BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 2

# Atributos de calidad y requisitos arquitectónicamente significativos

La arquitectura existe para lograr cualidades: velocidad bajo carga, seguir funcionando cuando fallan algunas partes, poder cambiarse con seguridad, mantener los datos en secreto. Este capítulo muestra cómo convertir deseos vagos como 'rápido', 'fiable' y 'seguro' en requisitos precisos y comprobables que puedan guiar las decisiones de diseño. Trata la diferencia entre requisitos funcionales, atributos de calidad y restricciones, el modelo de calidad ISO/IEC 25010, los escenarios de atributos de calidad, la aritmética del rendimiento y de la disponibilidad, el diseño para el cambio, la seguridad, la usabilidad y el costo como cualidades, y cómo elicitarlas, priorizarlas y establecer compromisos entre ellas. Es adecuado para desarrolladores que se acercan a la arquitectura y para cualquiera que redacte o revise requisitos.

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

**DE UN VISTAZO**

**Aprenderás a**

- Distinguir los requisitos funcionales, los atributos de calidad y las restricciones, y reconocer un requisito arquitectónicamente significativo.
- Usar el modelo de calidad del producto de ISO/IEC 25010:2023 como lista de comprobación sin dejar que se convierta en papeleo.
- Escribir escenarios de atributos de calidad de seis partes con medidas de la respuesta medibles, y convertir enunciados vagos en enunciados comprobables.
- Calcular con percentiles, la ley de Little, fórmulas sencillas de colas y la ley de Amdahl para razonar sobre el rendimiento y la escalabilidad.
- Calcular la disponibilidad a partir de los nueves, a partir del MTBF y el MTTR, y para componentes en serie y en paralelo.
- Explicar cómo se especifican y se logran la modificabilidad, la capacidad de prueba, la desplegabilidad, la seguridad, la usabilidad, el costo y la sostenibilidad.
- Elicitar y priorizar requisitos de calidad con un árbol de utilidad, y explicar los compromisos entre cualidades que compiten entre sí.

**Antes de empezar**

El Capítulo 1, en especial las secciones sobre las partes interesadas y sus inquietudes y sobre las decisiones y los compromisos. Cierta soltura con la aritmética sencilla (porcentajes, multiplicación y división).

**Tiempo**

Unas 3 horas de lectura; 45 minutos para la prueba.

**Términos clave**

atributo de calidad · restricción · requisito arquitectónicamente significativo · ISO/IEC 25010 · escenario de atributo de calidad · medida de la respuesta · latencia y rendimiento (throughput) · percentil · Ley de Little · disponibilidad · MTTR · táctica · árbol de utilidad · función de aptitud (fitness function)

## Por qué son las cualidades, y no las funcionalidades, las que deciden la arquitectura

Pregunta a cualquier grupo de partes interesadas qué debe hacer un sistema y obtendrás una lista de funcionalidades: buscar un vuelo, pagar un carrito, reservar una cita. Pregunta con qué calidad debe hacerlo y las respuestas suelen ser más vagas: debe ser rápido, no debe caerse, debe ser seguro, debe ser fácil de cambiar. Esas segundas respuestas son atributos de calidad, y son de lo que trata principalmente la arquitectura. La misma lista de funcionalidades puede ofrecerla un único programa en un servidor o cuarenta servicios repartidos por tres

continentes. Lo que separa esos diseños no es lo que hace el sistema, sino a cuántas personas debe atender a la vez, cuánto tiempo de inactividad puede tolerar el negocio, con qué frecuencia cambiará el código y lo sensibles que son los datos.

El Capítulo 1 describió la arquitectura como el conjunto de decisiones significativas que satisfacen las inquietudes de las partes interesadas mediante compromisos. Este capítulo precisa esas inquietudes. Un requisito que tiene un efecto medible en la estructura de un sistema es un requisito arquitectónicamente significativo, y la mayoría de ellos resultan ser atributos de calidad o restricciones más que funciones. Encontrarlos pronto, y enunciarlos de forma que se puedan probar, es una de las cosas más valiosas que hace un arquitecto.

## Cómo encaja el capítulo

La Sección 2.1 separa los requisitos funcionales, los atributos de calidad y las restricciones, explica por qué muchos arquitectos evitan el término 'requisitos no funcionales' y analiza un sistema de venta de entradas para conciertos el día en que sale a la venta una gran gira. La Sección 2.2 presenta el modelo de calidad del producto de ISO/IEC 25010:2023, con sus nueve características, y muestra cómo usarlo como recordatorio y no como un formulario que rellenar. La Sección 2.3 enseña la técnica más útil del capítulo: el escenario de atributo de calidad de seis partes, que convierte 'debe ser rápido' en un enunciado que se puede probar.

Las Secciones 2.4 a 2.7 recorren después las cualidades que con más frecuencia dan forma a las arquitecturas. La Sección 2.4 trata el rendimiento y la escalabilidad con aritmética real: latencia y rendimiento (throughput), percentiles, latencia de cola, la ley de Little, el encolamiento y la ley de Amdahl. La Sección 2.5 hace lo mismo con la disponibilidad y la fiabilidad: los nueves, el tiempo medio entre fallos y el tiempo medio de reparación, y la disponibilidad de componentes en serie y en paralelo. La Sección 2.6 trata las cualidades del cambio (modificabilidad, capacidad de prueba y desplegabilidad), y la Sección 2.7 reúne la seguridad, la usabilidad y la accesibilidad, la interoperabilidad, el costo, la eficiencia energética, la seguridad (safety) y las cualidades regulatorias.

La Sección 2.8 cierra el círculo. Muestra de dónde salen los requisitos de calidad, cómo un taller de atributos de calidad y un árbol de utilidad convierten un montón de opiniones en una lista ordenada, qué cualidades entran en conflicto con cuáles, y cómo las funciones de aptitud siguen comprobando automáticamente las cualidades importantes una vez construida la arquitectura.

## De los deseos vagos a los números

Este es el capítulo en el que el curso empieza a calcular. Calcularás cuántas solicitudes están en curso a la vez a partir de una tasa de llegada y un tiempo de respuesta, por qué una solicitud a un servidor ocupado al 90% tarda, de media, diez veces su tiempo de servicio, por qué repartir un trabajo que es un 10% secuencial entre dieciséis procesadores solo da una aceleración de 6.4 veces, y cuántos minutos de inactividad al año permite realmente una promesa de 'cuatro nueves'. Nada de las matemáticas es más difícil que la aritmética escolar, pero cambia las conversaciones: 'la base de datos es lenta' se convierte en 'el pool de conexiones está dimensionado para 100 solicitudes simultáneas y la ley de Little dice que necesitamos 400 durante el pico de la noche'.

Los números también desenmascaran la falsa tranquilidad. Un tiempo de respuesta medio de 120 milisegundos puede ocultar que una de cada cincuenta solicitudes tarda más de un segundo, y por eso los profesionales especifican el percentil 95 o 99. Un requisito sin medida de la respuesta no se puede probar, y un requisito que no se puede probar se sacrificará discretamente cuando se acerque una fecha límite.

## Los compromisos son el trabajo

Ningún sistema puede ser a la vez lo más rápido, disponible, seguro, barato y fácil de cambiar posible. La caché mejora la velocidad pero complica la consistencia; el cifrado y las comprobaciones adicionales protegen los datos pero añaden latencia; la redundancia aumenta a la vez la disponibilidad y el costo; los diseños flexibles con plug-ins facilitan el cambio y hacen el rendimiento más difícil de predecir. Por eso el capítulo trata las cualidades como objetivos que compiten entre sí y que hay que ordenar según el contexto, no como casillas que marcar. La habilidad del arquitecto consiste en saber qué dos o tres cualidades importan más para este sistema, enunciarlas con precisión y elegir diseños cuyas debilidades caigan donde el negocio pueda permitírselas.

Las técnicas de diseño que logran cada cualidad, llamadas tácticas, solo se presentan aquí a grandes rasgos. Los capítulos posteriores las desarrollan: los principios de diseño en el Capítulo 3, los estilos arquitectónicos en los Capítulos 5 y 6, las prácticas de datos, seguridad, resiliencia y nube en los Capítulos 7 a 10, y los métodos de evaluación, como ATAM y las funciones de aptitud, en el Capítulo 11.

## La idea que debes llevarte

Si recuerdas una sola cosa de este capítulo, que sea esta: una cualidad no es un requisito hasta que tiene un estímulo, un contexto y una medida. 'El proceso de pago debe ser rápido' es un deseo. 'Cuando 2000 clientes por minuto envían pedidos durante el pico del viernes por la

noche, el 99% de los pagos se completa en 1.5 segundos' es un requisito. Se puede diseñar para él, probar, monitorizar y discutir con pruebas, y le dice a todos, desde el product owner hasta el equipo de operaciones, cómo es el éxito.

SECCIÓN 2.1 DE 2.8 · INTERMEDIO · 23 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

# Requisitos funcionales, atributos de calidad y restricciones

## ANTES DE LEER ESTO

Esta sección se basa en:

- 0.8 Cómo se hace el software
- 1.3 Por qué importa
- 1.5 Partes interesadas e inquietudes

Términos anteriores que utiliza: Parte interesada, Historia de usuario, Compromiso

## LA IDEA EN BREVE

Piensa en encargar una tarta de boda. 'Tres pisos, bizcocho de limón, glaseado blanco' describe qué es la tarta: sus funciones. 'Lista a las 2 de la tarde del sábado, capaz de aguantar un trayecto de 40 minutos, apta para un invitado con alergia a los frutos secos' describe con qué calidad debe funcionar: sus cualidades. 'El lugar del banquete solo admite tartas de pasteleros autorizados' es una restricción: una elección que ya se ha hecho por ti. Un pastelero puede hacer el mismo bizcocho de limón de formas muy distintas, y son las cualidades y las restricciones, no el sabor, las que deciden cómo se hace y se transporta la tarta. Con el software ocurre lo mismo.

## LO QUE NECESITAS SABER

- Los requisitos funcionales indican qué debe hacer un sistema; los atributos de calidad indican con qué calidad debe hacerlo; las restricciones son decisiones ya tomadas, como normativas, tecnologías obligatorias, plazos y presupuestos.
- Muchos arquitectos evitan el término 'requisitos no funcionales' porque define una gran categoría por exclusión, sugiere que los requisitos están separados de las funciones y suena opcional.
- Un requisito arquitectónicamente significativo (ASR) es aquel cuyo cambio obligaría a cambiar la estructura del sistema; la mayoría de los ASR son atributos de calidad o restricciones, pero algunas funciones también son significativas.
- Las mismas funciones pueden ofrecerse con arquitecturas muy distintas; son los atributos de calidad y las restricciones los que deciden entre ellas.

- Los requisitos implícitos, como no perder nunca pedidos o no cobrar nunca dos veces, rara vez se ponen por escrito y hay que obtenerlos activamente de las partes interesadas, los incidentes y las normativas.

## Tres tipos de requisito

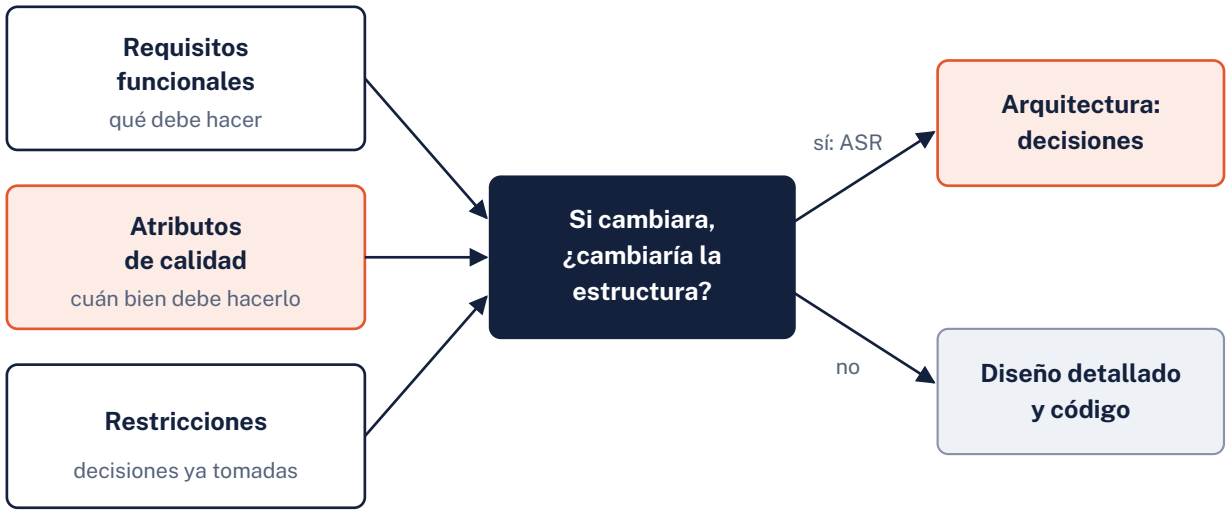
Un requisito es cualquier cosa que un sistema debe cumplir para que sus partes interesadas lo acepten. A los arquitectos les resulta útil clasificar los requisitos en tres tipos, porque cada tipo influye en el diseño de una forma distinta. Un requisito funcional dice qué debe hacer el sistema: 'un cliente puede añadir un artículo al carrito', 'el sistema de farmacia comprueba cada nueva receta frente a las alergias del paciente', 'un pasajero puede ver la posición del conductor en un mapa'. Los requisitos funcionales describen el comportamiento, los datos y las reglas que los relacionan. Son lo que recogen la mayoría de las historias de usuario y los casos de uso.

Un atributo de calidad dice con qué calidad debe hacerlo el sistema. 'Los resultados de búsqueda aparecen en medio segundo', 'el servicio de reservas está disponible el 99.95% del tiempo', 'se puede añadir un nuevo proveedor de pagos en menos de dos semanas', 'los historiales de los pacientes solo pueden leerlos los clínicos que tratan a ese paciente'. Los atributos de calidad son propiedades del sistema en su conjunto o de una función, y solo tienen sentido con una medida asociada: ¿suficientemente rápido para quién, con qué carga, medido cómo?

Una restricción es una decisión que ya se ha tomado por ti. 'El sistema debe ejecutarse en el tenant de Azure que ya tiene la empresa', 'todos los datos personales permanecen en la Unión Europea', 'el equipo sabe Java y no tiene presupuesto para formarse en otra cosa', 'el primer lanzamiento debe estar en producción antes de que empiece el año fiscal el 6 de abril'. Las restricciones eliminan opciones en lugar de describir comportamiento. No son negociables dentro del proyecto, aunque un buen arquitecto comprueba que cada una sea de verdad una restricción y no solo una costumbre o una suposición.

Los límites no siempre son nítidos, y no pasa nada. El objetivo de la clasificación no es el orden burocrático; es asegurarse de que nadie reúna cincuenta historias de usuario y se olvide de preguntar cuántos usuarios habrá, cuánto tiempo puede sobrevivir el negocio a una caída o qué regulador está vigilando.

