera más confianza que cualquier diagrama. La Sección 11.8 muestra cómo presentar el trabajo de arquitectura en la forma que usan los directivos para comparar inversiones: el caso de negocio.

El ascensor del arquitecto

The Software Architect Elevator (O'Reilly, 2020), de Gregor Hohpe, ofrece la imagen más memorable de este trabajo. Las grandes organizaciones son como edificios altos. En el ático, los directivos fijan la estrategia y los presupuestos; en la sala de máquinas, los ingenieros construyen y operan los sistemas; entre ambos hay muchas plantas de gestión, y la información que pasa por ellas se filtra y se distorsiona por el camino. El ascensor del arquitecto (architect elevator) de Hohpe se desplaza entre el ático y la sala de máquinas y lleva información en ambas direcciones: baja la estrategia y sus motivos a las personas que construyen los sistemas, y sube las realidades técnicas (restricciones, riesgos, costos y opciones que el negocio no sabía que tenía) a las personas que toman las decisiones estratégicas.

La idea de Hohpe es que el valor del arquitecto está en conectar las plantas. Un arquitecto que se queda en el ático se convierte en un estratega de torre de marfil cuyos planes ignoran lo que es técnicamente posible; uno que se queda en la sala de máquinas se convierte en un experto respetado cuyas advertencias nunca llegan a las personas que podrían actuar en consecuencia. Usar bien el ascensor significa cambiar de lenguaje y de nivel de detalle en cada planta sin cambiar el fondo, que es exactamente la habilidad de adaptarse al público de la Sección 4.8 aplicada a toda una organización.

Viajar en el ascensor



El valor del arquitecto está en conectar las plantas

Según Gregor Hohpe: el arquitecto conecta la estrategia del ático con la realidad técnica de la sala de máquinas, traduciendo en ambas direcciones.

Hacer crecer a otros arquitectos

Un arquitecto que toma personalmente todas las decisiones significativas se convierte en un cuello de botella, y la capacidad arquitectónica de la organización se va con él. Por eso hacer crecer a otros forma parte del rol, no es un extra opcional. Ayuda distinguir tres actividades. La mentoría comparte tu experiencia y tus consejos. El coaching hace preguntas que ayudan a alguien a llegar a su propia respuesta, lo que desarrolla el criterio más rápido. El patrocinio usa tu credibilidad para dar oportunidades a alguien: recomendarle para dirigir un diseño, poner su nombre en la propuesta, invitarle a la reunión con los directivos.

Entre las técnicas prácticas están trabajar en pareja en diseños y revisiones, dejar que ingenieros con menos experiencia redacten el ADR de una decisión y revisarlo con ellos, hacer katas de arquitectura (Sección 11.2) como práctica, delegar decisiones dentro de unos guardrails (salvaguardas) establecidos y explicar en voz alta tu razonamiento durante las discusiones de diseño, para que los demás vean cómo se sopesan los compromisos y no solo la conclusión.

Muchas organizaciones ofrecen hoy una trayectoria profesional técnica paralela a la de gestión. *Staff Engineer: Leadership beyond the management track* (2021), de Will Larson, popularizó el término staff-plus para los contribuidores individuales staff, principal y de nivel superior, y describió cuatro arquetipos habituales a partir de entrevistas con personas en esos puestos. *The Staff Engineer's Path* (O'Reilly, 2022), de Tanya Reilly, organiza el trabajo en torno a la visión de conjunto, la ejecución de grandes proyectos y el crecimiento de los ingenieros

que te rodean. En su charla de 2019 "Being Glue", Reilly también llamó la atención sobre el trabajo de coordinación (desbloquear a las personas, escribir el documento que falta, detectar el hueco entre equipos) que mantiene vivos los proyectos pero que a menudo no se recompensa en los ascensos; los arquitectos deberían fijarse en quién lo hace y asegurarse de que cuente.

Arquetipos staff-plus (según Will Larson)

Arquetipo	Enfoque	Contribución arquitectónica típica
Tech lead (líder técnico)	Guía el enfoque y la ejecución de un equipo, en colaboración con su responsable	Asume las decisiones de arquitectura del equipo y las mantiene alineadas con el sistema en su conjunto
Arquitecto	Responsable de la dirección, la calidad y el enfoque de un área crítica	Establece principios, revisa diseños y vela por las cualidades clave en varios equipos
Solucionador (solver)	Se adentra en problemas difíciles y ambiguos y encuentra una salida, a menudo pasando de uno a otro	Resuelve la migración atascada, la crisis de rendimiento o el límite poco claro
Mano derecha	Amplía el alcance de la atención de un directivo, tomando prestados su ámbito y su autoridad	Impulsa iniciativas técnicas para toda la organización y conecta la estrategia con la ejecución

Las trampas de decisión, de nuevo

La Sección 1.6 enumeró los sesgos individuales que distorsionan las decisiones arquitectónicas: el anclaje, el costo hundido, el pensamiento de grupo, el desarrollo dirigido por el currículum, el bombo publicitario (hype) y el sesgo de confirmación. Dirigir decisiones en grupo añade trampas propias, que surgen del estatus, el cansancio y la inercia organizativa más que del razonamiento individual.

Trampas de grupo y organizativas

Trampa	Cómo se manifiesta	Contramedida
Deferencia ante la opinión de quien más cobra	La persona de mayor rango habla primero y la discusión converge hacia su punto de vista	Pide a la persona de mayor rango que hable la última; recoge opiniones por escrito antes de la reunión
Escalada del compromiso	Una iniciativa que fracasa recibe más inversión porque ya se ha comprometido mucho en ella (Barry Staw, 1976)	Fija desde el principio criterios de cancelación y fechas de revisión; revisa con personas que no participaron
Bikeshedding (discutir lo trivial)	Largos debates sobre detalles triviales y fáciles de entender mientras la decisión difícil recibe unos minutos (ley de la trivialidad de Parkinson, 1957)	Limita el tiempo según la importancia; deja los detalles de puerta de doble sentido al equipo responsable de ellos
Decisión por desgaste	Gana la opción respaldada por quien más aguanta, una vez que todos los demás están cansados	Acuerda de antemano el método de decisión, quién decide y el plazo
Falso consenso	El silencio se toma como acuerdo; las objeciones aparecen después de la decisión	Pide objeciones de forma explícita, por nombre si es necesario, y regístralas
No inventado aquí (not invented here)	Las soluciones externas se rechazan porque no se construyeron internamente	Compara construir y comprar con los mismos criterios, incluido el costo de operar lo que construyes
Parálisis por análisis	Siempre se pide más información; no se decide nada antes del último momento responsable	Fija la fecha de la decisión y qué pruebas serían suficientes; toma rápido las decisiones reversibles

EJEMPLO PRÁCTICO

Ejemplo práctico: cómo conducir una decisión disputada sobre un bus de eventos

Este ejemplo se remonta a la decisión que introdujo el intermediario de eventos (broker) de la plataforma de reparto de comida, antes de que existiera la arquitectura evaluada en la Sección 11.1. El equipo de plataforma propone que todos los servicios se comuniquen a través de un clúster de Kafka gestionado para toda la empresa. El equipo de pedidos está entusiasmado, el equipo de pagos se opone con fuerza, finanzas está preocupada por el costo y el CTO quiere una decisión en un mes. Según el proceso de consejo de la empresa, quien toma la decisión es el arquitecto jefe del equipo de plataforma. Se pide a un arquitecto principal que facilite el proceso.

1. Replantear la pregunta	¿Kafka para todo? es una posición. El problema de fondo, tras dos caídas en viernes, es que hacer un pedido depende de llamadas síncronas a notificaciones y al despacho. La pregunta pasa a ser: ¿cómo deberían publicar los servicios los eventos de negocio, empezando por los eventos de pedido, para que los consumidores dejen de estar en la ruta del pedido? El alcance y los derechos de decisión se escriben al principio de un borrador de ADR.
2. Escuchar primero en privado	Antes de cualquier reunión, el facilitador habla con cada parte. Pagos: los datos de las tarjetas deben quedar fuera de los sistemas nuevos, y el equipo no puede asumir más trabajo de guardias tras un incidente doloroso con un broker autoalojado. Pedidos: necesita desacoplamiento para los picos. Finanzas: quiere limitar los costos de operación. Seguridad: necesita cifrado y control de acceso por topic. El equipo de plataforma: quiere un único estándar con soporte en lugar de cinco. Cada uno se registra como un interés, no como una posición.
3. Acordar los criterios antes que las opciones	En el foro asesor de arquitectura, el grupo acuerda criterios ponderados: garantías de entrega y orden por pedido (5), carga operativa para los equipos de producto (5), alcance de cumplimiento normativo (4), costo de operación con el triple del volumen actual (3), conocimientos (3) y reversibilidad (3).
4. Generar opciones, incluidas las de quienes discrepan	A: Kafka gestionado, obligatorio para todos los servicios. B: Kafka gestionado para los eventos del dominio de pedidos mediante una bandeja de salida transaccional (transactional outbox); los demás equipos se suman más adelante si quieren. C: los servicios más sencillos de publicación-suscripción y de colas del proveedor de nube. D: mantener las llamadas síncronas y añadir reintentos y circuit breakers. Pagos aporta la opción D y el análisis de costos de C.

5. Reunir pruebas	Un spike de dos semanas ejecuta B y C con una carga similar a la de producción: latencia de extremo a extremo, comportamiento durante la conmutación por error del broker y costo mensual estimado. Seguridad confirma que los eventos de pedido pueden diseñarse sin datos de tarjetas, lo que los mantiene fuera del ámbito de las tarjetas de pago.
6. Ronda de consentimiento	Propuesta: B, con el equipo de plataforma operando el clúster como X como servicio (X-as-a-Service). Pagos objeta que aun así podría verse arrastrado a la operación del broker; la objeción se resuelve escribiendo la responsabilidad y el horario de soporte del equipo de plataforma en su API del equipo. La inquietud de finanzas se convierte en una alerta de costos y un umbral de revisión. Una ingeniera de pedidos sigue prefiriendo C; puede vivir con B, y su razonamiento queda registrado.
7. Decidir, registrar y comunicar	Quien toma la decisión acepta B. El ADR registra las opciones, las puntuaciones, los resultados del spike, la discrepancia y los desencadenantes de revisión (que más de tres equipos lo adopten, que los costos superen el umbral acordado o que pasen seis meses, lo que ocurra primero). El CTO y el CFO reciben un resumen de una página sobre el beneficio, el costo y el riesgo; los equipos reciben una sesión explicativa; la entrada del radar tecnológico se actualiza a Trial.
8. Hacer el seguimiento	Una comprobación de compatibilidad de esquemas se ejecuta en el pipeline de cada productor, se vuelve a probar el escenario de la caída y, en la revisión de los seis meses, las pruebas se usan para decidir si ampliar, mantener o revertir la decisión.

El facilitador nunca defendió una opción; el trabajo se centró en el planteamiento, los intereses, los criterios, las pruebas y un método de decisión claro.

La discrepancia cambió el resultado (se redujo el alcance y se hizo explícita la responsabilidad de la plataforma) y quedó registrada en lugar de descartarse en silencio.

La decisión se hizo reversible en la práctica gracias a un alcance limitado y a unos desencadenantes de revisión acordados.

ATENCIÓN

Antipatrones en el liderazgo arquitectónico

- Ganar las discusiones por jerarquía, lo que corta el flujo de malas noticias.
- Decidir en privado y anunciarlo en público, de modo que la consulta se percibe como teatro.
- Tratar cada desacuerdo como una resistencia que hay que gestionar en lugar de como información que se puede aprovechar.
- Prometer a los directivos una certeza que el trabajo de ingeniería no puede respaldar.
- Acaparar las decisiones interesantes, de modo que nadie más aprende a tomarlas.
- Mantenerse neutral en cuestiones de seguridad (safety), de seguridad de la información o éticas en las que el arquitecto debería mantenerse firme.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué hablar en privado con las partes interesadas antes de una reunión de decisión?

Para entender sus intereses en privado, evitar sorprenderlas en público y sacar a la luz las objeciones pronto, cuando todavía pueden dar forma a la propuesta.

▼ ¿En qué se diferencian la mentoría, el coaching y el patrocinio?

La mentoría comparte tu experiencia y tus consejos, el coaching hace preguntas que ayudan a alguien a llegar a su propia respuesta y el patrocinio usa tu credibilidad para crearle oportunidades.

▼ ¿Qué pierde un arquitecto si nunca deja la sala de máquinas?

La capacidad de influir en la estrategia: su conocimiento de los riesgos, las restricciones y las opciones nunca llega a las personas que toman las decisiones de inversión y organizativas.

▼ ¿Por qué debería registrarse la discrepancia en el registro de decisiones?

Para que se conserve el razonamiento, quienes discreparon sepan que se les escuchó y, si sus inquietudes se hacen realidad, la organización sepa qué revisar y por qué.

▼ **Inténtalo: El equipo de plataforma quiere un único clúster compartido de Kafka; Pagos se opone firmemente; no tienes autoridad sobre ninguno de los dos equipos, y el CTO quiere una decisión en un mes. ¿Cómo gestionarías esto sin forzar un resultado?**

Primero acuerda quién decide y cómo, por ejemplo por consentimiento con el CTO como árbitro de desempate. Después separa posiciones de intereses: Pagos puede temer por el alcance de la auditoría o por la pérdida de control, y el equipo de plataforma quiere tener menos que operar. Acuerda criterios objetivos, como costo, latencia y cumplimiento, y ponlos a prueba con un spike pequeño. Las opciones podrían incluir un clúster separado para Pagos. Anota el desacuerdo en el ADR. Si hay un bloqueo, la escalada debe ser conjunta, con una declaración compartida de opciones, no como rivales.

FUENTES

- Gregor Hohpe: The Architect Elevator: https://architectelevator.com/
- Will Larson: Staff archetypes: https://staffeng.com/guides/staff-archetypes/
- Andrew Harmel-Law: Scaling the Practice of Architecture, Conversationally: https://martinfowler.com/articles/scaling-architecture-conversationally.html
- RFC 7282: On Consensus and Humming in the IETF: https://www.rfc-editor.org/rfc/rfc7282

SECCIÓN 11.8 DE 11.8 · AVANZADO · 31 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Ética, sostenibilidad y el caso de negocio

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.7 Deuda técnica
- 2.7 Seguridad, usabilidad y más
- 8.7 Privacidad desde el diseño

Términos anteriores que utiliza: Deuda técnica, Accesibilidad, Modelado de amenazas, Sistema heredado (legacy)

LA IDEA EN BREVE

Antes de pagar un tejado nuevo, un propietario sensato pregunta cuánto cuesta, cuánto ahorrará en reparaciones y calefacción, en cuánto tiempo se amortiza y qué pasa si espera otro invierno. Un buen constructor responde a esas preguntas con honestidad. Un buen constructor también se niega a instalar un cableado que podría provocar un incendio, aunque el cliente lo quiera más barato, y propone un aislamiento que reduce las facturas de energía. Los arquitectos necesitan ambos hábitos: presentar un argumento financiero honesto para el trabajo que proponen y asumir la responsabilidad por el daño o el despilfarro que sus diseños podrían causar.

LO QUE NECESITAS SABER

- Un caso de negocio compara opciones, incluida la de no hacer nada, según sus costos, beneficios y riesgos en dinero, tiempo y probabilidad, y recomienda una; la financiación por etapas convierte una gran apuesta en opciones más pequeñas.
- El valor actual neto (NPV) descuenta el flujo de caja de cada año: $NPV = \Sigma\ C_t \div (1 + r)^t$; un NPV positivo significa que la inversión supera la tasa de descuento, que normalmente fija el área de finanzas.
- El periodo de recuperación (payback) es el tiempo hasta que el flujo de caja acumulado pasa a ser positivo; preséntalo junto al NPV y a un análisis de sensibilidad de las estimaciones que más importan.
- La deuda técnica se convierte en un caso de negocio cuando el interés se expresa como costo anual (tiempo, retraso, incidentes, riesgo) y el principal como la inversión; las cifras de corrección de las herramientas miden el principal, no el interés.

- El Código de Ética de la ACM (2018) y el Código de Conducta de la BCS exigen a los arquitectos tener en cuenta el interés público, evitar daños, evaluar los riesgos a fondo y ser honestos sobre ellos.
- Entre los daños previsibles están la privacidad, la seguridad, el sesgo, la falta de accesibilidad, la seguridad física (safety) y los patrones oscuros manipuladores, varios de los cuales necesitan soporte arquitectónico y ya están regulados.
- Las prácticas de software verde abarcan la eficiencia en carbono y en energía, la conciencia del carbono, la eficiencia del hardware, la medición y las reducciones reales; la especificación SCI (ISO/IEC 21031:2024) mide las emisiones por unidad funcional.

Por qué los arquitectos necesitan casos de negocio

El trabajo de arquitectura compite por el dinero y las personas con todo lo demás que una organización quiere hacer. Una migración, una plataforma, un programa de resiliencia o el pago de una deuda rara vez producen una funcionalidad que un cliente pueda ver, así que es fácil aplazarlos en favor del trabajo que sí la produce. Los arquitectos que consiguen financiación para ese trabajo no suelen ser los que tienen los diseños más elegantes, sino los que saben expresar el argumento en los términos que usan los responsables de decisión para comparar inversiones: cuánto cuesta, qué rinde, en cuánto tiempo, con qué certeza y qué ocurre si no lo hacemos. Ese argumento estructurado es un caso de negocio (business case).