**Cómo llegan los requisitos a la arquitectura**




Los tres tipos de requisito alimentan la arquitectura, pero solo son arquitectónicamente significativos aquellos cuyo cambio obligaría a un cambio estructural.

## Por qué muchos arquitectos evitan el término 'requisitos no funcionales'

Los documentos de requisitos dividen tradicionalmente todo en requisitos funcionales y requisitos no funcionales (NFR). El término está en todas partes, en contratos, pliegos de licitación y ofertas de empleo, así que necesitas reconocerlo. Aun así, muchos arquitectos e investigadores lo evitan, por tres razones.

Primero, define una categoría grande e importante por lo que no es. Todo lo que no es una función acaba en el mismo cajón: los tiempos de respuesta, las reglas de cifrado, la elección del proveedor de nube y el color del logotipo. El artículo de Martin Glinz *On Non-Functional Requirements* (IEEE International Requirements Engineering Conference, 2007) mostró que las definiciones publicadas no coinciden en qué pertenece a la categoría y que una misma necesidad puede clasificarse de una u otra forma según cómo se formule: 'solo el personal autorizado puede ver los salarios' se lee como una función cuando se escribe como regla de acceso y como una cualidad cuando se escribe como objetivo de seguridad.

Segundo, el nombre sugiere que estos requisitos están separados de las funciones y pueden añadirse más tarde. No es así. Un atributo de calidad califica una función: es el proceso de pago el que debe ser rápido, el pago el que nunca debe perderse, el historial médico el que debe ser confidencial. Por eso Bass, Clements y Kazman, en *Software Architecture in Practice*, hablan de requisitos de atributos de calidad y tratan las restricciones como una categoría aparte.

Tercero, 'no funcional' suena opcional, como si un sistema que funciona pero es lento, inseguro o imposible de cambiar siguiera 'funcionando'. En la práctica, esos son exactamente los sistemas que acaban reescribiéndose. Algunos equipos prefieren 'requisitos de calidad'; otros, en particular en Thoughtworks, dicen 'requisitos transversales' (cross-functional requirements) para subrayar que atraviesan todas las funcionalidades. Este curso usa atributos de calidad y restricciones. Si tu organización dice NFR, no hace falta pelearse por la palabra; basta con asegurarse de que cada NFR sea una cualidad medible o una restricción claramente enunciada.

## Restricciones: técnicas, organizativas, regulatorias, de plazo y de presupuesto

Las restricciones merecen más atención de la que suelen recibir, porque a menudo deciden más de la arquitectura que cualquier funcionalidad. Una restricción no es más que un requisito sin libertad restante. Ayuda agruparlas según su origen.

**Tipos de restricción y su efecto en la arquitectura**

| Tipo | Ejemplo | Efecto típico en la arquitectura |
|---|---|---|
| **Técnica** | Debe integrarse con el sistema financiero SAP existente a través de sus interfaces publicadas; debe ejecutarse en la plataforma Kubernetes de la empresa. | Fija los estilos de integración, los formatos de datos y el modelo de despliegue; puede descartar algunas bases de datos o lenguajes. |
| **Organizativa** | Tres equipos en dos husos horarios; el equipo de operaciones solo da soporte a PostgreSQL; el equipo de seguridad debe aprobar cada dependencia externa. | Determina cómo se divide el sistema (ley de Conway, Capítulo 3) y qué tecnologías son realistas. |
| **Regulatoria y legal** | Los datos personales de residentes en la UE se tratan conforme al RGPD (GDPR); los datos de tarjetas están sujetos a PCI DSS; un dispositivo médico debe cumplir la normativa de seguridad. | Determina la ubicación de los datos, el cifrado, el registro de auditoría, las reglas de conservación y la separación de los componentes sensibles. |
| **Tiempo** | Las nuevas normas fiscales se aplican desde una fecha fija; una plataforma de comercio minorista debe estar lista antes del pico de noviembre. | Favorece la tecnología probada y los diseños más sencillos; puede obligar a un lanzamiento por fases con deuda técnica conocida. |
| **Presupuesto** | Los costos de operación deben mantenerse por debajo de una cantidad fija al mes; no hay dinero para licencias comerciales. | Limita la redundancia, las regiones y los servicios gestionados; convierte el costo por transacción en una cualidad explícita. |

Dos hábitos ayudan. Primero, anota el origen de cada restricción (una normativa, un contrato, una persona concreta con poder de decisión) para que pueda comprobarse y, si cambian las circunstancias, revisarse. Segundo, distingue las restricciones reales de las preferencias disfrazadas de restricciones. 'Debemos usar microservicios' rara vez es una restricción; es una decisión de diseño que alguien tomó pronto, y debe justificarse con atributos de calidad como cualquier otra decisión. La sección del Capítulo 1 sobre decisiones y compromisos explica por qué las decisiones tempranas y difíciles de revertir merecen ese examen.

## Requisitos arquitectónicamente significativos

La mayoría de los requisitos no afectan a la arquitectura. Que la factura muestre el código postal del cliente encima o debajo del nombre de la calle es un detalle; se puede cambiar en una tarde. Un requisito arquitectónicamente significativo (ASR) es aquel que tiene un efecto medible en la arquitectura: si cambiara, la estructura del sistema tendría que cambiar. Reconocer los ASR pronto importa porque es caro acomodarlos tarde. Añadir una sala de espera a un sitio de venta de entradas la semana antes de una venta es posible; particionar el inventario de asientos entre servidores después del lanzamiento es reescribirlo.

Lianping Chen, Muhammad Ali Babar y Bashar Nuseibeh entrevistaron a profesionales sobre cómo reconocen los ASR (*Characterizing Architecturally Significant Requirements*, IEEE Software, 2013). Sus conclusiones coinciden con la experiencia de la mayoría de los arquitectos. Los ASR suelen ser difíciles de satisfacer, a menudo afectan al sistema en su conjunto y no a una sola funcionalidad, con frecuencia entran en conflicto entre sí y a menudo se enuncian de forma vaga o no se enuncian en absoluto. Los atributos de calidad y las restricciones son la mayoría, pero algunos requisitos funcionales también son significativos.

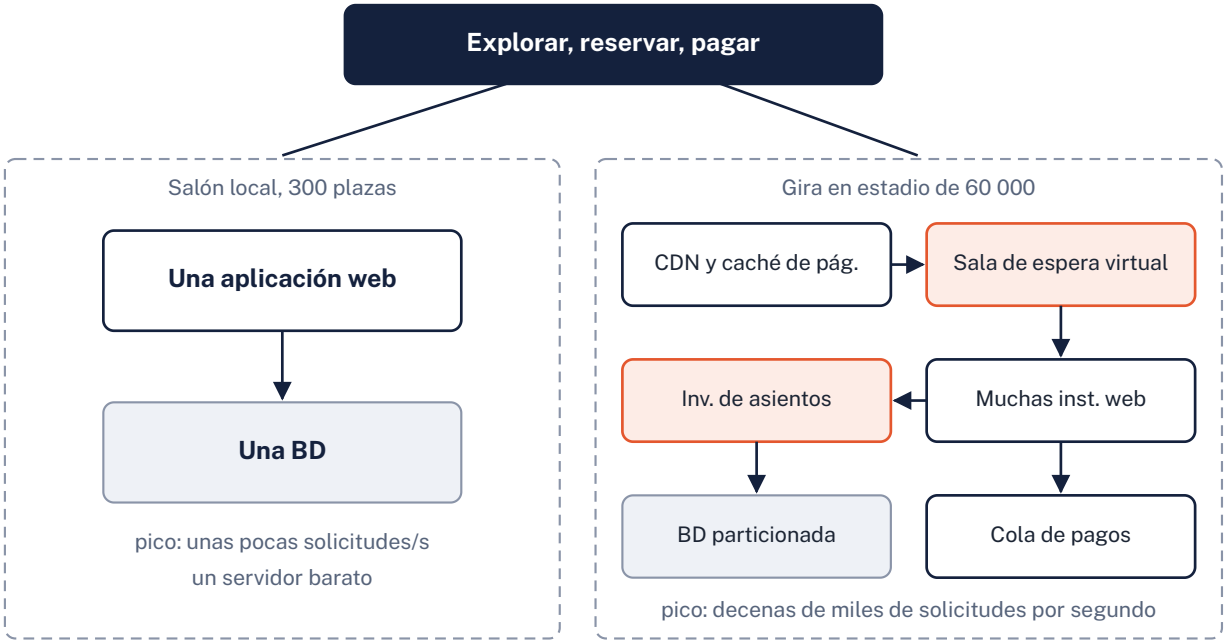
Ejemplos de ASR funcionales: 'todo cambio en el historial de un paciente debe poder atribuirse a una persona y a un momento' (necesita un mecanismo de auditoría en todos los componentes), 'los usuarios pueden colaborar en el mismo documento en tiempo real' (exige un modelo de sincronización y probablemente otro estilo de comunicación) y 'los clientes pueden deshacer una transferencia bancaria en un plazo de 30 minutos' (afecta a cómo se mueve el dinero entre cuentas). Cada uno obliga a una decisión estructural, no solo a escribir más código.

## Por qué las cualidades dan forma a la arquitectura más que las funciones

Esta es la idea central de este capítulo. Si solo importara la funcionalidad, casi cualquier estructura serviría: podrías escribir todo el sistema como un único programa grande y seguiría calculando las respuestas correctas. Bass, Clements y Kazman lo dicen directamente: la funcionalidad no determina la arquitectura, porque muchas arquitecturas distintas pueden ofrecer las mismas funciones. Lo que las separa es lo bien que cada una logra las cualidades.

Piensa en dos sistemas de venta de entradas con funcionalidades idénticas: explorar eventos, reservar asientos, pagar. Uno vende entradas para un salón local de 300 plazas; el otro vende una gira por estadios de 60 000 plazas. El momento de más actividad del salón local puede traer unas pocas solicitudes por segundo, y una única aplicación web con una base de datos es la respuesta correcta: barata, sencilla y fácil de cambiar. La gira por estadios debe absorber una ráfaga miles de veces mayor, no debe vender nunca el mismo asiento dos veces y debe impedir que los bots lo compren todo en el primer segundo. Ese sistema necesita una red de distribución de contenidos, una cola delante de la tienda, un inventario de asientos cuidadosamente diseñado, un almacenamiento particionado y un flujo de pago capaz de esperar a un proveedor sobrecargado. Las mismas funciones, arquitecturas completamente distintas, y la diferencia procede por completo de los atributos de calidad y las restricciones.

**Mismas funciones, cualidades distintas**




Unas funcionalidades idénticas pueden dar lugar a estructuras radicalmente distintas cuando difieren los requisitos de carga, de consistencia y de seguridad.

**SI ESTO ES NUEVO PARA TI**

### Paso a paso: ¿es significativo este requisito?

La idea más difícil de esta sección es saber qué requisitos son arquitectónicamente significativos (ASR). La prueba no es si un requisito se llama función o cualidad, sino si cambiarlo cambiaría la estructura del sistema. Aplica la prueba a tres requisitos.

- La prueba: imagina que el requisito cambia (la cifra se duplica o se reduce, la regla se endurece o se elimina) y pregúntate qué tendría que moverse: componentes, datos o los caminos entre ellos.
- Una factura muestra el código postal del cliente debajo del nombre de la calle. Si se pasa arriba, cambia una plantilla en una tarde. No se mueve nada más, así que esto es diseño detallado, no arquitectura.
- La gira por estadios debe absorber una ráfaga miles de veces mayor que las pocas solicitudes por segundo del salón local. Esa ráfaga explica en gran parte por qué necesita una red de distribución de contenidos, una sala de espera y un almacenamiento particionado. Si se reduce la carga a la del salón local, la mayor parte de esa maquinaria podría desaparecer; al salón le basta con una aplicación web y una base de datos.
- Así que la cifra de carga es un ASR. Lo que la hace significativa es su efecto en la estructura, no su etiqueta de cualidad.
- Las funciones también pueden pasar la prueba. Toma la regla de que cada modificación del historial de un paciente debe mostrar quién la hizo y cuándo. Parece una funcionalidad, pero necesita un mecanismo de auditoría en todos los componentes. Si se elimina, ese mecanismo desaparece, así que es significativa.
- **La lección:** pregúntate '¿qué movería este cambio?', no '¿qué tipo de requisito es?'. La mayoría de los requisitos, de cualquier tipo, quedan dentro de una sola parte. Los pocos que moverían componentes, datos o los caminos entre ellos son aquellos en torno a los cuales hay que construir la arquitectura.

## Cualidades implícitas que nadie pone por escrito

Algunas de las cualidades más importantes nunca se enuncian porque todas las partes interesadas las dan por supuestas. Estos requisitos implícitos solo salen a la luz cuando se incumplen, normalmente en producción. Nadie escribe 'no perder pedidos', 'no cobrar dos veces al cliente', 'seguir funcionando cuando se adelanta la hora', 'mostrar el mismo saldo en el teléfono y en el sitio web' o 'no filtrar los datos de un cliente a otro'. Sin embargo, cada uno de ellos es un ASR con consecuencias importantes: un tratamiento idempotente de los pagos, escrituras duraderas, una programación segura frente a los husos horarios, consistencia entre canales y un aislamiento estricto entre tenants.

Los requisitos implícitos se esconden por razones sociales. Las partes interesadas del negocio creen que son demasiado obvios para mencionarlos; los desarrolladores suponen que otra persona se ha ocupado de ellos; y las plantillas de requisitos rara vez los piden. Algunas preguntas útiles para sacarlos a la luz son: '¿Qué te pondría furioso si ocurriera?', '¿Qué nos haría salir en los periódicos?', '¿Qué ha fallado en el sistema que vamos a sustituir?' y '¿Por qué despiertan de noche al equipo de operaciones?'. Los informes de incidentes y las incidencias de soporte de los sistemas anteriores son una fuente muy rica; también lo son las guías publicadas por los reguladores y las obligaciones de los contratos con clientes.