Esta es la cara orientada al negocio de todo lo que este curso ha dicho sobre los compromisos. Un caso de negocio es un registro de decisión para una inversión: contexto, opciones, criterios, consecuencias y una recomendación, con las consecuencias expresadas en la medida de lo posible en dinero, tiempo y riesgo. No sustituye al criterio arquitectónico, y gran parte de lo que protege la arquitectura (la seguridad, el aprendizaje, la opcionalidad) es difícil de valorar en dinero. Pero poner cifras honestas donde se pueden poner, y decir con claridad dónde no se puede, da derecho a ser escuchado sobre el resto.

Las organizaciones usan formatos distintos. El Green Book del Gobierno del Reino Unido, publicado por HM Treasury, estructura los casos del sector público en torno a cinco preguntas: el caso estratégico (por qué hace falta el cambio), el caso económico (qué opción ofrece el mejor valor), el caso comercial (si se puede contratar), el caso financiero (si es asequible) y el caso de gestión (si se puede llevar a cabo). La mayoría de las empresas usan algo más ligero, pero vale la pena responder a las mismas preguntas para cualquier inversión de arquitectura significativa.

Opciones, costos, beneficios y riesgos

Todo caso de negocio debería comparar al menos tres opciones: no hacer nada (la línea de base), hacer lo mínimo y la opción recomendada, además de cualquier alternativa seria. No hacer nada rara vez sale gratis. El contrato de soporte del sistema heredado sube, la tasa de incidentes crece con el tráfico y los cambios se vuelven cada vez más lentos; esa deriva es la línea de base con la que se miden los beneficios de las demás opciones, y omitirla hace que toda inversión parezca peor de lo que es.

Los costos y los beneficios deberían enumerarse por categorías para que no se olvide ninguno ni se cuente ninguno dos veces. Los riesgos deben figurar explícitamente en el caso: tanto los riesgos que la inversión reduce, que pueden valorarse como pérdidas esperadas evitadas (probabilidad multiplicada por impacto), como los riesgos de la propia inversión, como un sobrecosto o las interrupciones durante la migración. El costo del retraso de Don Reinertsen, de *The Principles of Product Development Flow* (2009), es una perspectiva adicional útil: el valor que se pierde cada semana que un trabajo no se entrega. Dividir el costo del retraso por la duración (weighted shortest job first, primero el trabajo más corto ponderado) ofrece una forma defendible de ordenar trabajos que compiten entre sí.

Por último, estructura la inversión como una serie de etapas con puntos de decisión en lugar de un único gran compromiso. La migración incremental de la Sección 11.6 es un buen ejemplo: cada paso entrega algo y genera pruebas, así que la financiación puede liberarse paso a paso y el programa puede detenerse si los beneficios no aparecen. Las compuertas por etapas de este tipo convierten una apuesta arriesgada en un conjunto de opciones más pequeñas, que es el razonamiento sobre la reversibilidad de la Sección 1.6 aplicado al dinero.

Qué incluye un caso de negocio de arquitectura

Categoría	Partidas típicas	Cómo estimar
Costos puntuales	Tiempo de ingeniería, arquitectura de transición, migración de datos, licencias, formación, ejecución en paralelo, desmantelamiento	Tamaño del equipo multiplicado por la duración y por la tarifa diaria con costos totales; presupuestos de proveedores
Costos recurrentes	Infraestructura, licencias y suscripciones, soporte, guardias, nuevos conocimientos que hay que retener	Facturas actuales y modelos de capacidad; precios del proveedor con el volumen previsto
Beneficios de eficiencia	Menos esfuerzo por cambio, menor tiempo de entrega, menos trabajo manual y menos traspasos	Volumen de cambios multiplicado por el tiempo ahorrado por cambio, a partir de los datos de entrega
Reducción de riesgos	Incidentes menos frecuentes o más cortos, brechas de seguridad evitadas, sanciones regulatorias evitadas	Pérdida esperada: frecuencia multiplicada por impacto, antes y después
Ingresos y opciones	Funcionalidades que se hacen posibles, mercados que se abren, experimentos más rápidos	Costo del retraso de las oportunidades habilitadas; a menudo se muestra aparte como potencial adicional
Costos evitados	Actualizaciones por fin de soporte, renovación de hardware, subidas de tarifas de licencia que dejan de aplicarse	Contratos y hojas de ruta de los proveedores

SI ESTO ES NUEVO PARA TI

Paso a paso: por qué el dinero futuro cuenta menos

La idea más difícil de esta sección es el descuento: por qué el dinero que llega dentro de unos años cuenta menos que el dinero de hoy, y cómo el valor actual neto lo usa para sopesar un costo que se paga ahora frente a beneficios que llegan más tarde. Unas cifras sencillas preparan el ejemplo práctico que sigue.

- La migración del módulo de pedidos de la Sección 11.6 cuesta unas £420 000 ahora; sus beneficios llegan a lo largo de los cuatro años siguientes. Sumar sin más las libras supondría tratar una libra del año 4 como si valiera una libra hoy, y no es así.
- El dinero que se tiene hoy puede ponerse a trabajar, y un beneficio previsto puede no llegar nunca. Por eso finanzas fija una tasa de descuento, aquí del 10%, que refleja lo que le cuesta el dinero a la organización y el riesgo de la inversión.
- Al 10%, £100 000 de hoy se convierten en £110 000 en un año y en £121 000 en dos. A la inversa, £121 000 que llegan dentro de dos años valen hoy £100 000.
- Descontar es hacer esa cuenta al revés: dividir un flujo de caja entre 1.1 una vez por cada año de espera. Cuatro divisiones convierten un beneficio de £190 000 en el año 4 en solo unas £129 800 de hoy.
- El valor actual neto suma el flujo de caja descontado de cada año, incluida la inversión, que se paga ahora y por tanto no se descuenta. Un total positivo significa que la inversión rinde más que la tasa de descuento. Como el costo llega primero y los beneficios después, el descuento hace que un caso así parezca menor que con un simple total.
- **La lección:** no sumes nunca libras de años distintos como si fueran iguales. Convierte el flujo de caja de cada año a su valor de hoy con la tasa que fija finanzas y después súmalos. La respuesta solo es tan buena como las estimaciones, y por eso el ejemplo práctico también prueba beneficios menores.

Valor actual neto y periodo de recuperación

El dinero de ahora vale más que el dinero futuro: puede invertirse, y el dinero futuro es menos seguro. Para comparar flujos de caja que llegan en años distintos, los equipos de finanzas convierten cada uno a su valor de hoy, usando una tasa de descuento que refleja el costo de capital de la organización y el riesgo de la inversión. Normalmente la tasa la fija el departamento de finanzas, y los arquitectos deberían usarla en lugar de elegir la suya propia.

El valor actual neto (NPV) de una inversión es la suma de todos sus flujos de caja, cada uno descontado a hoy: $NPV = \Sigma\ C_t \div (1 + r)^t$, sumado para los años t = 0 a n, donde C_t es el flujo de caja neto del año t (negativo para los costos) y r es la tasa de descuento anual. El flujo de caja del año 0 es la inversión inicial y no se descuenta. Un NPV positivo significa que la inversión rinde más que la tasa de descuento; al elegir entre opciones de riesgo similar, la de mayor NPV es la que crea más valor.

El periodo de recuperación (payback) es más sencillo: cuánto tiempo pasa hasta que el flujo de caja acumulado se vuelve positivo. A los directivos les gusta porque es fácil de entender y porque recoge el riesgo de forma aproximada (cuanto antes se recupera el dinero, menos puede salir mal), pero ignora todo lo que ocurre después del punto de recuperación. El periodo de recuperación descontado hace el mismo cálculo con flujos de caja descontados. Presenta tanto el NPV como el periodo de recuperación, y muestra lo sensibles que son a las estimaciones que más importan.

EJEMPLO PRÁCTICO

Ejemplo práctico: el caso de negocio para estrangular el módulo de pedidos

El arquitecto de la plataforma de reparto de comida elabora el caso de negocio para la migración del módulo de pedidos descrita en la Sección 11.6. Finanzas fija la tasa de descuento en el 10%. Los beneficios se estiman a partir de los datos de entrega y de incidentes; los ingresos de nuevas funcionalidades se dejan fuera y solo se mencionan como potencial adicional.