EJEMPLO PRÁCTICO

## Ejemplo práctico: un sistema de venta de entradas para conciertos el día de la venta

Una empresa de venta de entradas pondrá a la venta cuatro noches seguidas en un estadio (60 000 plazas por noche) a las 10:00 de un viernes. En un día normal, el sitio recibe unos 2000 visitantes por hora. Marketing espera que lleguen 1.5 millones de personas en los diez primeros minutos. Las funcionalidades son las habituales: explorar, elegir asientos, reservarlos durante diez minutos, pagar. ¿Cuáles son los requisitos arquitectónicamente significativos?

| | |
|---|---|
| **Entradas a la venta** | 4 noches × 60 000 plazas = 240 000 entradas |
| **Tasa de llegada normal** | 2000 visitantes ÷ 3600 s ≈ 0.56 visitantes por segundo |
| **Tasa de llegada el día de la venta** | 1 500 000 visitantes ÷ 600 s = 2500 visitantes por segundo, unas 4500 veces la tasa normal |
| **Tasa de solicitudes** | Si el navegador de cada visitante hace unas 8 solicitudes al sitio en el primer minuto (páginas, mapas de asientos, llamadas a la API): 2500 × 8 = 20 000 solicitudes por segundo |
| **Demanda frente a oferta** | 1 500 000 visitantes para 240 000 entradas: la mayoría de la gente se llevará una decepción sea cual sea el diseño, así que la equidad y la comunicación clara importan tanto como la velocidad |
| **Restricción de pago** | El contrato del proveedor de pagos permite 300 autorizaciones de pago por segundo. Con una media de 2.5 entradas por pedido, 240 000 ÷ 2.5 = 96 000 pedidos; 96 000 ÷ 300 = 320 s, así que solo los pagos tardan al menos 5.3 minutos aunque todo lo demás sea instantáneo |
| **ASR 1: carga pico (rendimiento)** | Absorber 2500 llegadas por segundo sin colapsar; servir las páginas estáticas y los mapas de asientos desde una caché o una CDN para que solo las compras reales lleguen al núcleo |
| **ASR 2: ningún asiento vendido dos veces (consistencia)** | Las reservas de asientos deben ser atómicas incluso cuando miles de personas hacen clic en el mismo bloque a la vez; esto determina el diseño del inventario y su almacenamiento |
| **ASR 3: equidad (regla de negocio con impacto estructural)** | Admitir a los compradores por orden de llegada mediante una sala de espera virtual, para que el sistema central reciba un flujo controlado y no una estampida |
| **ASR 4: resistencia a los bots (seguridad)** | Detectar y bloquear a los compradores automatizados antes de que lleguen al inventario; aplicar limitación de tasa (rate limiting) por cuenta y por dispositivo |

| | |
|---|---|
| **ASR 5: degradación elegante (disponibilidad)** | Si los pagos se ralentizan, mantener válidas las reservas y poner en cola los intentos de pago en lugar de hacer fallar pedidos completos |
| **Restricción** | Fecha y hora fijas anunciadas de antemano; límite del proveedor de pagos de 300 por segundo |

Ninguno de estos ASR aparece en la lista de funcionalidades. Todos proceden de la carga, la consistencia, la equidad, la seguridad y una restricción de un tercero.

El límite de pagos se traduce directamente en una decisión de diseño: como los compradores no pueden pagar todos a la vez, el sistema debe reservar los asientos y poner en cola los pagos, lo que a su vez fija el tiempo de reserva.

El mundo real ofrece un recordatorio. En noviembre de 2022, la preventa de Ticketmaster para una gran gira por estadios sufrió largos retrasos y fallos por la demanda; la empresa informó de 3.5 mil millones de solicitudes en total al sistema, cuatro veces su pico anterior.

## Lista de comprobación: detectar requisitos arquitectónicamente significativos

Repasa estas preguntas para cada requisito candidato, y para el sistema en su conjunto, al principio de un proyecto.

- [ ] **¿Cambiaría la estructura si cambiara?**
  Si duplicar la cifra, relajar la regla o eliminar el requisito movería componentes, datos o caminos de comunicación, es significativo.
- [ ] **¿Se aplica a muchas funcionalidades?**
  Los requisitos que atraviesan todo el sistema (auditoría, seguridad, disponibilidad, internacionalización) suelen necesitar una respuesta estructural.
- [ ] **¿Es difícil o nuevo?**
  Una cifra exigente (menos de 100 ms, 99.99%), una carga muy superior a cualquiera que haya manejado el equipo o una funcionalidad sin precedentes son una señal fuerte.
- [ ] **¿Tiene un gran valor o riesgo para el negocio?**
  Si no cumplirlo supondría perder ingresos, infringir la ley o causar daño a alguien, trátalo como significativo aunque parezca sencillo.
- [ ] **¿Entra en conflicto con otra cualidad?**
  Los requisitos en tensión (rápido y fuertemente consistente, abierto y seguro) necesitan compromisos deliberados, que son decisiones arquitectónicas.
- [ ] **¿Es una restricción disfrazada?**
  Las tecnologías fijadas, las normativas, los plazos y los presupuestos eliminan opciones; regístralos junto con su origen.
- [ ] **¿Ha causado problemas antes?**
  Revisa los incidentes, las incidencias de soporte y los post-mortems del sistema que se sustituye, y de sistemas similares de otros lugares.
- [ ] **¿Alguien lo está dando por supuesto en silencio?**
  Pregunta qué pondría furiosas a las partes interesadas o haría salir a la empresa en las noticias; esas respuestas revelan los requisitos implícitos.

**ATENCIÓN**

### Errores comunes

- Reunir cientos de historias de usuario y una línea que dice 'el sistema debe ser rápido, seguro y escalable'.
- Tratar todos los requisitos como significativos. Si todo es un ASR, no se prioriza nada; la mayoría de los requisitos pertenecen al diseño detallado.
- Aceptar una decisión de diseño ('usar Kafka', 'usar microservicios') como si fuera una restricción, sin un atributo de calidad que la justifique.
- Escribir cualidades sin medidas. 'Altamente disponible' no se puede diseñar ni probar; la Sección 2.3 muestra cómo corregirlo.
- Suponer que la calidad se puede añadir después de construir las funcionalidades. El rendimiento, la seguridad y la disponibilidad dependen de la estructura y es caro incorporarlos a posteriori.

## Cómo trabajan los profesionales con los ASR

Los arquitectos con experiencia mantienen una lista breve y viva de ASR, normalmente de cinco a quince elementos, en lugar de un largo documento de requisitos. Cada elemento enuncia la cualidad como un escenario comprobable (Sección 2.3), nombra a la parte interesada a la que le importa, registra su prioridad y enlaza con las decisiones que motivó. La plantilla arc42, por ejemplo, pide los tres a cinco objetivos de calidad principales en su primera página y una lista más completa de requisitos de calidad más adelante. La lista se revisa cada vez que se toma una decisión importante o el negocio cambia de rumbo, y los elementos más importantes acaban respaldados por comprobaciones automatizadas llamadas funciones de aptitud, que se presentan en la Sección 2.8.

**COMPRUEBA LO QUE HAS ENTENDIDO**

▼ **¿'Los pacientes pueden reservar citas en línea' es un requisito funcional, un atributo de calidad o una restricción?**

Un requisito funcional: dice qué debe hacer el sistema. Una cualidad añadiría con qué calidad, por ejemplo con qué rapidez se confirman las reservas o cuántas se pueden atender a las 8 de la mañana de un lunes.

▼ **¿Por qué 'el sistema debe ser escalable' todavía no es un requisito útil?**

No tiene estímulo, contexto ni medida. No dice cuánta carga, con qué rapidez crece la carga ni qué respuesta debe mantenerse, así que nadie puede diseñar para él ni probarlo.

▼ **Da un ejemplo de requisito funcional que sea arquitectónicamente significativo.**

Por ejemplo, una pista de auditoría completa de cada cambio en los historiales médicos, o la edición colaborativa en tiempo real. Cada uno necesita un mecanismo estructural en todo el sistema, no solo más código en un lugar.

▼ **¿Qué pregunta separa mejor un requisito significativo de uno corriente?**

¿Tendría que cambiar la estructura del sistema si cambiara este requisito? Si la respuesta es sí, es arquitectónicamente significativo.

▼ **Inténtalo: Un enunciado de venta de entradas dice: los fans pueden elegir asientos; confirmación en 2 segundos en el pico del día de la venta; el alojamiento debe seguir con el proveedor de nube actual; a nadie se le cobra dos veces. Clasifica cada punto y di cuáles son arquitectónicamente significativos.**

Elegir asientos es funcional; importa, pero apenas decide la estructura. La confirmación en menos de 2 segundos en el pico es un atributo de calidad y claramente significativo, ya que determina las colas y la capacidad. La regla de alojamiento es una restricción, una decisión ya tomada, y es significativa porque limita las opciones. Que a nadie se le cobre dos veces es un atributo de calidad implícito, normalmente no escrito, y muy significativo: da forma a la idempotencia y a cómo se registran los pagos.

**FUENTES**


- Martin Glinz, On Non-Functional Requirements (RE 2007): https://files.ifi.uzh.ch/rerg/amadeus/publications/papers/RE07_Glinz.pdf
- Chen, Ali Babar and Nuseibeh, Characterizing Architecturally Significant Requirements (IEEE Software, 2013): https://pure.itu.dk/en/publications/characterizing-architecturally-significant-requirements/
- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- arc42: Introduction and goals (quality goals): https://docs.arc42.org/section-1/

SECCIÓN 2.2 DE 2.8 · INTERMEDIO · 21 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

# Los modelos de calidad: ISO/IEC 25010 y cómo usarla

## ANTES DE LEER ESTO

Esta sección se basa en:

- 1.5 Partes interesadas e inquietudes
- 2.1 Requisitos funcionales y de calidad

Términos anteriores que utiliza: Atributo de calidad, Parte interesada, Requisito

## LA IDEA EN BREVE

Cuando compras un vehículo de segunda mano, una buena lista de comprobación evita que te olvides de cosas: motor, frenos, neumáticos, óxido, documentación, consumo de combustible. No necesitas que todo sea perfecto, pero quieres haber revisado cada punto y decidido si te importa. Un modelo de calidad es esa lista de comprobación para el software. La norma internacional ISO/IEC 25010 enumera nueve áreas, como el rendimiento, la fiabilidad, la seguridad y la facilidad para cambiar el software, para que no se pase por alto nada importante. La habilidad está en usarla para plantear buenas preguntas, no para rellenar formularios.

## LO QUE NECESITAS SABER

- Un modelo de calidad ofrece un vocabulario compartido de características y subcaracterísticas de calidad y funciona como lista de comprobación para encontrar las cualidades que importan.
- ISO/IEC 25010:2023 define nueve características del producto: adecuación funcional, eficiencia de rendimiento, compatibilidad, capacidad de interacción, fiabilidad, seguridad, mantenibilidad, flexibilidad y seguridad (safety).
- En comparación con 2011, la usabilidad pasó a ser capacidad de interacción, la portabilidad pasó a ser flexibilidad, se añadió la seguridad (safety) y la calidad en uso pasó a una norma separada, ISO/IEC 25019:2023.
- Otras listas, como FURPS+, las cualidades de Software Architecture in Practice y Q42, se solapan en gran medida con ISO/IEC 25010; ninguna es la única correcta.
- Usa un modelo para provocar una decisión rápida y explícita sobre cada característica, y convierte solo las pertinentes en unos pocos escenarios medibles y priorizados.

## Para qué sirve un modelo de calidad

La Sección 2.1 argumentó que los atributos de calidad deciden la mayor parte de una arquitectura. La siguiente pregunta obvia es: ¿qué cualidades? Si se las deja a su aire, las partes interesadas mencionan las dos o tres que les hicieron daño la última vez, normalmente la velocidad y el tiempo de actividad, y se olvidan del resto hasta que es demasiado tarde. Un modelo de calidad es una lista estructurada de características de calidad, cada una dividida en subcaracterísticas más concretas, con nombres y definiciones acordados. Cumple dos funciones. Da a las personas un vocabulario compartido, para que 'fiabilidad' signifique lo mismo para el product owner, el tester y el proveedor. Y actúa como lista de comprobación, que te lleva a considerar cualidades que nadie ha planteado todavía.

Un modelo no te dice qué cualidades importan para tu sistema ni cuánto necesitas de cada una. Eso sale de las partes interesadas y de los objetivos de negocio (Sección 2.8). Tampoco te dice cómo lograrlas; esa es la tarea de las tácticas, los patrones y el resto de este curso. Es un mapa del territorio, no una ruta a través de él.

## ISO/IEC 25010:2023: el modelo de calidad del producto

El modelo más citado es ISO/IEC 25010, que forma parte de la familia de normas SQuaRE (Systems and software Quality Requirements and Evaluation). La segunda edición, publicada en noviembre de 2023, define un modelo de calidad del producto con nueve características. Se aplica a los productos de software y a los productos TIC en general, y la norma enumera sus usos: especificar requisitos, comprobar que los requisitos están completos, fijar objetivos de diseño y de prueba, definir criterios de aceptación y establecer medidas de calidad. Otras normas de la familia definen cómo medir cada subcaracterística (ISO/IEC 25023) y cómo evaluar un producto.

Las nueve características son adecuación funcional, eficiencia de rendimiento, compatibilidad, capacidad de interacción, fiabilidad, seguridad, mantenibilidad, flexibilidad y seguridad (safety). La primera describe si el sistema hace lo correcto y lo hace bien; las otras ocho describen con qué calidad lo hace y lo bien que puede operarse, protegerse y cambiarse. El diagrama siguiente muestra la estructura; la tabla posterior enumera todas las subcaracterísticas con una pregunta de ejemplo que podría hacerse un arquitecto.

### Las nueve características




ISO/IEC 25010:2023 agrupa la calidad del producto en nueve características; respecto a la edición de 2011, dos se renombraron y se añadió una.

**Características, subcaracterísticas y una pregunta de ejemplo para una app de banca móvil**

| Característica | Subcaracterísticas | Pregunta de ejemplo |
|---|---|---|
| **Adecuación funcional** | Completitud funcional, corrección funcional, pertinencia funcional | ¿Calcula la app correctamente la cuota del préstamo, y hace todo lo que el cliente necesita para completar la tarea? |
| **Eficiencia de rendimiento** | Comportamiento temporal, utilización de recursos, capacidad | ¿Con qué rapidez aparece el saldo en un teléfono de gama media con 4G, y a cuántos usuarios simultáneos puede atender el backend? |
| **Compatibilidad** | Coexistencia, interoperabilidad | ¿Puede la app intercambiar datos de pago con el sistema nacional de pagos inmediatos, y funcionar junto a otras apps sin agotar los recursos compartidos? |
| **Capacidad de interacción (antes usabilidad)** | Reconocibilidad de la adecuación, capacidad de aprendizaje, operabilidad, protección contra errores del usuario, implicación del usuario, inclusividad, asistencia al usuario, autodescriptividad | ¿Puede un usuario nuevo configurar una orden de pago periódica sin ayuda, y puede hacer lo mismo un usuario de lector de pantalla? |
| **Fiabilidad** | Ausencia de defectos, disponibilidad, tolerancia a fallos, capacidad de recuperación | Si el socio de procesamiento de tarjetas está caído, ¿pueden los clientes seguir viendo sus saldos y mover dinero entre sus propias cuentas? |
| **Seguridad** | Confidencialidad, integridad, no repudio, responsabilidad, autenticidad, resistencia | ¿Podemos demostrar qué cliente autorizó una transferencia, y resiste la app los ataques de credential stuffing? |
| **Mantenibilidad** | Modularidad, reutilizabilidad, analizabilidad, modificabilidad, capacidad de prueba | ¿Cuánto se tarda en añadir un nuevo tipo de cuenta, y se pueden probar las reglas de transferencia sin un core bancario real? |
| **Flexibilidad (antes portabilidad)** | Adaptabilidad, escalabilidad, instalabilidad, reemplazabilidad | ¿Puede el backend crecer de 1 a 5 millones de clientes, y podríamos sustituir al proveedor de notificaciones push en un trimestre? |

| Característica | Subcaracterísticas | Pregunta de ejemplo |
|---|---|---|
| **Seguridad (safety)** | Restricción operativa, identificación de riesgos, seguridad ante fallos (fail safe), advertencia de peligros, integración segura | Pertinente sobre todo para sistemas que pueden causar daños a personas, bienes o el medio ambiente: una bomba de infusión, un controlador de trenes, un robot industrial. |

Conviene conocer con precisión algunas definiciones, porque difieren del uso cotidiano. En ISO/IEC 25010, la **disponibilidad** es una subcaracterística de la fiabilidad: el grado en que un sistema está operativo y accesible cuando se necesita. La **ausencia de defectos** es el grado en que un sistema realiza sus funciones sin defectos en funcionamiento normal. La **capacidad**, dentro de la eficiencia de rendimiento, es el límite máximo de un parámetro, como los usuarios simultáneos o las transacciones por segundo, que todavía cumple los requisitos. La **escalabilidad** se sitúa dentro de la flexibilidad, lo que refleja una visión del escalado como la capacidad de adaptarse a una carga cambiante más que como una cuestión de velocidad bruta. La **resistencia**, nueva dentro de la seguridad, es la capacidad de mantener las operaciones mientras se sufre un ataque. Las Secciones 2.4 a 2.7 vuelven a estas cualidades en profundidad; la Sección 2.5 muestra cómo calcular la disponibilidad.

## Qué cambió respecto a la edición de 2011

La primera edición, ISO/IEC 25010:2011, sustituyó al antiguo modelo ISO/IEC 9126 y fue la referencia estándar durante una década. Muchos libros de texto, pliegos de licitación y plantillas todavía usan sus ocho características, así que merece la pena saber qué cambió.

**ISO/IEC 25010: 2011 comparada con 2023**

| Tema | Edición de 2011 | Edición de 2023 |
|---|---|---|
| **Número de características** | Ocho | Nueve |
| **Usabilidad** | Usabilidad, incluidas la estética de la interfaz de usuario y la accesibilidad | Renombrada capacidad de interacción; la estética se sustituye por la implicación del usuario; la accesibilidad se divide en inclusividad y asistencia al usuario; se añade la autodescriptividad |
| **Portabilidad** | Portabilidad: adaptabilidad, instalabilidad, reemplazabilidad | Renombrada flexibilidad; se añade la escalabilidad como subcaracterística |
| **Fiabilidad** | Madurez, disponibilidad, tolerancia a fallos, capacidad de recuperación | La madurez se sustituye por la ausencia de defectos |
| **Seguridad** | Confidencialidad, integridad, no repudio, responsabilidad, autenticidad | Se añade la resistencia (soportar ataques) |
| **Seguridad (safety)** | Tratada solo dentro de la calidad en uso (ausencia de riesgos) | Nueva característica del producto con cinco subcaracterísticas |
| **Calidad en uso** | Parte de ISO/IEC 25010 (efectividad, eficiencia, satisfacción, ausencia de riesgos, cobertura del contexto) | Pasa a una norma separada, ISO/IEC 25019:2023 (beneficio, ausencia de riesgos, aceptabilidad) |

Los cambios de nombre son más que cosméticos. 'Capacidad de interacción' amplía el enfoque de la facilidad de uso a todo el intercambio de información entre los usuarios y el producto en distintos contextos de uso, y la nueva subcaracterística de inclusividad sitúa la accesibilidad para personas de distintas edades y capacidades en el centro y no en los márgenes. 'Flexibilidad' agrupa las cualidades que permiten a un producto hacer frente a los cambios en su entorno: nuevas plataformas, nuevos destinos de instalación y más o menos carga. Añadir la seguridad (safety) como característica del producto refleja cuánto controla hoy el software cosas físicas, desde automóviles hasta dispositivos médicos.

El segundo cambio estructural es que la calidad en uso, la calidad que experimentan las personas que usan el producto en un contexto real, tiene ahora su propia norma, ISO/IEC 25019:2023. Sus tres características son el beneficio (usabilidad, accesibilidad e idoneidad en

el uso), la ausencia de riesgos (riesgo económico, para la salud, para la vida humana, y riesgo ambiental y social) y la aceptabilidad (experiencia, confiabilidad y cumplimiento). La separación aclara una distinción útil para los arquitectos: la calidad del producto es lo que construyes y puedes probar en el producto; la calidad en uso es el resultado para personas reales, que también depende de la formación, el contexto y cómo se despliega el producto.

**ATENCIÓN**

### Atención a la edición

Cuando un contrato o la plantilla de un cliente dice 'ISO 25010', comprueba a qué edición se refiere. Un requisito que enumera 'usabilidad y portabilidad' casi con seguridad se basa en el modelo de 2011. Eso no es un problema en sí mismo, pero mezclar las dos listas en un documento provoca lagunas (la seguridad (safety) y la resistencia se pasan por alto con facilidad) y duplicados (la accesibilidad aparece dos veces). Elige una edición y dilo.

**SI ESTO ES NUEVO PARA TI**

### Paso a paso: lo que construyes y lo que obtienen las personas

La idea que requiere más atención aquí es la diferencia entre la calidad del producto y la calidad en uso. La calidad del producto es una propiedad del software, que el equipo puede construir y probar. La calidad en uso es lo que las personas reales obtienen realmente de él, que también depende de quiénes son y de la situación en que se encuentran. La app de banca móvil muestra la diferencia.

- Dentro de la protección contra errores del usuario, que forma parte de la capacidad de interacción, el equipo del banco escribe un escenario: la app avisa al cliente antes de un pago a un beneficiario cuyo nombre no coincide con la cuenta.
- Eso es calidad del producto. Los testers envían pagos con nombres que coinciden y que no coinciden y comprueban que el aviso aparece siempre que debe.
- Supón que supera todas las pruebas. El banco todavía quiere saber otra cosa: ¿están perdiendo realmente los clientes menos dinero por pagos erróneos o fraudulentos? Eso es la ausencia de riesgos, que forma parte de la calidad en uso, la cual desde 2023 tiene su propia norma, ISO/IEC 25019.
- La respuesta depende de algo más que la app. Un cliente con prisa puede pasar de largo el aviso con un toque; un estafador al teléfono puede convencerlo de que lo ignore; un cliente que ha aprendido sobre este tipo de estafas puede detenerse y comprobarlo. El software es el mismo cada vez; el resultado no.
- Así que las dos se comprueban de forma distinta: la calidad del producto con pruebas sobre el producto, la calidad en uso observando los resultados reales, por ejemplo cuántos clientes informan de que han pagado a la persona equivocada.
- **La lección:** el equipo puede diseñar y probar directamente la calidad del producto; la calidad en uso solo se ve en el uso real. Una buena calidad del producto hace más probable una buena calidad en uso, pero no puede garantizarla por sí sola.

## Otras listas que encontrarás

ISO/IEC 25010 no es la única taxonomía, y los profesionales usan a menudo listas más sencillas o divididas de otra forma. FURPS+ surgió de FURPS, una clasificación desarrollada en Hewlett-Packard y expuesta por Robert Grady y Deborah Caswell en *Software Metrics: Establishing a Company-Wide Program* (1987). FURPS agrupa los requisitos en funcionalidad, usabilidad, fiabilidad, rendimiento y soportabilidad; la extensión '+' posterior añade restricciones: de diseño, de implementación, de interfaz y físicas y, en algunas versiones, reglas de negocio.

FURPS+ se popularizó a través del Rational Unified Process y sus sucesores. Su punto fuerte es la brevedad; su punto débil es que la 'soportabilidad' mete en el mismo saco la mantenibilidad, la capacidad de prueba, la configurabilidad, la instalabilidad y la localización.

Bass, Clements y Kazman, en la cuarta edición de *Software Architecture in Practice* (2021), no intentan ser exhaustivos. Dedican un capítulo a cada una de las cualidades que con más frecuencia guían las arquitecturas: disponibilidad, desplegabilidad, eficiencia energética, integrabilidad, modificabilidad, rendimiento, seguridad (safety), seguridad, capacidad de prueba y usabilidad, seguidos de un capítulo sobre cómo trabajar con otras cualidades. Para cada una, ofrecen un escenario general y un catálogo de tácticas. Su lista está organizada en torno al diseño y no a la evaluación, lo que la hace especialmente útil para los arquitectos.

Mark Richards y Neal Ford, en *Fundamentals of Software Architecture*, usan el término características de la arquitectura y las ordenan en grupos como el operativo (disponibilidad, rendimiento, escalabilidad), el estructural (modificabilidad, extensibilidad, portabilidad) y el transversal (seguridad, accesibilidad, aspectos legales). El modelo abierto Q42 de Gernot Starke, mantenido junto a la plantilla arc42, adopta otro enfoque más: etiqueta mucho más de un centenar de cualidades con nombre con una lista breve de dimensiones (como eficiente, flexible, operable, fiable, seguro ante daños, seguro ante ataques, adecuado y usable) y ofrece requisitos de ejemplo para cada una.

Estas listas discrepan en los detalles, pero se solapan en gran medida. La propia discrepancia es instructiva: no existe una única descomposición correcta de la 'calidad', solo descomposiciones más o menos útiles para un fin. Usar un modelo estándar ayuda cuando necesitas comunicarte con compras, auditores u otras organizaciones; una lista más ligera puede servirle mejor a un equipo de producto en el día a día.

## Usar un modelo como lista de comprobación sin crear burocracia

El gran riesgo de un modelo de calidad es que se convierta en un formulario. Una plantilla de requisitos con un apartado para cada una de las 40 subcaracterísticas invita a rellenar todas las casillas, y el resultado es un documento lleno de afirmaciones como 'el sistema será mantenible' que nadie lee y nadie prueba. El esfuerzo se reparte por igual entre cualidades que importan de forma muy desigual, y las dos o tres cualidades que de verdad decidirán la arquitectura se pierden entre las demás.

Un enfoque mejor trata el modelo como punto de partida para una conversación. Recorre las características con las partes interesadas una vez, rápidamente, preguntando para cada una: ¿importa esto para nuestro sistema y, si es así, cómo sería un fallo? Descarta explícitamente las que no sean pertinentes (escribir 'seguridad (safety): no aplicable, la app no puede causar

daños físicos' es útil, porque registra una decisión). Convierte las pertinentes en un pequeño número de escenarios concretos y medibles (Sección 2.3). Ordénalos (Sección 2.8). Después guarda el modelo hasta la próxima revisión.

**Una forma ligera de aplicar el modelo**

1. **Recorre las nueve características**
   En una sesión de una hora, lee en voz alta cada característica y pregunta a las partes interesadas si importa. Anota a quién le importa y por qué.

2. **Marca la pertinencia**
   Califica cada característica como alta, media, baja o no aplicable, y registra una frase de justificación para cada decisión.

3. **Profundiza en las altas**
   Solo para las características de pertinencia alta, revisa las subcaracterísticas para encontrar la inquietud precisa, como la capacidad de recuperación en lugar de la fiabilidad en general.

4. **Escribe escenarios**
   Expresa cada inquietud precisa como uno o dos escenarios medibles con un estímulo, un contexto y una medida de la respuesta.

5. **Prioriza y enlaza**
   Ordena los escenarios y enlaza cada uno con las decisiones de diseño que guía, para que la lista siga siendo corta y viva.

EJEMPLO PRÁCTICO

## Ejemplo práctico: aplicar ISO/IEC 25010 a una app de banca móvil

Un banco mediano va a sustituir su app móvil y su backend. Tiene 1.2 millones de usuarios activos de la app; la tarde del día de cobro, alrededor del 25% de ellos inicia sesión entre las 18:00 y las 20:00. El equipo recorre las nueve características en una hora y registra el resultado.

| | |
|---|---|
| **Estimación de la carga pico** | 1 200 000 × 25% = 300 000 inicios de sesión en 2 horas = 300 000 ÷ 7200 s ≈ 42 inicios de sesión por segundo de media; con un margen de 3× para las ráfagas ≈ 125 por segundo |
| **Adecuación funcional: alta** | Los intereses, las comisiones y los saldos deben ser exactos hasta el último céntimo. Escenario: todo saldo mostrado concilia con el libro mayor central en un plazo de 5 segundos desde que se registra una transacción. |
| **Eficiencia de rendimiento: alta** | Escenario: con 125 inicios de sesión por segundo, la pantalla de inicio con los saldos se muestra en 2 segundos para el 95% de los usuarios con una conexión 4G. El consumo de batería y de datos en el teléfono también importa (utilización de recursos). |
| **Compatibilidad: media** | Debe interoperar con el sistema nacional de pagos inmediatos y con las API de banca abierta que exige la normativa; debe funcionar en las dos últimas versiones principales de los dos sistemas operativos móviles. |
| **Capacidad de interacción: alta** | Lo que más importa es la inclusividad y la protección contra errores del usuario. Escenario: un usuario de lector de pantalla puede completar un pago a un nuevo beneficiario sin ayuda de una persona que vea; la app avisa antes de un pago a un nombre de beneficiario que no coincide con la cuenta. |
| **Fiabilidad: alta** | Capacidad de recuperación y tolerancia a fallos. Escenario: si falla la plataforma de tarjetas, los saldos y las transferencias entre cuentas propias siguen funcionando; el servicio completo se restablece en 30 minutos. |
| **Seguridad: alta** | Autenticidad, no repudio y resistencia. Escenario: cada pago se autoriza con autenticación reforzada del cliente y se registra de forma que el banco pueda demostrar quién lo aprobó; 10 000 intentos de credential stuffing por minuto se frenan sin bloquear a los usuarios legítimos. |
| **Mantenibilidad: media** | Modificabilidad y capacidad de prueba. Escenario: se puede lanzar un nuevo producto de ahorro en menos de 6 semanas sin cambiar el componente de pagos. |
| **Flexibilidad: media** | Escalabilidad. Escenario: el backend puede crecer hasta 3 millones de usuarios añadiendo instancias, sin rediseño. |

| | |
|---|---|
| **Seguridad (safety): no aplicable** | La app no puede causar daños físicos; los riesgos para el dinero se tratan en seguridad y fiabilidad. Se registra explícitamente para que la decisión sea visible. |
| **Resultado** | Cinco características calificadas como altas, tres como medias y una como no aplicable. Once escenarios en total, cada uno con un número. Los tres principales (seguridad, fiabilidad y rendimiento en el pico del día de cobro) se convierten en los objetivos de calidad de la primera página del documento de arquitectura. |

La aritmética convierte 'muchos usuarios el día de cobro' en una cifra de diseño (unos 125 inicios de sesión por segundo en ráfaga) que se puede probar.

Las subcaracterísticas cumplieron su función: 'seguridad' se convirtió en autenticidad, no repudio y resistencia, y cada una sugiere mecanismos distintos.

Marcar la seguridad (safety) como no aplicable es una decisión, no una omisión. Si más adelante el banco añade una funcionalidad que pudiera causar daños, el registro muestra que hay que revisarla.