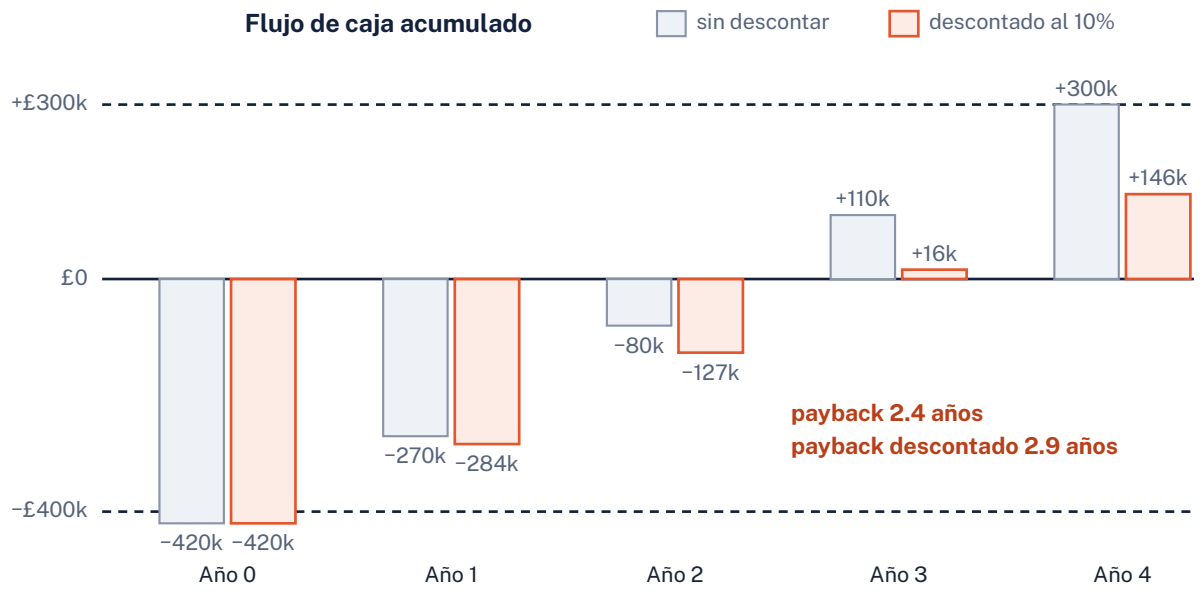
Inversión (año 0)	Seis ingenieros durante unos cinco meses (unos 630 días-ingeniero a la misma tarifa con costos totales de £600 al día, £378 000), más la infraestructura de transición y la ejecución en paralelo (£42 000): £420 000.
Beneficio anual por cambios más rápidos (años 2 a 4)	Unos 100 cambios relacionados con los pedidos al año, cada uno de los cuales ahorra 2 días-desarrollador de coordinación, espera de lanzamientos y retrabajo, a una tarifa con costos totales de £600 al día: 100 × 2 × £600 = £120 000.
Beneficio anual por menos incidentes en los picos	Seis incidentes al año en el pico del viernes, cada uno con un costo de unas £20 000 en reembolsos, vales y pedidos perdidos; se espera que la migración elimine dos tercios de ellos: 4 × £20 000 = £80 000.
Costo de operación adicional neto	Nuevo servicio y broker menos la capacidad liberada del monolito: −£10 000 al año.
Flujos de caja netos	Año 0: −£420 000. Año 1 (los beneficios aumentan gradualmente durante la migración): +£150 000. Años 2, 3 y 4: +£190 000 cada uno.
Flujos de caja descontados	$-420{,}000 + 150{,}000 \div 1.1 + 190{,}000 \div 1.1^2 + 190{,}000 \div 1.1^3 + 190{,}000 \div 1.1^4 =$ $-420{,}000 + 136{,}364 + 157{,}025 + 142{,}750 + 129{,}773$.
Valor actual neto	NPV ≈ +£145 900 en cuatro años. Con una tasa de descuento del 15% seguiría siendo de unas +£87 700.
Recuperación	Flujo de caja acumulado sin descontar: −420 000, −270 000, −80 000, +110 000. Periodo de recuperación = 2 + 80 000 ÷ 190 000 ≈ 2.4 años. Con los flujos descontados, el total acumulado se vuelve positivo en el año 3: 2 + 126 612 ÷ 142 750 ≈ 2.9 años.
Sensibilidad	Si los beneficios son un 30% inferiores a lo estimado, el NPV baja a unas −£23 900. El caso es robusto frente a la tasa de descuento, pero sensible a las estimaciones de beneficios.
Recomendación	Aprobar los pasos 1 a 4 de la migración (£250 000), medir el tiempo de entrega de cambios y los incidentes de pedidos, y liberar la financiación de los pasos 5 y 6 solo si los ahorros medidos respaldan al menos el 80% de la estimación.

El cálculo es sencillo; el valor está en hacer explícitos los supuestos, para que finanzas pueda cuestionarlos y el equipo pueda medirlos.

El análisis de sensibilidad muestra dónde invertir el esfuerzo: validar pronto el ahorro de tiempo de los desarrolladores importa mucho más que afinar la tasa de descuento.

La financiación por etapas convierte una única gran apuesta en una secuencia de opciones, cada una liberada en función de las pruebas.

El periodo de recuperación de un vistazo



Los flujos de caja acumulados se vuelven positivos durante el año 3; el descuento retrasa la recuperación y reduce el valor final a un NPV de unas £146 000.

```
def npv(rate, cash_flows):
    """cash_flows[0] is year 0 (the investment, usually negative)."""
    return sum(c / (1 + rate) ** t for t, c in enumerate(cash_flows))

flows = [-420_000, 150_000, 190_000, 190_000, 190_000]
print(round(npv(0.10, flows)))        # 145911
print(round(npv(0.15, flows)))        # 87663

pessimistic = [flows[0]] + [c * 0.7 for c in flows[1:]]
print(round(npv(0.10, pessimistic)))  # -23862
```

El mismo cálculo en unas pocas líneas, práctico para probar distintos supuestos.

La deuda técnica en dinero y tiempo

La Sección 1.7 explicó la deuda técnica como principal (el costo de corregir un problema) e interés (el costo adicional que se paga en cada cambio mientras el problema sigue ahí), y estimó el interés de un módulo enmarañado. Un caso de negocio necesita esas mismas magnitudes en una forma que finanzas reconozca. El interés se convierte en un costo anual; el principal se convierte en la inversión; y saldar la deuda se convierte en una inversión cuyo rendimiento es el interés que se deja de pagar, más cualquier reducción de riesgos y las oportunidades que se habilitan.

Se pueden convertir varios tipos de pruebas. Tiempo de los desarrolladores: días adicionales por cambio, a partir de comparaciones con cambios similares en otras partes o de estimaciones de los desarrolladores, multiplicados por el volumen de cambios y por una tarifa diaria con costos totales. Retraso: tiempos de entrega más largos multiplicados por el costo del retraso del trabajo afectado. Incidentes: el número y la duración de los incidentes atribuidos a la deuda, multiplicados por su costo. Riesgo: la probabilidad y el impacto de los fallos que la deuda hace más probables, como una brecha de seguridad a través de un componente sin soporte. Oportunidad: las iniciativas de negocio que la deuda bloquea por completo.

Ten cuidado con las cifras de deuda que producen las herramientas de análisis estático, que normalmente indican un esfuerzo de corrección estimado (cuánto se tardaría en corregir cada problema señalado). Eso es una estimación del principal, no del interés, y no dice nada sobre si el código se modifica alguna vez. La deuda en código que rara vez se toca cuesta poco; los hotspots de la Sección 1.7 son donde realmente se pagan los intereses, y donde saldar la deuda tiene el mejor rendimiento. Presentar la deuda como una cartera de pequeñas inversiones, cada una con un rendimiento y un periodo de recuperación estimados, suele ser mucho más persuasivo que una única cifra llamativa de deuda total.

Mapas de Wardley

Los casos de negocio comparan opciones; un **mapa de Wardley**, ideado por Simon Wardley, te ayuda a ver qué opciones existen. El mapa se ancla en un usuario y su necesidad. El eje vertical es la cadena de valor, con lo que el usuario puede ver en la parte superior y aquello de lo que depende debajo. El eje horizontal es la evolución, desde la génesis, pasando por lo construido a medida y el producto, hasta la mercancía (commodity), y los componentes se desplazan hacia la derecha a medida que la competencia los madura. Para la plataforma de reparto de comida, la necesidad es que te entreguen una comida. La app de pedidos está en lo alto; la optimización del despacho se construye a medida y diferencia; los pagos con tarjeta son un producto; la computación es una commodity. Construir a mano algo situado muy a la derecha es una señal de alerta. El GOV.UK Service Manual sugiere hacer mapas para saber qué construir y qué comprar. Las posiciones son juicios, así que el valor está en la conversación.

Construir, comprar o algo intermedio

El consejo del Government Digital Service es un valor predeterminado sólido: compra los servicios comunes y construye donde la capacidad sea central para tu organización. Esto coincide con los subdominios principales y genéricos de la Sección 5.7. La plataforma de reparto de comida compraría los pagos y la mensajería, pero construiría el despacho. Entre esos polos están adoptar código abierto, configurar un producto y asociarse con otros. GDS advierte de que la personalización intensiva es cara y hace que los productos sean más difíciles de actualizar, así que prefiere la configuración. También advierte de que *no puedes externalizar el riesgo*: sea cual sea la vía que elijas, sigues necesitando las personas y el conocimiento para supervisarla. Compara las opciones a lo largo de varios años, con los métodos anteriores, y cuenta la integración, las operaciones, el tiempo del personal, las subidas de precio y el costo de salir (Sección 10.8). Mantén al proveedor detrás de tu propia interfaz. Revisa la decisión a medida que evolucionan los componentes.

Evaluar proveedores y realizar una prueba de concepto

Fija los criterios antes de la primera demostración, o los fijará el mejor vendedor. Separa los criterios eliminatorios de los puntuados, y pondera estos últimos como en la Sección 1.6. Los eliminatorios suelen incluir la ubicación de los datos, las condiciones de salida y las pruebas de seguridad. Cuando un proveedor trata datos personales, el artículo 28(1) del RGPD del Reino Unido (UK GDPR) solo permite recurrir a encargados del tratamiento que ofrezcan garantías suficientes. Después, pon a prueba la afirmación más arriesgada con una **prueba de concepto**: una pregunta, un plazo fijo y una nota de aprobado acordada de antemano. Usa tus propios datos y tu carga de pico, e incluye los fallos: ¿qué hace un nuevo proveedor de mapas para el despacho cuando su API se ralentiza durante la cena del viernes? Prueba al menos dos candidatos cuando puedas. Trata el código como desechable, y acuerda de antemano qué resultado pondría fin a la evaluación, para que no decida el esfuerzo ya invertido.

Ética: qué piden los códigos profesionales

Las decisiones arquitectónicas determinan lo que los sistemas hacen a las personas que los usan y a las personas a las que afectan sin su consentimiento, así que los arquitectos tienen responsabilidades éticas además de técnicas. Hay dos códigos profesionales especialmente pertinentes. El Código de Ética y Conducta Profesional de la ACM (ACM Code of Ethics and Professional Conduct), actualizado por la Association for Computing Machinery en 2018, empieza con unos principios generales: contribuir a la sociedad y al bienestar humano, reconociendo que todas las personas son partes interesadas en la informática; evitar daños; ser honesto y digno de confianza; ser justo y actuar para no discriminar; respetar el trabajo necesario para producir ideas nuevas; respetar la privacidad; y respetar la confidencialidad. Entre sus responsabilidades profesionales están ofrecer evaluaciones completas y rigurosas

de los sistemas informáticos y de sus impactos, incluido el análisis de los posibles riesgos, y diseñar sistemas que sean seguros de forma robusta y utilizable; sus principios de liderazgo piden un cuidado especial con los sistemas que pasan a integrarse en la infraestructura de la sociedad.

El Código de Conducta de la BCS, de BCS, The Chartered Institute for IT, del Reino Unido, se organiza en torno a cuatro deberes: hacia el interés público (incluidos la salud pública, la privacidad, la seguridad, el bienestar, el medio ambiente y la igualdad de acceso), hacia la competencia y la integridad profesionales, hacia la autoridad pertinente (el empleador o el cliente) y hacia la profesión. El Código de Ética de la Ingeniería de Software conjunto de la ACM y la IEEE Computer Society (1999) añade ocho principios, empezando por el interés público, que prevalece sobre los intereses del cliente y del empleador.

Para los arquitectos, estos códigos se traducen en obligaciones concretas. Analiza los efectos previsibles de un diseño, no solo su corrección funcional. Informa con honestidad a los responsables de decisión sobre los riesgos, incluidos los que preferirían no oír. Niégate a diseñar sistemas destinados a engañar o a causar daño, y escala la cuestión cuando te lo pidan. Registra en el registro de decisiones las discrepancias por motivos éticos. Y sabe a quién acudir si la escalada interna fracasa: en el Reino Unido, por ejemplo, la Public Interest Disclosure Act 1998 protege a los trabajadores que denuncian determinados tipos de irregularidades.

Daños previsibles y respuestas arquitectónicas

Daño	Ejemplo de reparto de comida	Respuestas arquitectónicas
Privacidad	Direcciones particulares e historiales de pedidos de los clientes conservados indefinidamente y compartidos ampliamente con analítica	Minimización de datos, límites de conservación, acceso según la finalidad, seudonimización; consulta el Capítulo 8 sobre la privacidad desde el diseño
Seguridad	Documentos de identidad de los repartidores en un bucket accesible desde internet	Modelado de amenazas, mínimo privilegio, cifrado, gestión de secretos, controles de la cadena de suministro
Sesgo y resultados injustos	Un algoritmo de despacho que ofrece sistemáticamente menos encargos a los repartidores de ciertas zonas o con vehículos más antiguos	Criterios de equidad en los requisitos, monitorización de los resultados por grupo, explicabilidad, revisión humana de las decisiones impugnadas
Accesibilidad	Un proceso de pago que no se puede completar con un lector de pantalla	WCAG 2.2 nivel AA como restricción, componentes accesibles en el sistema de diseño, pruebas automatizadas y manuales
Seguridad (safety)	Incentivos que premian la velocidad de los repartidores, o información sobre alérgenos que se muestra de forma incoherente	Requisitos de seguridad (safety) y análisis de peligros, límites a los incentivos, una única fuente de verdad para los alérgenos
Manipulación	Comisiones de servicio ocultas que solo se muestran en el último paso; suscripciones fáciles de contratar y difíciles de cancelar	Precios transparentes en el contrato de la API, recorridos de alta y de baja simétricos, revisión del diseño persuasivo

Los patrones oscuros y otros daños previsibles

La tabla enumera los daños que con más frecuencia están al alcance de un arquitecto. Varios se tratan a fondo en otras partes: la privacidad y la seguridad en el Capítulo 8, la accesibilidad como cualidad en la Sección 2.7 y el sesgo en los sistemas de aprendizaje automático en el Capítulo 12. La accesibilidad es cada vez más una restricción legal además de ética; en la UE, el Acta Europea de Accesibilidad (European Accessibility Act) se aplica a muchos servicios de consumo, incluido el comercio electrónico, desde el 28 de junio de 2025.

El diseño manipulador merece una mención especial porque a menudo se solicita deliberadamente. Harry Brignull acuñó en 2010 el término patrón oscuro (dark pattern) para las interfaces diseñadas para engañar a las personas y llevarlas a decisiones que de otro modo no tomarían; ahora prefiere el término patrones engañosos (deceptive patterns). Entre los

ejemplos conocidos están los precios por goteo (drip pricing, comisiones que solo se revelan al final), el confirmshaming (hacer que la opción de decir que no resulte vergonzosa), la falsa urgencia y escasez, las casillas de consentimiento premarcadas y las suscripciones fáciles de iniciar y difíciles de cancelar. Los reguladores han actuado: la Ley de Servicios Digitales de la UE (Digital Services Act, 2022) prohíbe a las plataformas en línea diseñar interfaces que engañen o manipulen a sus usuarios, y la Federal Trade Commission de EE. UU. publicó en 2022 un informe de su personal técnico sobre los patrones oscuros. Muchos de estos patrones necesitan soporte arquitectónico (una API de precios que oculta las comisiones hasta el proceso de pago, un flujo de cancelación que pasa por un servicio de retención), así que los arquitectos están en posición de detectarlos y cuestionarlos.

La forma más sencilla de hacer visibles los daños es buscarlos a propósito, en los mismos momentos en que buscas los riesgos técnicos. Añade una pregunta sobre daños al modelado de amenazas y a las revisiones de diseño: ¿quién podría resultar perjudicado por este sistema si funciona tal como está diseñado, si falla o si se usa de forma indebida? Prácticas como el consequence scanning (análisis de consecuencias), desarrollado por el think tank británico Doteveryone, dan a los equipos un formato breve y estructurado para esta conversación durante la planificación.

Sostenibilidad: software verde

El software no quema combustible, pero la electricidad y el hardware que usa sí, y las decisiones arquitectónicas fijan gran parte de esa huella durante años. La Green Software Foundation, creada en 2021 bajo la Linux Foundation, publica seis áreas fundamentales de las prácticas de software verde: la eficiencia en carbono (emitir el menor carbono posible), la eficiencia energética (usar la menor energía posible), la conciencia del carbono (hacer más cuando la electricidad es más limpia y menos cuando es más sucia), la eficiencia del hardware (usar el menor carbono incorporado posible), la medición y los compromisos climáticos (entender exactamente cómo se logra una reducción).