## Preguntas frecuentes

**¿Tengo que citar ISO/IEC 25010 para hacer bien la arquitectura?**
No. Es valiosa como vocabulario compartido y como lista de comprobación, e imprescindible cuando los contratos o los reguladores hacen referencia a ella. Muchos equipos excelentes usan una lista más ligera. Lo que importa es que las cualidades importantes se identifiquen, se hagan medibles y se prioricen.

**¿Dónde encaja la escalabilidad: en el rendimiento o en la flexibilidad?**
En ISO/IEC 25010:2023 es una subcaracterística de la flexibilidad, mientras que la capacidad se sitúa dentro de la eficiencia de rendimiento. En la práctica, las dos están muy relacionadas, y la Sección 2.4 las trata juntas. No dejes que las discusiones de clasificación retrasen la redacción del escenario.

**¿Está la 'observabilidad' en el modelo?**
No con ese nombre. Algunas partes corresponden a la analizabilidad (mantenibilidad) y a la responsabilidad (seguridad). Muchos profesionales la tratan hoy como una cualidad por derecho propio; la Sección 2.7 la analiza.

**COMPRUEBA LO QUE HAS ENTENDIDO**

**▼ ¿Cómo pasaron a llamarse la usabilidad y la portabilidad en ISO/IEC 25010:2023?**

La usabilidad pasó a ser capacidad de interacción y la portabilidad pasó a ser flexibilidad.

**▼ ¿Dentro de qué característica sitúa ISO/IEC 25010:2023 la disponibilidad?**

Dentro de la fiabilidad, junto con la ausencia de defectos, la tolerancia a fallos y la capacidad de recuperación.

**▼ ¿Por qué es útil escribir 'no aplicable' junto a una característica en lugar de dejarla en blanco?**

Registra una decisión deliberada con su justificación, de modo que los revisores pueden ver que se consideró y la decisión puede revisarse si el sistema cambia.

**▼ ¿Cuál es el principal peligro de convertir un modelo de calidad en una plantilla de requisitos?**

La gente rellena todas las casillas con afirmaciones vagas, el esfuerzo se reparte por igual entre cualidades de importancia muy desigual y las pocas cualidades que de verdad deciden la arquitectura se pierden.

**▼ Inténtalo: Al recorrer ISO/IEC 25010 para una app de repartidores, un colega marca las nueve características como altas "por si acaso". ¿Qué tiene eso de malo? Nombra dos que marcarías como altas y una como baja, y da tus razones.**

Si todo es alto, nada guía el diseño, y el modelo se ha convertido en burocracia. La fiabilidad es alta: los repartidores con mala señal no deben perder trabajos, así que atiende a la recuperabilidad. La capacidad de interacción es alta: los repartidores necesitan pasos cortos y sencillos, y la seguridad (safety) merece atención por la misma razón. La compatibilidad podría ser menor si la app habla solo con los servicios propios de la empresa. Anota una frase de razonamiento para cada una y luego escribe escenarios solo para las altas.

**FUENTES**


- ISO/IEC 25010:2023 Product quality model (ISO catalogue page): https://www.iso.org/standard/78176.html
- ISO/IEC 25019:2023 Quality-in-use model (ISO catalogue page): https://www.iso.org/standard/78177.html
- Q42: ISO 25010 update 2023 (summary of changes): https://quality.arc42.org/articles/iso-25010-update-2023

- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/

# Escribir escenarios de atributos de calidad que se puedan probar

### ANTES DE LEER ESTO

Esta sección se basa en:

- 2.1 Requisitos funcionales y de calidad
- 2.2 Modelos de calidad

Términos anteriores que utiliza: Atributo de calidad, Parte interesada, Producción

### LA IDEA EN BREVE

Decirle a un constructor 'haz el tejado resistente' no le da nada que comprobar. Decirle 'el tejado debe soportar 50 centímetros de nieve húmeda durante una tormenta invernal sin filtraciones ni hundirse más de 2 centímetros' sí. Nombra lo que ocurre (la nieve), las condiciones (una tormenta), la parte implicada (el tejado), lo que debe ocurrir (sin filtraciones) y un número que se puede medir (2 centímetros). Un escenario de atributo de calidad hace lo mismo con el software. Convierte un deseo como 'el sitio web debe ser rápido' en una afirmación lo bastante precisa para diseñar en función de ella y probarla.

### LO QUE NECESITAS SABER

- Un escenario de atributo de calidad tiene seis partes: fuente del estímulo, estímulo, artefacto, entorno, respuesta y medida de la respuesta.
- Los escenarios generales son plantillas independientes del sistema para una cualidad; los escenarios concretos rellenan valores específicos para un sistema y sirven como requisitos.
- Un escenario sin medida de la respuesta, o con una media en lugar de un percentil para los tiempos, no se puede probar de forma fiable.
- Entre los errores habituales están mezclar varios estímulos en un mismo escenario, omitir el entorno y escribir una solución, como la elección de una tecnología, en lugar de la necesidad.
- Los escenarios se traducen directamente en pruebas de aceptación y, más adelante, en funciones de aptitud que comprueban la cualidad de forma continua.

## Por qué escenarios

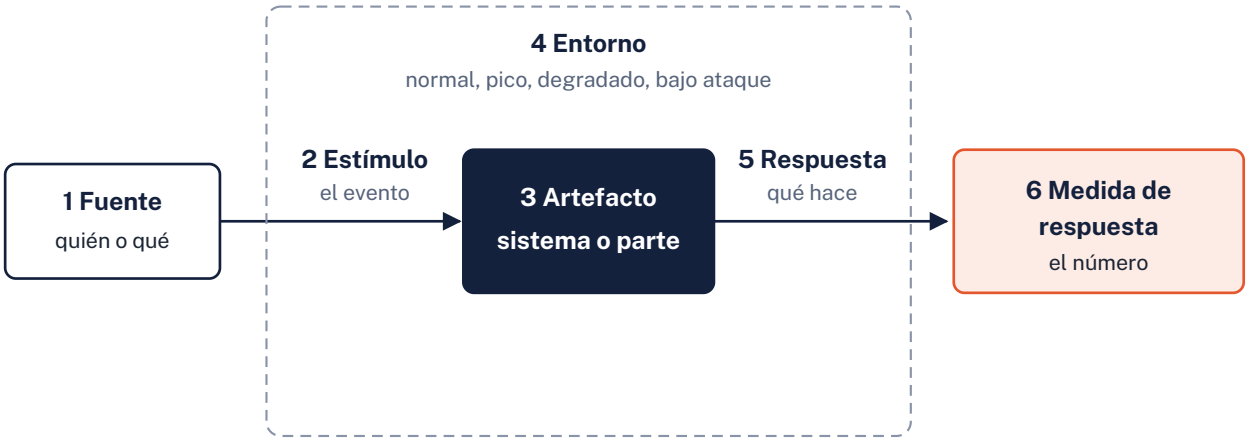
La Sección 2.2 te dio el vocabulario de las cualidades; esta sección te da la gramática. El nombre de una cualidad por sí solo, como rendimiento o seguridad, no permite diseñar ni probar nada. Ni siquiera basta con un número por sí solo: '2 segundos' no significa nada hasta que sabes para qué operación, con qué carga, medido dónde y para qué proporción de solicitudes. El escenario de atributo de calidad, desarrollado en el Software Engineering Institute (SEI) y expuesto por Len Bass, Paul Clements y Rick Kazman en *Software Architecture in Practice*, resuelve esto obligando a que todo requisito de calidad adopte la misma forma de seis partes.

La idea es sencilla. Toda cualidad se manifiesta cuando algo le ocurre a un sistema: una avalancha de usuarios, una caída, una solicitud de cambio, un ataque. Un escenario nombra ese evento, las circunstancias, la parte del sistema afectada, lo que debe ocurrir y cómo sabrás que ocurrió lo bastante bien. Es breve (normalmente de una a tres frases), concreto y comprobable. Y como todas las cualidades usan la misma forma, los escenarios de rendimiento, seguridad y modificabilidad pueden figurar uno junto a otro en una sola lista priorizada.

## Las seis partes

Las seis partes son la fuente del estímulo, el estímulo, el artefacto, el entorno, la respuesta y la medida de la respuesta. Apréndelas en ese orden, porque juntas cuentan una pequeña historia: *alguien o algo* hace *algo* a *una parte del sistema* en *ciertas condiciones*; el sistema *hace algo*, y lo *medimos* así. El estímulo y la medida de la respuesta son las dos partes que más a menudo se olvidan, y son las dos que más importan.

### Anatomía de un escenario




Ejemplo: 500 compradores por segundo (fuente, estímulo) llegan al proceso de pago (artefacto) en Black Friday (entorno); pedidos confirmados (respuesta) con p99 < 1.5 s y ninguno perdido (medida de la respuesta).

Una fuente produce un estímulo que llega a un artefacto dentro de un entorno; la respuesta del artefacto se evalúa con una medida de la respuesta.

### Las seis partes en detalle

| Parte | Significado | Ejemplos |
|---|---|---|
| **Fuente del estímulo** | La persona, el sistema o el evento que genera el estímulo. | Clientes en teléfonos móviles; un proceso por lotes interno; un desarrollador; un atacante; un disco que falla. |
| **Estímulo** | El evento al que debe responder el sistema. | Una ráfaga de solicitudes; la caída de un servidor; una solicitud de cambio; un intento de inicio de sesión con credenciales robadas. |
| **Artefacto** | La parte del sistema que recibe el estímulo: el sistema completo o un componente concreto. | El servicio de pago; la base de datos de pedidos; el módulo de precios; el pipeline de despliegue. |
| **Entorno** | Las condiciones en las que llega el estímulo. | Funcionamiento normal; pico de Black Friday; un centro de datos caído; durante un despliegue; en desarrollo. |
| **Respuesta** | Lo que hace el sistema (o las personas que lo rodean) como resultado. | Procesa el pedido; conmuta a la réplica; el cambio se hace, se prueba y se despliega; el intento se bloquea y se registra. |
| **Medida de la respuesta** | Cómo se mide la respuesta, para que el escenario se pueda probar. | p99 por debajo de 800 ms; servicio restablecido en 60 s sin pedidos perdidos; menos de 2 días-persona; 100% de los intentos bloqueados. |

El entorno merece especial atención porque el mismo estímulo puede exigir respuestas muy distintas en condiciones distintas. La caída de un servidor durante el funcionamiento normal puede requerir una conmutación por error automática en menos de un minuto; la misma caída durante una ventana de mantenimiento planificada puede ser aceptable con una recuperación manual. Una solicitud de cambio en tiempo de diseño (antes del lanzamiento) cuesta mucho menos que el mismo cambio cuando ya se han acumulado datos en producción. Escribir el entorno de forma explícita evita que se comparen escenarios que en realidad tratan de situaciones distintas.

El artefacto importa por una razón parecida. 'El sistema responde en menos de 200 ms' oculta qué parte es responsable. Nombrar el artefacto (el servicio de búsqueda, el API gateway, el cliente móvil) indica al equipo dónde deben hacerse el trabajo de diseño y la medición. Para algunas cualidades, como la modificabilidad y la desplegabilidad, el artefacto puede ser el código, la configuración o el pipeline de entrega en lugar de un componente en ejecución.

## Escenarios generales y concretos

Bass, Clements y Kazman distinguen dos tipos. Un escenario general es independiente del sistema: para un atributo de calidad, enumera los valores típicos que puede tomar cada parte. Para la disponibilidad, por ejemplo, las fuentes pueden ser internas o externas, personas, hardware, software o infraestructura; los estímulos pueden ser una caída, una omisión, una temporización incorrecta o una respuesta incorrecta; las respuestas pueden ser detectar, registrar, recuperar, enmascarar o degradar; y las medidas pueden ser el tiempo de detección, el tiempo de reparación, el porcentaje de disponibilidad o la proporción de solicitudes atendidas.

Un escenario concreto elige valores específicos para un sistema determinado: 'la base de datos principal de pedidos se cae durante el funcionamiento normal; la réplica toma el relevo y las escrituras se reanudan en menos de 60 segundos sin perder ningún pedido confirmado'. Los escenarios generales son plantillas y listas de comprobación; los escenarios concretos son requisitos. Usas el escenario general para asegurarte de haber considerado las fuentes, los estímulos y las medidas adecuados, y luego escribes el puñado de escenarios concretos que importan para tu sistema.

## De afirmaciones vagas a escenarios comprobables

La mayoría de los requisitos de calidad nacen como adjetivos. La habilidad consiste en hacer las preguntas que convierten cada adjetivo en un escenario: ¿qué operación?, ¿provocada por quién?, ¿con qué frecuencia o en qué cantidad?, ¿en qué condiciones?, ¿qué debe ocurrir?,

¿qué número nos dice que funcionó? La tabla muestra pares típicos de antes y después. Observa que las versiones de 'después' a veces revelan que la parte interesada quería decir varias cosas distintas, que pasan a ser varios escenarios.

**Antes y después**

| Afirmación vaga | Escenario comprobable |
|---|---|
| **El sitio debe ser rápido.** | Durante el pico de las tardes de los días laborables, de 300 búsquedas por segundo, la búsqueda de productos devuelve resultados en menos de 400 ms para el 95% de las solicitudes y en menos de 1 s para el 99%, medido en el balanceador de carga. |
| **El sistema debe tener alta disponibilidad.** | Si uno de los tres servidores de aplicaciones falla durante el funcionamiento normal, el servicio de reservas sigue aceptando reservas; no se pierde ninguna reserva confirmada y los usuarios ven errores durante no más de 30 segundos. |
| **El sistema debe ser seguro.** | Cuando un atacante envía 10 000 intentos de inicio de sesión por minuto con credenciales filtradas, el sistema bloquea el origen tras 20 fallos, avisa al equipo de seguridad en menos de 5 minutos, y los usuarios legítimos desde otras direcciones pueden seguir iniciando sesión. |
| **El código debe ser fácil de cambiar.** | Cuando el negocio pide una nueva opción de entrega, un desarrollador que conozca el código puede implementarla, probarla y desplegarla en no más de 3 días, cambiando solo el módulo de entregas y su configuración. |
| **La app debe ser fácil de usar.** | Un usuario nuevo sin formación puede registrarse y hacer un pedido en menos de 4 minutos; el 90% de los participantes en la prueba lo consigue al primer intento. |
| **Los despliegues deben ser fáciles.** | Un desarrollador puede lanzar a producción un cambio en el servicio de pedidos en menos de 30 minutos tras el merge, sin tiempo de inactividad, y revertirlo en menos de 5 minutos si suben las tasas de error. |

SI ESTO ES NUEVO PARA TI

### Si eres nuevo en esto: ¿de dónde salen los números?

A menudo se duda en escribir un número porque no se sabe cuál es el 'correcto'. Empieza por la evidencia: los registros de tráfico actuales, la monitorización del sistema antiguo, el comportamiento de la competencia, los objetivos de negocio ('prevemos duplicar los pedidos el año que viene') y los estudios sobre la paciencia de los usuarios. Después propón una cifra y deja que las partes interesadas reaccionen. '¿Son aceptables 400 milisegundos para el 95% de las búsquedas?' da pie a una conversación mucho mejor que '¿qué tan rápido debería ser?'. Un número acordado hoy puede revisarse mañana; un requisito sin número ni siquiera puede discutirse.