La computación consciente del carbono aprovecha el hecho de que la intensidad de carbono de la electricidad varía según la hora y la región con la combinación de eólica, solar, gas y otras fuentes. El desplazamiento de la demanda (demand shifting) traslada el trabajo flexible, como los procesos por lotes, el entrenamiento de modelos, las copias de seguridad y la generación de informes, a momentos o regiones en los que la electricidad es más limpia; la modulación de la demanda (demand shaping) reduce lo que hace el software cuando es más sucia, por ejemplo sirviendo contenido multimedia de menor resolución o aplazando los recálculos no urgentes. Los servicios de datos de la red eléctrica, como la Carbon Intensity API para Gran Bretaña, y las herramientas de código abierto, como el Carbon Aware SDK de la fundación,

proporcionan las señales. Como ilustración sencilla, un proceso por lotes nocturno que usa 400 kWh emite unos 120 kg de CO_2e a unos ilustrativos 300 g por kWh a primera hora de la noche, pero unos 48 kg si se traslada a una noche ventosa a 120 g por kWh.

El carbono incorporado, es decir, las emisiones de fabricar y desechar el hardware, es la razón por la que importa la eficiencia del hardware: un mayor aprovechamiento de menos máquinas, escalar a cero cuando no hay actividad y alargar la vida útil de los dispositivos (incluidos los teléfonos de los clientes, al no exigir hardware nuevo en cada lanzamiento) lo reducen. La especificación Software Carbon Intensity de la fundación, publicada como ISO/IEC 21031:2024 y presentada en la Sección 2.7, combina las emisiones operativas y las incorporadas por unidad funcional, como por pedido entregado, y excluye las compensaciones (offsets) para que solo las reducciones reales mejoren la puntuación.

Entre las palancas arquitectónicas están el ajuste del tamaño (right-sizing) y el autoescalado, eliminar los entornos ociosos, los límites de conservación de datos (los datos almacenados tienen un costo en energía y en hardware), los formatos de datos eficientes y la caché, elegir regiones con redes eléctricas más limpias cuando la latencia y la residencia de datos lo permitan, y cuestionar si cada funcionalidad necesita una llamada a un modelo grande. Cuidado con el efecto rebote, la versión moderna de la observación de William Stanley Jevons sobre el carbón en 1865: la eficiencia que abarata algo puede aumentar su uso total, así que mide las emisiones totales además de la intensidad.

Lista de comprobación: una revisión ética y sostenible de la arquitectura

Úsala junto con los métodos de revisión técnica de las Secciones 11.1 y 11.2. Cada punto debería producir una decisión registrada o una acción con un responsable.

- [ ] **Identifica a quién afecta**
  Enumera las personas afectadas por el sistema, incluidas las que nunca lo usan directamente (repartidores, vecinos, personas cuyos datos se tratan), y sus intereses.
- [ ] **Pregunta cómo podría causar daño**
  Para los casos de funcionamiento según el diseño, de fallo y de uso indebido, pregunta quién podría resultar perjudicado y cómo; registra las respuestas junto con el modelo de amenazas.
- [ ] **Minimiza los datos personales**
  Comprueba que cada dato personal tenga una finalidad, un plazo de conservación y una regla de acceso, y que el sistema pueda cumplir con su eliminación.
- [ ] **Comprueba la equidad de las decisiones automatizadas**
  Identifica las decisiones que toman o clasifican los algoritmos, define criterios de equidad y planifica cómo se monitorizarán los resultados por grupo y cómo podrán impugnarse.
- [ ] **Confirma la accesibilidad**
  Trata WCAG 2.2 nivel AA (o la norma legal aplicable) como una restricción y confirma cómo se probará.
- [ ] **Busca el diseño manipulador**
  Revisa los recorridos de precios, consentimiento, alta y cancelación en busca de costos ocultos, casillas premarcadas, falsa urgencia y asimetría.
- [ ] **Revisa las implicaciones de seguridad (safety)**
  Identifica cualquier consecuencia física o para la salud, como los alérgenos, los incentivos o el uso de vehículos, y los controles que las rodean.
- [ ] **Estima la huella de carbono**
  Estima la energía, la intensidad de carbono de la red y las emisiones incorporadas por unidad funcional (SCI), e identifica los factores que más contribuyen.
- [ ] **Usa opciones conscientes del carbono**
  Identifica el trabajo que puede desplazarse en el tiempo o de región, la capacidad ociosa que puede eliminarse y los datos que pueden borrarse antes.

- [ ] **Haz que el caso de negocio sea honesto**
  Incluye los costos de operación, la línea de base de no hacer nada y un análisis de sensibilidad; separa los beneficios medidos del potencial adicional que se espera.
- [ ] **Registra las discrepancias y la escalada**
  Registra las objeciones éticas en el registro de decisiones y conoce la vía para escalar las inquietudes que no se resuelvan.
- [ ] **Planifica la medición**
  Elige indicadores (informes de daños, defectos de accesibilidad, SCI, costo por pedido) y una fecha para revisarlos frente a las expectativas anteriores.

ATENCIÓN

Errores comunes

- Suponer que la opción de no hacer nada no cuesta nada, lo que hace que toda inversión parezca peor de lo que es.
- Contar dos veces el mismo beneficio, por ejemplo como tiempo de desarrollo ahorrado y otra vez como entrega más rápida de las mismas funcionalidades.
- Falsa precisión: un NPV calculado a la libra con estimaciones que tienen un margen del 30%.
- Omitir los costos de operación, los costos de transición y el desmantelamiento.
- Tratar la ética como una casilla de cumplimiento al final en lugar de como una pregunta que se plantea durante el diseño.
- Atribuirse reducciones de carbono mediante compensaciones o cambios contables en lugar de mediante reducciones reales de energía y de hardware.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué un caso de negocio debe incluir la opción de no hacer nada?**

Porque no hacer nada tiene sus propios costos y riesgos crecientes, que forman la línea de base con la que se miden los beneficios de todas las demás opciones.

▼ **¿Por qué debería un arquitecto usar la tasa de descuento que fija finanzas en lugar de elegir una?**

Porque refleja el costo de capital y la política de riesgos de la organización, y usar la misma tasa hace que las inversiones de arquitectura sean comparables con todas las demás inversiones que compiten por el dinero.

▼ **¿Qué es el carbono incorporado? Nombra una forma arquitectónica de reducirlo.**

Las emisiones de fabricar, transportar y desechar el hardware; disminuye con un mayor aprovechamiento de menos máquinas, escalando a cero cuando no hay actividad o alargando la vida útil de los dispositivos.

▼ **¿Cuál es la diferencia entre el desplazamiento y la modulación de la demanda?**

El desplazamiento de la demanda traslada el trabajo flexible a momentos o regiones más limpios; la modulación de la demanda cambia cuánto trabajo hace el software según lo limpia que sea la electricidad.

▼ **Inténtalo: Una migración cuesta £200 000 ahora y ahorra £90 000 al año durante tres años. La tasa de descuento de Finanzas es del 10%. Calcula el NPV y la recuperación (payback). Luego prueba con ahorros de £75 000 al año. ¿Qué le dices al consejo?**

VAN con £90 000: 81 818 + 74 380 + 67 618 - 200 000 = unos +£23 800. Periodo de recuperación: unos 2.2 años. Con £75 000, el VAN es de unos -£13 500, de modo que el caso se vuelve negativo. Dile al consejo que supera la tasa, pero por poco, y que depende de la estimación de ahorro. Recomienda una financiación por etapas: financia una primera porción, mide los ahorros reales y después decide sobre el resto. Muestra también la opción de no hacer nada.

FUENTES

- ACM Code of Ethics and Professional Conduct: https://www.acm.org/code-of-ethics
- BCS Code of Conduct: https://www.bcs.org/membership-and-registrations/become-a-member/bcs-code-of-conduct/
- Green Software Foundation: Principles of green software: https://learn.greensoftware.foundation/introduction
- Software Carbon Intensity (SCI) specification: https://sci.greensoftware.foundation/
- Harry Brignull: Deceptive Patterns: https://www.deceptive.design/
- Finding a path (Simon Wardley, chapter 2): https://medium.com/wardleymaps/finding-a-path-cdb1249078c0

- On being lost (Simon Wardley, chapter 1): https://medium.com/wardleymaps/on-being-lost-2ef5f05eb1ec
- The Four Stages of Evolution (wardleymaps.com): https://www.wardleymaps.com/book-passages/wardley-on-evolution
- Choosing technology: an introduction (GOV.UK Service Manual): https://www.gov.uk/service-manual/technology/choosing-technology-an-introduction
- To build or to buy, that's the technology question (GDS, 2021): https://technology.blog.gov.uk/2021/02/03/to-build-or-to-buy-thats-the-technology-question
- UK GDPR Article 28: https://www.legislation.gov.uk/eur/2016/679/article/28

Lecturas recomendadas

- *Software Architecture in Practice, 4th edition*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. El libro de texto de referencia; sus capítulos sobre evaluación de la arquitectura describen el ATAM y su variante ligera, de la mano de los autores del método.
- *ATAM: Method for Architecture Evaluation (CMU/SEI-2000-TR-004)*, Rick Kazman, Mark Klein and Paul Clements (2000), Software Engineering Institute, Carnegie Mellon University. El informe técnico original que define los pasos, los roles y los resultados del ATAM, con un caso práctico.
  https://www.sei.cmu.edu/library/atam-method-for-architecture-evaluation/
- *Building Evolutionary Architectures: Automated Software Governance, 2nd edition*, Neal Ford, Rebecca Parsons, Patrick Kua and Pramod Sadalage (2022), O'Reilly Media. El origen de las funciones de aptitud arquitectónicas y la exposición más clara de la arquitectura evolutiva.
  https://oreilly.com/library/view/building-evolutionary-architectures/9781492097532
- *Performing a Project Premortem*, Gary Klein (2007), Harvard Business Review. El breve artículo que presentó el pre-mortem: imaginar primero el fracaso y después prevenirlo.
  https://hbr.org/2007/09/performing-a-project-premortem
- *Scaling the Practice of Architecture, Conversationally*, Andrew Harmel-Law (2021), martinfowler.com. Describe el proceso de consejo, los foros asesores, los principios elaborados por los equipos y un radar tecnológico interno.
  https://martinfowler.com/articles/scaling-architecture-conversationally.html
- *Design It! From Programmer to Software Architect*, Michael Keeling (2017), Pragmatic Bookshelf. Una guía práctica de la arquitectura como actividad de equipo, con muchas actividades de taller, diseño y evaluación para equipos.
- *Kill It with Fire: Manage Aging Computer Systems (and Future Proof Modern Ones)*, Marianne Bellotti (2021), No Starch Press. Una guía franca y basada en la experiencia para modernizar sistemas heredados sin una reescritura condenada al fracaso.

Glosario

API del equipo
Todo lo que un equipo ofrece a otros equipos y espera de ellos: sus interfaces de código, su documentación, sus reglas de versionado, sus formas de trabajar, sus canales de comunicación y cómo pedirle trabajo.

Arquitectura de transición
Software temporal, como capas de enrutamiento, adaptadores y tareas de sincronización, construido solo para que los sistemas antiguos y nuevos coexistan durante una migración y que se elimina cuando esta termina.

Arquitectura evolutiva
Una arquitectura que admite cambios guiados e incrementales en múltiples dimensiones, con funciones de aptitud que proporcionan la guía (Ford, Parsons, Kua y Sadalage).

Arquitectura sociotécnica
Diseñar a la vez la estructura del software y la estructura de los equipos, porque cada una da forma a la otra. La idea se remonta a la investigación sobre sistemas sociotécnicos del Tavistock Institute, que empezó con el estudio de 1951 de Eric Trist y Ken Bamforth sobre la minería del carbón.

Ascensor del arquitecto (architect elevator)
La imagen de Gregor Hohpe del arquitecto como alguien que se desplaza entre el ático de una organización (la estrategia y los directivos) y su sala de máquinas (el código y la infraestructura), conectando ambos.

ATAM (Architecture Tradeoff Analysis Method)
El método de evaluación basado en escenarios del SEI (Kazman, Klein y Clements, 2000). En nueve pasos a lo largo de dos reuniones, relaciona los impulsores de negocio y los escenarios priorizados con los enfoques arquitectónicos, y produce riesgos, no riesgos, puntos de sensibilidad, puntos de compromiso y temas de riesgo.

Branch by abstraction (ramificación por abstracción)
Una técnica para cambios grandes dentro de una misma base de código: poner una abstracción delante de la parte que se va a sustituir, trasladar a ella a quienes la llaman, construir la nueva implementación detrás, cambiar a ella y eliminar la antigua, manteniendo el sistema listo para lanzarse en todo momento.

Camino pavimentado (paved road, golden path)
Una forma predeterminada, con soporte y bien documentada, de construir y ejecutar software, como una plantilla de servicio con el registro de logs, la seguridad y el despliegue ya configurados, que los equipos pueden abandonar cuando tienen un buen motivo.

Capa anticorrupción
Una capa de traducción, procedente del diseño guiado por el dominio, que convierte entre el modelo de otro sistema y el tuyo, para que los conceptos y peculiaridades del otro modelo no se filtren en tu código.

Carbono incorporado
Los gases de efecto invernadero emitidos al fabricar, transportar y desechar el hardware, a diferencia de las emisiones de la electricidad que consume mientras funciona.

Caso de negocio (business case)
Un argumento estructurado a favor de una inversión que expone el problema, las opciones (incluida no hacer nada), sus costos, beneficios y riesgos, y una recomendación que los responsables de decidir pueden comparar con otros usos del dinero.

CBAM (Cost Benefit Analysis Method)
Un método del SEI que se basa en el ATAM y estima el beneficio de cada estrategia arquitectónica a partir de la utilidad que añade a los escenarios priorizados, dividiéndolo por su costo para comparar la relación entre valor y costo.

Código de Ética de la ACM (ACM Code of Ethics)
El Código de Ética y Conducta Profesional de la Association for Computing Machinery, actualizado en 2018, cuyos principios incluyen contribuir a la sociedad, evitar daños, la honestidad, la equidad, la privacidad y la confidencialidad.

Compuerta de gobernanza (gate)
Un punto de control en el que el trabajo se detiene hasta que alguien lo aprueba, como el visto bueno de una junta de revisión antes de la construcción o el lanzamiento.

Computación consciente del carbono
Ejecutar software de modo que haga más cuando la electricidad es más limpia y menos cuando es más contaminante, desplazando el trabajo en el tiempo o en la ubicación, o modulando la demanda.

Consenso
Una regla de decisión en la que el grupo continúa hasta que todos están activamente de acuerdo con la propuesta, o al menos la apoyan; es inclusiva pero lenta, y propensa al bloqueo en grupos grandes.

Costo del retraso
El valor que se pierde por cada unidad de tiempo en que un trabajo no se entrega; Don Reinertsen lo popularizó como forma de priorizar el trabajo con criterios económicos.

Dark launching (lanzamiento oculto)
Invocar un nuevo comportamiento del backend en producción con tráfico real ocultando sus resultados a los usuarios, para probar el rendimiento y la corrección antes de un lanzamiento visible.

Derechos de decisión
Una declaración explícita de quién puede tomar qué decisiones, a quién hay que consultar y a quién hay que informar; fijarlos antes de un debate evita la mayoría de las disputas sobre el proceso.

Disagree and commit (discrepar y comprometerse)
Una norma según la cual las personas expresan abiertamente su desacuerdo antes de una decisión y, una vez tomada, se comprometen plenamente a llevarla a cabo.

Ejecución en paralelo
Ejecutar la implementación antigua y la nueva lado a lado con las mismas entradas y comparar sus resultados, mientras la antigua sigue siendo la autoritativa, para demostrar que la nueva es equivalente antes del cambio.

Enfoque arquitectónico
Una decisión de diseño significativa o una combinación de decisiones, como un estilo, un patrón o una táctica, que la arquitectura usa para lograr una cualidad: por ejemplo, un intermediario de eventos con un outbox o una región de reserva en caliente (warm standby).

Equipo alineado al flujo (stream-aligned)
Equipo estable y multifuncional alineado con un flujo de cambio, como un producto, un recorrido de usuario o una capacidad de negocio, que puede construir, operar y cambiar su parte de extremo a extremo. La mayoría de los equipos de una organización deberían ser de este tipo.

Equipo de plataforma
Equipo que proporciona servicios internos, herramientas e infraestructura como un producto de autoservicio, para que los equipos alineados al flujo puedan entregar sin un conocimiento profundo de la tecnología subyacente.

Equipo de subsistema complicado
Un equipo responsable de una parte del sistema que requiere conocimientos especializados escasos, como un algoritmo de optimización o un códec de vídeo, para que los equipos alineados con el flujo puedan usarla sin dominarla.

Equipo facilitador
Un equipo de especialistas que ayuda a otros equipos a adquirir una capacidad que les falta, como la automatización de pruebas o la observabilidad, durante un periodo limitado y luego pasa a otra cosa, en lugar de hacer el trabajo por ellos.

Evaluación de la arquitectura
Un examen estructurado de una arquitectura, antes o después de construirla, para averiguar si sus decisiones permitirán que el sistema cumpla sus objetivos de negocio y sus requisitos de calidad, y dónde ponen en riesgo esos objetivos.