## Ejemplos de escenarios para seis cualidades

Los ejemplos siguientes usan una misma tienda en línea para que veas cómo se ven distintas cualidades en el mismo formato. Cada uno está escrito de forma compacta, con las seis partes etiquetadas; en la práctica, los equipos suelen escribirlos en una o dos frases.

**Escenarios concretos para una tienda en línea**

| Calidad | Escenario (fuente, estímulo, artefacto, entorno, respuesta, medida) |
|---|---|
| **Rendimiento** | Fuente: clientes con sesión iniciada. Estímulo: 2000 envíos de pedidos por minuto. Artefacto: servicio de pago. Entorno: pico del viernes por la tarde. Respuesta: pedidos validados, almacenados y confirmados. Medida: p95 ≤ 1.5 s, p99 ≤ 3 s, cero pedidos perdidos. |
| **Disponibilidad** | Fuente: hardware interno. Estímulo: se cae el nodo principal de la base de datos. Artefacto: base de datos de pedidos. Entorno: funcionamiento normal. Respuesta: réplica promovida automáticamente, solicitudes en curso reintentadas. Medida: las escrituras se reanudan en menos de 60 s; ningún pedido confirmado perdido (RPO = 0). |
| **Modificabilidad** | Fuente: product owner. Estímulo: solicitud para añadir un nuevo método de pago. Artefacto: componente de pagos. Entorno: tiempo de diseño, tras el lanzamiento inicial. Respuesta: cambio hecho, probado y desplegado. Medida: ≤ 5 días-persona; ningún cambio fuera del componente de pagos; ninguna regresión en los métodos existentes. |
| **Seguridad** | Fuente: un cliente con sesión iniciada que actúa de forma maliciosa. Estímulo: intenta leer los pedidos de otro cliente cambiando un ID en la URL. Artefacto: API de pedidos. Entorno: funcionamiento normal. Respuesta: solicitud denegada y registrada con la identidad de quien llama. Medida: 100% de esas solicitudes rechazadas; alerta generada en menos de 1 minuto tras 5 intentos. |
| **Usabilidad** | Fuente: cliente nuevo. Estímulo: quiere cambiar la dirección de entrega después de hacer el pedido. Artefacto: frontends web y móvil. Entorno: en ejecución, sin ayuda disponible. Respuesta: completa el cambio sin ayuda. Medida: el 90% de los participantes en la prueba de usabilidad lo consigue en menos de 2 minutos. |
| **Desplegabilidad** | Fuente: desarrollador. Estímulo: un cambio fusionado en el servicio de catálogo. Artefacto: pipeline de despliegue y servicio de catálogo. Entorno: horario laboral, carga normal. Respuesta: compilado, probado y desplegado de forma progresiva. Medida: en producción en menos de 45 minutos; cero tiempo de inactividad; rollback automático en menos de 5 minutos si la tasa de error supera el 1%. |

Vale la pena fijarse en dos rasgos de estos ejemplos. Primero, los escenarios no tratan solo del software en ejecución. Los escenarios de modificabilidad y desplegabilidad tienen como fuentes a desarrolladores y product owners, y sus medidas son el esfuerzo, el tiempo transcurrido y el alcance del cambio. Segundo, las buenas medidas suelen combinar un

número de velocidad con un número de corrección: 'en menos de 60 segundos' y 'ningún pedido confirmado perdido'. Una conmutación por error rápida que pierde datos no es lo que quería el negocio.

## Errores comunes

Al revisar cientos de escenarios escritos por equipos aparecen los mismos errores una y otra vez. Cada uno hace que un escenario sea imposible de probar o engañoso.

**Errores y cómo corregirlos**

| Error | Ejemplo | Corrección |
|---|---|---|
| **Sin medida de la respuesta** | 'Con carga máxima, el proceso de pago responde rápidamente.' | Añade un número y una estadística: 'p95 ≤ 1.5 s, p99 ≤ 3 s'. |
| **Media en lugar de percentil** | 'Tiempo de respuesta medio por debajo de 300 ms.' | Las medias ocultan los valores atípicos lentos; especifica percentiles (Sección 2.4). |
| **Varios estímulos en un escenario** | 'Cuando la carga se duplica, falla un servidor o se ejecuta un despliegue, el sistema sigue siendo rápido.' | Divídelo en tres escenarios, cada uno con su propio entorno y su propia medida. |
| **Solución disfrazada de requisito** | 'El sistema usa Redis para almacenar en caché las páginas de producto.' | Expresa la necesidad ('las páginas de producto cargan en menos de 300 ms para el 95% de las visualizaciones con 1000 visualizaciones por segundo') y deja la caché como opción de diseño. |
| **Falta el entorno** | 'El servicio se recupera en menos de 5 minutos.' | Di de qué y cuándo: 'tras la caída de una sola zona durante el funcionamiento normal'. |
| **Palabras que no se pueden medir** | 'Con elegancia', 'sin interrupciones', 'de forma robusta', 'fácil de usar'. | Sustituye cada una por un comportamiento observable y un número. |
| **Absolutos imposibles** | 'El sistema nunca se cae.' '100% de los ataques bloqueados.' | Usa objetivos realistas y di qué ocurre cuando no se cumplen. |

**ATENCIÓN**

### Soluciones disfrazadas de requisitos

Este es el error más dañino, porque parece un avance. 'Usar Kubernetes', 'usar microservicios', 'usar una cola de mensajes' son decisiones de diseño. Escribirlas como requisitos se salta el paso en el que el equipo se pregunta si son la mejor forma de lograr la cualidad, y elimina la evidencia necesaria para revisar la elección más adelante. Si una parte interesada insiste en una tecnología, pregunta qué problema resuelve; la respuesta es el escenario real. Si la tecnología viene impuesta de verdad por una política, regístrala como restricción indicando su origen.

GUÍA DE DECISIÓN

### Guía de decisión: ¿está listo este escenario?

Pasa cada borrador de escenario por estas preguntas antes de que entre en el documento de arquitectura o en el backlog.

**Q1** **¿Describe un único estímulo con una fuente nombrada?**

- **Sí** Continúa.
- **No** Divídelo, o nombra quién o qué provoca el evento.

**Q2** **¿Indica el entorno (normal, pico, degradado, durante un despliegue, en tiempo de diseño)?**

- **Sí** Continúa.
- **No** Añade las condiciones; de lo contrario, la medida no se puede interpretar.

**Q3** **¿Nombra el artefacto, la parte del sistema responsable?**

- **Sí** Continúa.
- **No** Nombra el componente, la interfaz o el pipeline para que el equipo sepa dónde diseñar y medir.

**Q4** **¿Es la respuesta un comportamiento observable y no la elección de una tecnología?**

- **Sí** Continúa.
- **No** Reescribe la necesidad y lleva la tecnología a una decisión de diseño o a una restricción.

**Q5** **¿Contiene la medida de la respuesta un número, una unidad y, para los tiempos, un percentil?**

- **Sí** El escenario es comprobable: añade una prioridad y un responsable.
- **No** Añade la medida; sin ella, el escenario es un deseo.

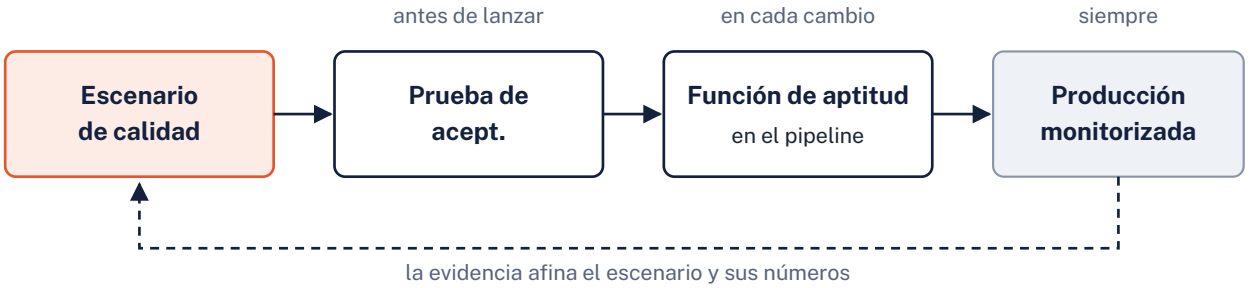
## De los escenarios a las pruebas de aceptación y las funciones de aptitud

Un escenario bien escrito ya es la mayor parte de una prueba. La fuente y el estímulo te dicen qué carga o qué evento generar, el entorno te dice cómo preparar la prueba, el artefacto te dice qué observar y la medida de la respuesta da la condición para aprobarla. Un escenario de rendimiento se convierte en una prueba de carga; un escenario de disponibilidad, en un

simulacro de conmutación por error o un experimento de caos; un escenario de modificabilidad, en una revisión de cuánto código tocó un cambio de muestra y cuánto tardó; un escenario de seguridad, en un caso de prueba de penetración.

El siguiente paso es seguir comprobando. Una función de aptitud (fitness function), término popularizado por Neal Ford, Rebecca Parsons y Patrick Kua en *Building Evolutionary Architectures*, es cualquier comprobación automatizada o repetible de en qué medida el sistema cumple un objetivo arquitectónico. Ejecutada en el pipeline de entrega o sobre la telemetría de producción, convierte un escenario de una prueba de aceptación puntual en una protección permanente: la compilación falla, o salta una alerta, cuando la cualidad se desvía. El Capítulo 11 trata las funciones de aptitud en profundidad en Funciones de aptitud y arquitectura evolutiva.

**La vida de un escenario**




Un escenario guía una prueba de aceptación y luego una función de aptitud que se ejecuta en cada cambio, y la evidencia de producción vuelve para afinar el escenario.

```
# A performance scenario expressed as an automated check (illustrative)
# Scenario: at 300 searches/s, p95 ⩽ 400 ms and p99 ⩽ 1000 ms.
import statistics

def check_search_latency(latencies_ms, p95_target=400, p99_target=1000):
    cuts = statistics.quantiles(latencies_ms, n=100)  # 99 cut points
    p95, p99 = cuts[94], cuts[98]
    assert p95 ⩽ p95_target, f"p95 {p95:.0f} ms exceeds {p95_target} ms"
    assert p99 ⩽ p99_target, f"p99 {p99:.0f} ms exceeds {p99_target} ms"
    return p95, p99

# latencies_ms comes from a load test that replays 300 searches per second
# for ten minutes against a production-like environment.
```

Un escenario de rendimiento convertido en una sencilla comprobación automatizada de la latencia p95 y p99.

## Cómo usan los escenarios los profesionales

En la práctica, los equipos rara vez escriben largas tablas de escenarios de seis partes para todo. Escriben un número reducido, a menudo de diez a treinta para un sistema de cierta envergadura, para las cualidades que son arquitectónicamente significativas, y los guardan donde todos puedan verlos: en el documento de arquitectura (la sección 10 de arc42 pide exactamente esto), en el backlog como criterios de aceptación o en un tablero de calidad compartido. Algunos equipos usan una forma más corta; el proyecto Q42 de Gernot Starke, por ejemplo, recomienda una estructura compacta de contexto, desencadenante y criterios de aceptación. La forma importa menos que la disciplina: toda cualidad importante tiene un estímulo, un contexto y un número.

Los escenarios también sirven de motor a los métodos de evaluación. El Quality Attribute Workshop y el Architecture Tradeoff Analysis Method (ATAM), ambos del SEI, recogen y priorizan escenarios con las partes interesadas y después ponen a prueba la arquitectura frente a los más importantes. La Sección 2.8 muestra cómo se recogen los escenarios y se clasifican en un árbol de utilidad.

**COMPRUEBA LO QUE HAS ENTENDIDO**

▼ **Nombra en orden las seis partes de un escenario de atributo de calidad.**

Fuente del estímulo, estímulo, artefacto, entorno, respuesta y medida de la respuesta.

▼ **¿Qué tiene de malo 'el sistema usa una CDN para que las páginas carguen rápido'?**

Expresa una solución, no una necesidad, y no tiene medida. El requisito debería describir la carga, las condiciones y el tiempo de respuesta objetivo; la CDN es un diseño posible para cumplirlo.

▼ **¿Quién suele ser la fuente del estímulo en un escenario de modificabilidad?**

Una persona que quiere un cambio, como un product owner, un desarrollador o un regulador; el estímulo es la solicitud de cambio.

▼ **¿En qué se diferencia un escenario de una función de aptitud?**

Un escenario es la formulación de un requisito; una función de aptitud es una comprobación automatizada o repetible, ejecutada de forma continua, que mide si el sistema lo cumple.

▼ **Inténtalo: Convierte "el sitio de entradas debe seguir siendo rápido el día de la venta" en un escenario de atributo de calidad de seis partes, inventando cifras razonables. Nombra cada parte. Luego indica una razón por la que la frase original nunca podría probarse.**

Fuente: navegadores de los fans. Estímulo: llegan 2500 personas por segundo. Artefacto: el flujo de selección de asientos y compra. Entorno: pico de la salida a la venta, todas las zonas en buen estado. Respuesta: se atiende cada solicitud de página o se coloca al fan en una sala de espera. Medida de respuesta: respuesta de página p99 inferior a 2 segundos. El original no se podía probar porque "rápido" no tiene número, y no nombra ninguna carga ni condición, así que nadie podría decir si lo había cumplido.

---

**FUENTES**


- SEI: Software Architecture in Practice, 4th edition: https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- arc42 section 10: Quality requirements and quality scenarios: https://docs.arc42.org/section-10/
- Q42: How to specify quality requirements: https://quality.arc42.org/articles/specify-quality-requirements

SECCIÓN 2.4 DE 2.8 · INTERMEDIO · 27 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

# Rendimiento y escalabilidad: latencia, rendimiento (throughput), percentiles y carga

## ANTES DE LEER ESTO

Esta sección se basa en:

- 0.2 Datos y algoritmos
- 0.4 Comunicación en red
- 0.7 Servidores y la nube

Términos anteriores que utiliza: Servidor, Caché, Base de datos

## LA IDEA EN BREVE

Imagina un supermercado. El tiempo que pasas en la caja es la latencia; cuántos compradores atienden las cajas en una hora es el rendimiento (throughput). Tu tiempo en la caja es en parte el escaneo (el trabajo en sí) y en parte la cola que tienes delante. Cuando las cajas están ocupadas solo la mitad del tiempo apenas haces cola; cuando están ocupadas casi siempre, las colas crecen muchísimo y muy deprisa, aunque cada escaneo tarde lo mismo. Abrir más cajas ayuda, hasta que todos tienen que compartir un único encargado para los reembolsos. El rendimiento y la escalabilidad tratan precisamente de estos efectos en el software.

## LO QUE NECESITAS SABER

- La latencia es cuánto tarda una unidad de trabajo; el rendimiento (throughput) es cuánto trabajo se completa por unidad de tiempo. Un requisito de rendimiento debe indicar ambos, junto con la carga a la que se aplican.
- El tiempo de respuesta es el tiempo de servicio más el tiempo de espera (más el tiempo de red); con carga, el tiempo de espera suele ser el dominante.
- Especifica y monitoriza percentiles como p95 y p99, no medias; en los sistemas con fan-out, una respuesta lenta poco frecuente en un servidor se vuelve habitual para los usuarios (Dean y Barroso, 2013).
- La ley de Little, $L = \lambda \times W$, relaciona la tasa de solicitudes, el tiempo de respuesta y la concurrencia, y es la base para dimensionar grupos (pools) y límites.
- En el modelo de cola más sencillo, M/M/1, el tiempo de respuesta medio es $S \div (1 - \rho)$, por lo que se duplica entre el 80% y el 90% de utilización; otras colas difieren en los detalles, pero muestran el mismo abismo, así que mantén los recursos críticos muy por debajo de la saturación.

- La ley de Amdahl limita la aceleración a 1 ÷ s para una fracción serie s; la Ley de escalabilidad universal añade costos de coherencia que hacen que la capacidad alcance un máximo y luego caiga.

## Dos preguntas distintas: qué tan rápido y cuánto

El rendimiento es la cualidad que la gente menciona primero y especifica peor. Abarca dos preguntas distintas. La latencia pregunta cuánto tarda una unidad de trabajo: cuántos milisegundos pasan entre tocar 'Pagar' y ver 'Pago aceptado'. El rendimiento (throughput) pregunta cuánto trabajo completa el sistema por unidad de tiempo: 1200 pagos por segundo, 40 000 pedidos por hora. Ambos están relacionados, pero no son intercambiables. Un autobús que lleva a 50 personas de Londres a Oxford en 100 minutos tiene peor latencia y un throughput mucho mejor que un deportivo que lleva a dos personas en 60 minutos.

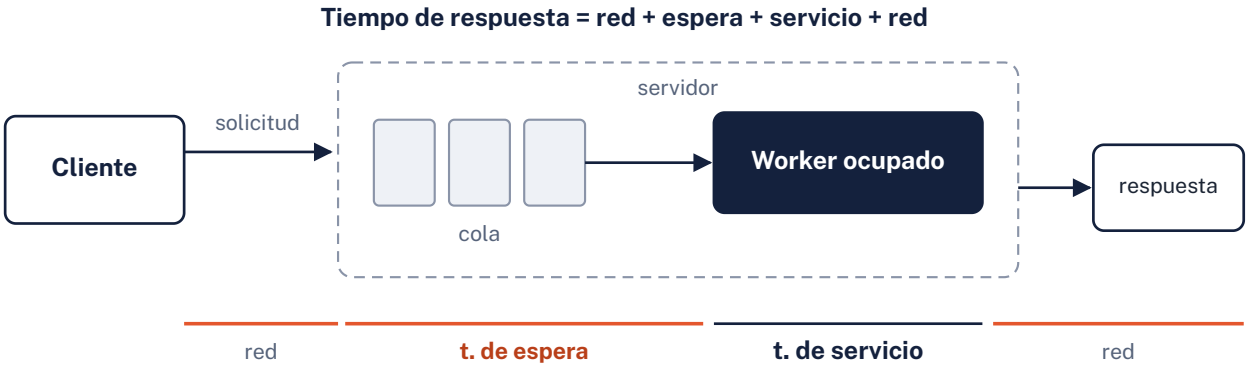
En el software, la misma tensión aparece en todas partes. Agrupar las escrituras en la base de datos mejora el throughput, pero hace que cada escritura individual espere a su lote. Añadir una caché mejora ambos para las lecturas repetidas, pero no hace nada por la primera. Por eso un buen requisito de rendimiento dice a qué pregunta se refiere y, normalmente, indica ambas cosas: 'con 1200 pagos por segundo (throughput), el 99% se completa en menos de 800 ms (latencia)'. ISO/IEC 25010 las clasifica dentro de la eficiencia de rendimiento como comportamiento temporal, utilización de recursos y capacidad (Sección 2.2).

## Tiempo de respuesta = tiempo de servicio + tiempo de espera

Cuando mides la latencia desde el lado del cliente obtienes el tiempo de respuesta: todo lo que ocurrió entre el envío de la solicitud y la recepción de la respuesta. Conviene dividirlo en partes. El tiempo de servicio es el tiempo que un recurso dedica realmente a trabajar en tu solicitud: analizarla, ejecutar la consulta, generar el resultado. El tiempo de espera es el tiempo que tu solicitud pasa en una cola porque el recurso está ocupado con la de otro. A esto se suma el tiempo de transferencia por la red, en ambos sentidos.

La distinción importa porque las dos partes responden a remedios distintos. El tiempo de servicio baja cuando abaratas el propio trabajo: un índice mejor, una carga útil más pequeña, un algoritmo más rápido. El tiempo de espera baja cuando añades capacidad o reduces la carga, y a menudo es mucho mayor que el tiempo de servicio cuando un sistema está ocupado. Muchas investigaciones de 'base de datos lenta' terminan con el descubrimiento de que cada consulta tarda 5 ms, pero espera 300 ms a que quede libre una conexión del grupo (pool).

**Adónde va el tiempo**




El tiempo de respuesta que ve un cliente incluye la transferencia por la red, la espera en una cola hasta que un worker ocupado queda libre y el propio tiempo de servicio.

## Percentiles: por qué las medias engañan

Los tiempos de respuesta nunca son un único número; forman una distribución, y casi siempre es asimétrica. La mayoría de las solicitudes son rápidas, y una minoría son lentas porque se topan con una caché fría, una pausa de recolección de basura, un bloqueo, un reintento o un vecino sobrecargado. La media aritmética mezcla los dos grupos en una cifra que no describe a nadie. Los profesionales usan percentiles en su lugar. El percentil 50 (p50, la mediana) es el valor que supera en rapidez la mitad de las solicitudes; p95 es el valor que supera el 95% de las solicitudes; p99 y p99.9 describen el 1% y el 0.1% más lentos.

El extremo lento de la distribución se llama cola (tail), e importa más de lo que parece. Los usuarios de la cola suelen ser tus mejores clientes: los de carritos más grandes, historiales más largos y más datos. Un usuario que carga 40 recursos para ver una página experimenta el más lento de esos 40, no la media. Y la probabilidad de que una página de 40 recursos se tope con al menos una solicitud de p99 es $1 - 0.99^{40} \approx 33\%$: un evento 'raro' del 1% afecta a un tercio de las visualizaciones de página.

EJEMPLO PRÁCTICO

## Ejemplo práctico: la media que oculta el problema

Una API de búsqueda de productos registra 1000 tiempos de respuesta en un minuto: 900 de 80 ms, 80 de 150 ms, 15 de 1200 ms (una caché fría) y 5 de 4000 ms (un timeout y un reintento).

| | |
|---|---|
| **Suma de todos los tiempos** | 900 × 80 + 80 × 150 + 15 × 1200 + 5 × 4000 = 72 000 + 12 000 + 18 000 + 20 000 = 122 000 ms |
| **Media** | 122 000 ÷ 1000 = 122 ms. Parece saludable frente a un objetivo de 200 ms. |
| **p50 (mediana)** | La solicitud número 500 más rápida: 80 ms |
| **p95** | La solicitud número 950 más rápida cae dentro del grupo de 150 ms (solicitudes 901 a 980): 150 ms |
| **p99** | La número 990 cae dentro del grupo de 1200 ms (solicitudes 981 a 995): 1200 ms |
| **p99.9** | La número 999 está en el grupo de 4000 ms (996 a 1000): 4000 ms |
| **A quién afecta** | 20 solicitudes de cada 1000 (el 2%) tardaron más de un segundo. Con 300 búsquedas por segundo, eso son 6 usuarios cada segundo, o unos 21 600 por hora. |

La media de 122 ms cumple un objetivo de 200 ms mientras el 2% de los usuarios espera más de un segundo. Un escenario escrito como 'media por debajo de 200 ms' daría por bueno este sistema.

Escribir el requisito como 'p95 ≤ 200 ms y p99 ≤ 500 ms' lo suspende, con razón, y dirige al equipo hacia las rutas de caché fría y de reintento.

Los percentiles no se pueden promediar. El p99 de diez servidores no es la media de sus diez valores p99; debe calcularse a partir de los datos combinados, o aproximarse a partir de histogramas fusionados.

```
# Why the mean misleads: 1,000 response times from one minute of traffic
import statistics

times_ms = [80] * 900 + [150] * 80 + [1200] * 15 + [4000] * 5

mean = statistics.mean(times_ms)
cuts = statistics.quantiles(times_ms, n=1000, method="inclusive")
p50, p95, p99, p999 = cuts[499], cuts[949], cuts[989], cuts[998]
print(f"mean {mean:.0f} ms, p50 {p50:.0f}, p95 {p95:.0f}, p99 {p99:.0f}, p99.9
{p999:.0f}")
# mean 122 ms, p50 80, p95 150, p99 1200, p99.9 4000
```

El mismo cálculo en unas pocas líneas de Python; la media parece correcta, mientras que p99 y p99.9 revelan el problema.

## La latencia de cola a gran escala

La cola pasa a ser lo principal en los sistemas grandes que usan fan-out (abanico), en los que una solicitud de usuario se divide en muchas solicitudes paralelas a otros servidores y la respuesta debe esperar a todas ellas. Un buscador, un feed social o una página de recomendaciones pueden consultar decenas o cientos de shards de backend en cada consulta. Jeffrey Dean y Luiz André Barroso, de Google, lo analizaron en *The Tail at Scale* (Communications of the ACM, 2013).

Su ilustración es sencilla. Supón que cada servidor responde normalmente en 10 ms, pero tarda 1 segundo en una de cada 100 solicitudes. Una solicitud de usuario que espera a un solo servidor es lenta el 1% de las veces. Una solicitud de usuario que espera a 100 servidores así es lenta siempre que cualquiera de ellos lo sea: $1 - 0.99^{100} \approx 63\%$ de las veces. Un evento raro en un servidor se convierte en el caso habitual para el usuario. Dean y Barroso aportan mediciones de un servicio de Google en el que la latencia del percentil 99 para una única solicitud hoja era de 10 ms, pero el percentil 99 de la espera de todas las solicitudes hoja era de 140 ms.

Como no se puede eliminar toda fuente de variabilidad (hardware compartido, trabajos en segundo plano, recolección de basura, colas de red), defienden diseños tolerantes a la cola, igual que los diseños tolerantes a fallos aceptan que el hardware falla. Sus técnicas incluyen las solicitudes de cobertura (hedged requests: enviar una segunda copia a otra réplica si la primera no ha respondido en, por ejemplo, el tiempo del percentil 95, y usar la que vuelva primero), las solicitudes vinculadas que se cancelan entre sí, particiones más finas que se pueden reequilibrar, excluir temporalmente las máquinas lentas y devolver un resultado suficientemente bueno cuando han respondido la mayoría de los shards, pero no todos. En una prueba de rendimiento que describen, enviar la solicitud de cobertura a los 10 ms redujo la latencia del percentil 99.9 de 1800 ms a 74 ms enviando solo alrededor de un 2% más de solicitudes.

## La ley de Little: cuántas solicitudes hay en curso

La ley de Little, demostrada por John Little en 1961, es la fórmula individual más útil en la planificación de capacidad. Para cualquier sistema estable, el número medio de elementos dentro de él (L) es igual a la tasa media de llegada (λ, lambda) multiplicada por el tiempo medio que cada elemento pasa dentro (W): **L = λ × W**. Se cumple sea cual sea el patrón de llegada o el orden de servicio, siempre que el sistema sea estable (en promedio, salen tantos elementos como llegan) y uses unidades coherentes.

En el software, L es el número de solicitudes en curso a la vez (la concurrencia), λ es la tasa de solicitudes y W es el tiempo de respuesta. Si un servicio recibe 200 solicitudes por segundo y cada una pasa 0.25 segundos dentro, entonces en promedio hay 200 × 0.25 = 50 solicitudes en curso en cualquier momento. Ese número determina los grupos de hilos, los grupos de conexiones, la memoria y el número de licencias. La ley también funciona a la inversa: si puedes atender como máximo 50 solicitudes a la vez y cada una tarda 0.25 s, no puedes sostener más de 50 ÷ 0.25 = 200 solicitudes por segundo.

EJEMPLO PRÁCTICO

### Ejemplo práctico: dimensionar un grupo de conexiones a la base de datos con la ley de Little

El servicio de viajes de una app de transporte con conductor atiende 1800 solicitudes por segundo en el pico de la tarde. Cada solicitud hace llamadas a la base de datos que mantienen ocupada una conexión durante 40 ms en total. Cada una de las 6 instancias del servicio tiene un grupo de 10 conexiones. ¿Es suficiente?

| | |
|---|---|
| **Tasa de llegada λ** | 1800 solicitudes por segundo |
| **Tiempo que se retiene una conexión W** | 40 ms = 0.040 s |
| **Conexiones en uso L = λ × W** | 1800 × 0.040 = 72 conexiones ocupadas en promedio |
| **Conexiones disponibles** | 6 instancias × 10 = 60 |
| **Veredicto** | 72 > 60. Las solicitudes harán cola para obtener conexiones, el tiempo de espera crecerá sin límite en el pico y los tiempos de respuesta subirán hasta que empiecen los timeouts. |
| **Objetivo con margen** | Procura mantener el grupo como máximo a un 70% de utilización: 72 ÷ 0.7 ≈ 103 conexiones, es decir, unas 18 por instancia |
| **Comprueba la base de datos** | El servidor de base de datos también debe poder aceptar 103 conexiones; si no, reduce W (consultas más rápidas, menos viajes de ida y vuelta) o el número de llamadas a la base de datos por solicitud |
| **¿Y si las consultas se vuelven más rápidas?** | Reducir W a 25 ms da L = 1800 × 0.025 = 45 conexiones, que caben en las 60 existentes con un 75% de utilización |

La ley de Little convierte un objetivo de throughput y una latencia en una cifra de concurrencia que se puede comparar directamente con un límite configurado.

Da promedios. Las ráfagas elevan el número instantáneo, y por eso el ejemplo añade margen en lugar de dimensionar exactamente para 72.