Experimento de caos
Una prueba controlada que inyecta un fallo real, como eliminar una instancia o ralentizar una dependencia, para comprobar una hipótesis formulada sobre cómo el sistema mantiene su estado estable.

Foro asesor de arquitectura
Una reunión periódica y abierta en la que se debaten las decisiones propuestas, los spikes y los consejos recibidos. Ofrece consejo y visibilidad, pero no decide por sí misma.

Función de aptitud atómica
Una función de aptitud que comprueba una característica arquitectónica en un único contexto, como una prueba que prohíbe las importaciones entre dos paquetes.

Función de aptitud continua
Una función de aptitud que se ejecuta todo el tiempo contra el sistema en funcionamiento, como la supervisión de SLO o las transacciones sintéticas, en lugar de activarse por una compilación o un evento.

Función de aptitud holística
Una función de aptitud que comprueba varias características juntas en un contexto compartido, como una prueba de carga que mide la latencia con la seguridad y el registro activados.

Gobernanza de la arquitectura
Las prácticas mediante las que una organización se asegura de que las decisiones técnicas significativas se toman bien, son coherentes con sus objetivos y se cumplen realmente, incluido quién decide, cómo y con qué controles.

Impulsores de negocio (business drivers)
Los objetivos de negocio, las restricciones y el contexto que explican por qué existe un sistema y qué significa el éxito, como objetivos de crecimiento, normativa o contratos. En el ATAM se presentan primero para que cada hallazgo pueda vincularse con ellos.

Intercepción de eventos
Un patrón de desplazamiento de sistemas heredados que captura las solicitudes o los eventos que cambian el estado en su camino hacia un sistema heredado y reenvía algunos de ellos también a un nuevo componente, o en su lugar.

Junta de revisión de arquitectura
Un grupo permanente de arquitectos sénior que revisa y aprueba los diseños significativos, normalmente en momentos fijos de un proyecto. Puede aportar un juicio independiente, pero a menudo se convierte en un punto de control lento.

Kata de arquitectura
Un ejercicio práctico creado por Ted Neward en el que pequeños equipos diseñan una arquitectura para un enunciado breve y deliberadamente incompleto, y luego la presentan y defienden ante sus compañeros.

Las siete R (Seven Rs)
Las estrategias de migración que AWS enumera para trasladar aplicaciones a la nube: retire (retirar), retain (conservar), rehost (realojar), relocate (reubicar), repurchase (recomprar), replatform (cambiar de plataforma) y refactor o re-architect (refactorizar o rediseñar la arquitectura). Surgieron de las cinco R anteriores de Gartner.

Modo de interacción
En Team Topologies, una de las tres formas deliberadas en que dos equipos trabajan juntos: colaboración, X como servicio (X-as-a-Service) o facilitación. Se espera que los modos cambien a medida que los límites se descubren y se asientan.

No riesgo
Decisión considerada sólida para un escenario dado, registrada junto con los supuestos de los que depende, para poder reexaminarla si esos supuestos cambian.

Patrón oscuro (dark pattern)
Una interfaz de usuario diseñada para engañar o presionar a las personas para que tomen decisiones que de otro modo no tomarían, como costos ocultos o suscripciones difíciles de cancelar; Harry Brignull acuñó el término en 2010 y ahora los llama patrones engañosos (deceptive patterns).

Periodo de recuperación (payback)
Cuánto tiempo tardan los beneficios acumulados en devolver el costo de una inversión; el periodo de recuperación descontado hace lo mismo con flujos de caja descontados.

Plano de fractura
Una línea divisoria natural por la que puede dividirse un sistema de software para que cada parte se ajuste a un equipo, por ejemplo, un contexto delimitado, un límite regulatorio, un ritmo de cambio distinto o un perfil de riesgo distinto.

Pre-mortem
Técnica de Gary Klein (2007) en la que un equipo imagina que su proyecto ya ha fracasado y cada persona escribe por qué, lo que saca a la luz riesgos que de otro modo la gente se resiste a plantear.

Presupuesto de latencia
La parte de un objetivo de tiempo de respuesta de extremo a extremo asignada a cada paso o servicio en el recorrido de una solicitud, para que cada equipo conozca el límite que debe respetar.

Principio de arquitectura
Una regla duradera que orienta las decisiones arquitectónicas, redactada al estilo de TOGAF con un nombre, un enunciado, una justificación y sus implicaciones.

Prueba de caracterización
Una prueba que registra lo que el código existente hace realmente, y no lo que debería hacer, para que los cambios puedan contrastarse con el comportamiento actual. El término procede de Michael Feathers.

Punto de sensibilidad
Propiedad de uno o más componentes o conexiones que es crítica para lograr una respuesta de calidad concreta, como el número de saltos síncronos en la ruta de un pedido.

Radar tecnológico
Mapa de tecnologías y técnicas que se actualiza periódicamente, agrupadas en cuadrantes y situadas en anillos como Adopt, Trial, Assess y Hold (rebautizado Caution en ediciones recientes de Thoughtworks), popularizado por Thoughtworks.

Reescritura big-bang
Sustituir un sistema construyendo uno nuevo desde cero y cambiando a él de una sola vez; se considera ampliamente de alto riesgo porque el valor llega solo al final y es fácil perder comportamientos ocultos.

Registro de riesgos
Lista mantenida de los riesgos identificados con su probabilidad, impacto, responsable, mitigación y estado, para que se sigan hasta su cierre en lugar de olvidarse después de una reunión.

Revisión de diseño (revisión de RFC)
Una revisión escrita de un documento de diseño propuesto o de una solicitud de comentarios, en la que las personas afectadas comentan de forma asíncrona antes de que se apruebe el diseño y se registren sus decisiones.

Riesgo (en la evaluación de arquitectura)
Decisión arquitectónica, o decisión que falta, que puede impedir que el sistema cumpla un requisito de calidad o un objetivo de negocio importante.

Risk-storming
Técnica colaborativa de Simon Brown en la que las personas marcan en silencio los riesgos sobre diagramas de arquitectura con notas adhesivas, puntuados por probabilidad e impacto, y después convergen y acuerdan prioridades y mitigaciones.

Seguridad psicológica
Creencia compartida dentro de un equipo de que es seguro asumir riesgos interpersonales, como admitir errores o cuestionar a un colega de mayor rango; estudiada por Amy Edmondson desde 1999.

Sistema heredado (legacy)
Sistema existente que sigue aportando valor pero que se ha vuelto difícil, arriesgado o caro de cambiar, a menudo por su antigüedad, la falta de pruebas, la escasez de conocimientos o una tecnología sin soporte.

Software Carbon Intensity (SCI)
La especificación de la Green Software Foundation, publicada como ISO/IEC 21031:2024, que califica el software como ((E × I) + M) por R: energía multiplicada por la intensidad de carbono de la red eléctrica más las emisiones incorporadas, por unidad funcional.

Staff-plus engineer
Contribuidor individual sénior (staff, principal o distinguished engineer) que ejerce liderazgo técnico entre equipos sin dirigir personas; término popularizado por Will Larson.

Strangler fig (higuera estranguladora)
Enfoque de modernización incremental, bautizado por Martin Fowler, en el que componentes nuevos asumen gradualmente el comportamiento de un sistema heredado detrás de una capa de interceptación hasta que el sistema antiguo puede apagarse.

TARA
Tiny Architectural Review Approach, el método ligero de Eoin Woods para revisar un sistema existente combinando el juicio experto con el análisis automatizado de su código y de sus datos de ejecución.

Tasa de descuento
La tasa anual que se usa para convertir el dinero futuro a su valor actual, que refleja el costo del capital y el riesgo; normalmente la fija el departamento de finanzas.

Team Topologies
Enfoque para organizar los equipos de software con vistas a un flujo rápido, expuesto por Matthew Skelton y Manuel Pais en el libro del mismo nombre (2019; segunda edición 2025): cuatro tipos de equipo, tres modos de interacción y la carga cognitiva del equipo como principal restricción.

Tema de riesgo
Grupo de riesgos relacionados que comparten una causa subyacente, vinculado a los impulsores de negocio a los que amenaza. Los temas de riesgo suelen ser el resultado más útil de una evaluación para los directivos.

Toma de decisiones por consentimiento
Una regla de decisión en la que se adopta una propuesta salvo que alguien plantee una objeción razonada de que causaría daño o haría retroceder al grupo; pregunta si las personas pueden convivir con una propuesta, no si la prefieren.

Valor actual neto (NPV)
La suma de los flujos de caja futuros de una inversión, cada uno descontado a su valor actual, menos el costo inicial; un NPV positivo significa que la inversión rinde más que la tasa de descuento.

X como servicio (X-as-a-Service)
Modo de interacción en el que un equipo consume algo que proporciona otro equipo, a través de una interfaz clara y documentada y con poca colaboración en el día a día.