## Utilización y colas: por qué los sistemas caen al abismo

La utilización (ρ, rho) es la fracción del tiempo en que un recurso está ocupado. Un servidor que recibe 40 solicitudes por segundo, cada una de las cuales necesita 20 ms de servicio, está ocupado 40 × 0.020 = 0.8 del tiempo: un 80% de utilización. La intuición sugiere que un recurso

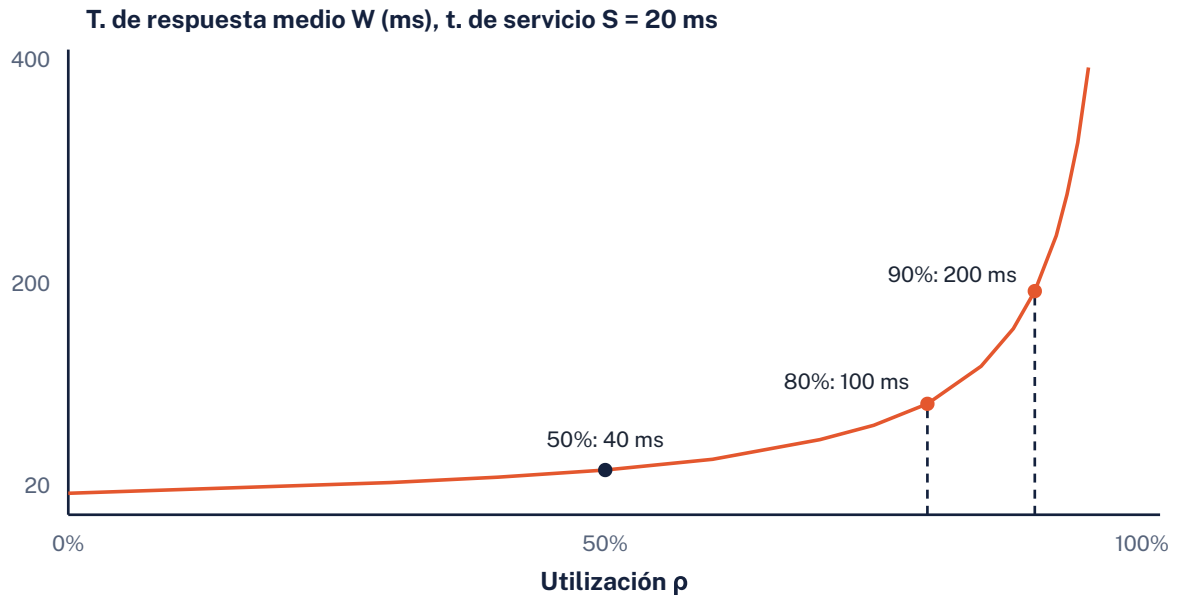
ocupado al 80% solo debería ser un poco más lento que uno ocupado al 50%. La teoría de colas dice lo contrario, porque las solicitudes no llegan de forma uniforme; se agrupan, y cuando se agrupan, las últimas esperan.

El modelo más sencillo es la cola M/M/1: llegadas aleatorias, tiempos de servicio aleatorios y un único servidor. Su tiempo de respuesta medio es **W = S ÷ (1 − ρ)**, donde S es el tiempo de servicio medio. Con un 50% de utilización, W = 2S. Con un 80%, W = 5S. Con un 90%, W = 10S. Con un 99%, W = 100S. El denominador se acerca a cero a medida que la utilización se aproxima al 100%, así que el tiempo de respuesta crece sin límite. Los sistemas reales no son exactamente M/M/1, y los grupos con varios servidores se comportan mejor con la misma utilización, pero la forma es universal: plana, luego un codo y luego un abismo.

**Cola M/M/1: tiempo de respuesta medio para un tiempo de servicio de 20 ms**

| Utilización ρ | W = S ÷ (1 − ρ) | Tiempo de respuesta medio | De ello, espera |
|---|---|---|---|
| **50%** | 20 ÷ 0.5 | 40 ms | 20 ms |
| **70%** | 20 ÷ 0.3 | 67 ms | 47 ms |
| **80%** | 20 ÷ 0.2 | 100 ms | 80 ms |
| **90%** | 20 ÷ 0.1 | 200 ms | 180 ms |
| **95%** | 20 ÷ 0.05 | 400 ms | 380 ms |
| **99%** | 20 ÷ 0.01 | 2000 ms | 1980 ms |

**El abismo de la utilización**




En una cola M/M/1, el tiempo de respuesta medio se duplica entre el 80% y el 90% de utilización y sigue subiendo con fuerza hacia el 100%.

Las lecciones prácticas son directas. No planifiques ejecutar recursos críticos cerca del 100% de ocupación; deja margen para que las ráfagas no te empujen más allá del codo. Vigila la utilización y la longitud de la cola a la vez (el método USE de Brendan Gregg comprueba la utilización, la saturación y los errores de cada recurso). Y recuerda que el abismo hace peligrosos los cambios pequeños: con un 90% de utilización, un aumento del tráfico de solo un 6% aproximadamente te lleva al 95% y duplica el tiempo de respuesta medio. El Capítulo 9 trata las colas, el back pressure y la planificación de capacidad en Asincronía, colas y back pressure.

**SI ESTO ES NUEVO PARA TI**

### Paso a paso: por qué un servidor ocupado hace esperar a las solicitudes

El punto en el que más gente tropieza aquí es por qué la espera crece tan deprisa a medida que un recurso se ocupa más, aunque cada solicitud no necesite más trabajo que antes. Sigue a un servidor mientras su tráfico aumenta.

- Cada solicitud necesita 20 ms de trabajo en promedio: su tiempo de servicio. Con 40 solicitudes por segundo, el servidor está ocupado 40 × 0.020 = 0.8 del tiempo, así que su utilización es del 80%.
- Si las solicitudes llegaran de forma uniforme, una cada 25 ms, cada una encontraría el servidor libre y tardaría solo 20 ms.
- Las solicitudes reales llegan al azar, así que se agrupan. Si llegan tres a la vez, la primera se atiende enseguida, la segunda espera unos 20 ms y la tercera unos 40 ms. Puede seguir un hueco en el que el servidor queda inactivo, y el tiempo inactivo no se puede guardar para el siguiente grupo.
- Con un 50% de ocupación hay muchos huecos, así que las colas se vacían pronto. Con un 80% hay menos, y la cola de un grupo a menudo sigue ahí cuando llega el siguiente. Cerca del 100% casi no queda tiempo libre, y las colas se vuelven muy largas.
- La tabla de la sección para un solo servidor con llegadas aleatorias (la cola M/M/1) muestra el efecto: un tiempo de respuesta medio de 40 ms con un 50% de ocupación, 100 ms con un 80% y 200 ms con un 90%. Pasar de 40 a 45 solicitudes por segundo lleva al servidor del 80% al 90% de ocupación y duplica el tiempo de respuesta medio.
- **La lección:** el trabajo en sí no se ha vuelto más lento; todo el tiempo extra es encolamiento. Mantén los recursos críticos muy por debajo de su capacidad máxima, porque cerca del máximo un pequeño aumento del tráfico provoca un gran aumento del tiempo de respuesta.

## Escalabilidad: carga, elasticidad, escalado vertical y horizontal

La escalabilidad es la capacidad de manejar más carga añadiendo recursos, idealmente con una capacidad que crezca en proporción y sin rediseñar. Solo significa algo cuando dices qué es la 'carga'. Los parámetros de carga difieren entre sistemas: solicitudes por segundo para una API, conexiones simultáneas para un servicio de chat, la proporción entre lecturas y escrituras para una red social, el volumen de datos para una plataforma de analítica o el número de seguidores de la cuenta más seguida para un feed. Un escenario de escalabilidad nombra el parámetro de carga, el crecimiento y la respuesta que debe mantenerse: 'cuando los pedidos diarios crecen de 50 000 a 500 000, la latencia p95 del proceso de pago se mantiene por debajo de 1.5 s añadiendo instancias, con un costo por pedido no más de un 20% mayor'.

La elasticidad añade el factor tiempo: la capacidad de añadir y quitar capacidad automáticamente a medida que la carga sube y baja, de modo que un sitio de venta de entradas pueda crecer para una venta y reducirse después. Es una parte importante del argumento económico a favor de las plataformas en la nube. Hay dos formas básicas de añadir capacidad. El escalado vertical (scaling up) da a una máquina más CPU, más memoria o un almacenamiento más rápido; es sencillo, no requiere cambios en el software y choca con límites estrictos y precios crecientes. El escalado horizontal (scaling out) añade más máquinas y reparte el trabajo; no tiene un techo fijo, pero exige que el software esté diseñado para ello, con instancias sin estado, balanceo de carga y datos particionados. El Capítulo 9 trata las estrategias en detalle en Estrategias de escalado: vertical, horizontal y el cubo de escalado.

## La ley de Amdahl: el límite del paralelismo

Añadir procesadores o nodos solo ayuda con el trabajo que se puede hacer en paralelo. La ley de Amdahl, del artículo de Gene Amdahl de 1967 sobre los límites de la computación paralela, recoge esta idea. Si una fracción s del trabajo es serie (debe hacerse paso a paso) y el resto, 1 − s, se puede repartir perfectamente entre N procesadores, la mejor aceleración posible es **aceleración(N) = 1 ÷ (s + (1 − s) ÷ N)**. A medida que N crece sin límite, la aceleración se aproxima a 1 ÷ s. Con un 10% de trabajo serie, ningún número de procesadores puede hacer el trabajo más de diez veces más rápido.

EJEMPLO PRÁCTICO

### Ejemplo práctico: un proceso de facturación nocturno

La facturación nocturna de una empresa de telecomunicaciones tarda 8 horas en un servidor. El perfilado muestra que el 10% del tiempo es serie (cargar las tarifas y después escribir un único archivo de resumen); el otro 90% tarifica las llamadas de forma independiente por cliente. La ejecución debe terminar en menos de 1.5 horas. ¿Cuántos servidores se necesitan?

| | |
|---|---|
| **Fracción serie s** | 0.10; fracción paralela 0.90 |
| **Aceleración necesaria** | 8 h ÷ 1.5 h ≈ 5.33 |
| **N = 4** | 1 ÷ (0.10 + 0.90 ÷ 4) = 1 ÷ 0.325 ≈ 3.08; tiempo de ejecución 8 ÷ 3.08 ≈ 2.6 h |
| **N = 8** | 1 ÷ (0.10 + 0.1125) = 1 ÷ 0.2125 ≈ 4.71; tiempo de ejecución ≈ 1.70 h |
| **N = 16** | 1 ÷ (0.10 + 0.05625) = 1 ÷ 0.15625 = 6.4; tiempo de ejecución = 1.25 h |
| **N más pequeño que cumple el objetivo** | Se necesita 0.10 + 0.90 ÷ N ≤ 1 ÷ 5.33 = 0.1875, así que N ≥ 0.90 ÷ 0.0875 ≈ 10.3: 11 servidores dan 1 ÷ (0.10 + 0.0818) ≈ 5.50, tiempo de ejecución ≈ 1.45 h |
| **N = 64** | 1 ÷ (0.10 + 0.0141) ≈ 8.77; tiempo de ejecución ≈ 0.91 h |
| **Límite cuando N crece** | 1 ÷ 0.10 = 10; el tiempo de ejecución nunca baja de 0.8 h |
| **Alternativa** | Reducir a la mitad la parte serie (s = 0.05), por ejemplo cargando las tarifas en paralelo. Entonces 8 servidores dan 1 ÷ (0.05 + 0.11875) ≈ 5.93, tiempo de ejecución ≈ 1.35 h: el objetivo se cumple con 8 servidores en lugar de 11. |

Pasar de 16 a 64 servidores (cuatro veces el costo) solo reduce la ejecución de 1.25 h a 0.91 h. Los rendimientos decrecen rápidamente.

Reducir la fracción serie suele ser mucho más barato que añadir hardware. Los arquitectos buscan las partes serie ocultas: un único bloqueo, una fila de la base de datos que todos actualizan, un coordinador central.

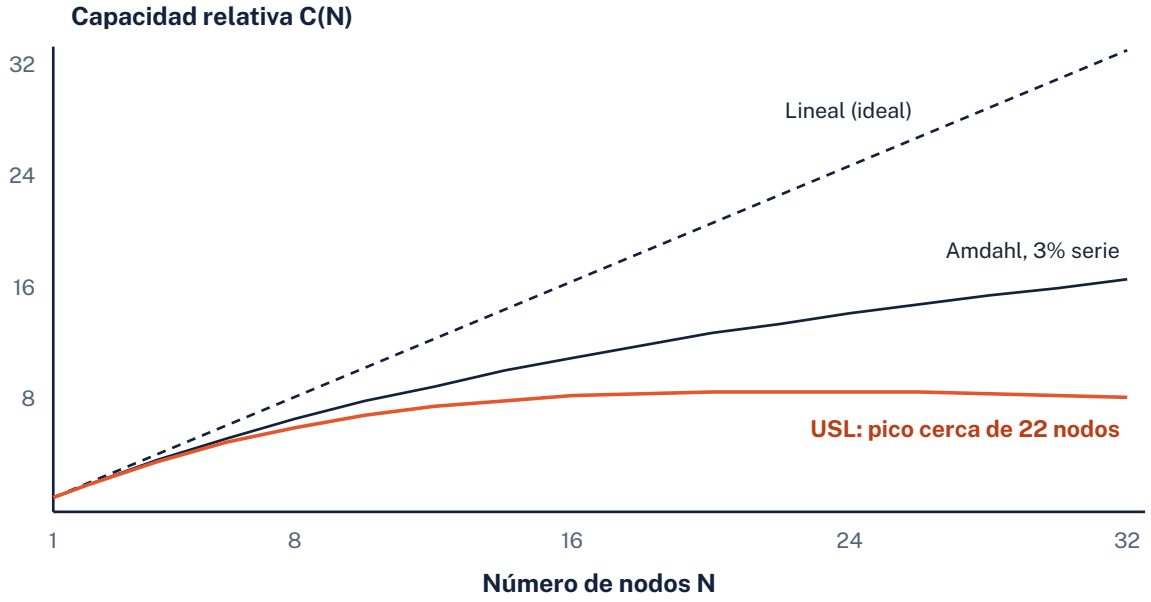
## La Ley de escalabilidad universal

La ley de Amdahl supone que los procesadores adicionales nunca empeoran las cosas; la aceleración simplemente se estanca. Los sistemas distribuidos reales a menudo sí empeoran: a partir de cierto punto, añadir nodos reduce el throughput. La Ley de escalabilidad universal (USL) de Neil Gunther lo modela con dos penalizaciones. La contención (α, alfa) es el costo de hacer cola por los recursos compartidos, el mismo efecto que la fracción serie de Amdahl. La

coherencia (β, beta) es el costo de mantener los datos consistentes entre nodos: cada nodo debe intercambiar información con todos los demás, así que este costo crece aproximadamente con el cuadrado del número de nodos.

La capacidad relativa es **C(N) = N ÷ (1 + α(N − 1) + βN(N − 1))**. Con β = 0 se reduce a la ley de Amdahl. Con cualquier β mayor que cero, la capacidad alcanza un máximo en torno a N = √((1 − α) ÷ β) y luego cae. Para α = 0.03 y β = 0.002, el máximo está cerca de √(0.97 ÷ 0.002) ≈ 22 nodos, con una capacidad de unas 8.6 veces la de un nodo; con 32 nodos la capacidad ha vuelto a bajar a unas 8.2. Gunther describe el modelo en *Guerrilla Capacity Planning* (2007). Su valor práctico es que α y β se pueden ajustar a partir de un puñado de mediciones de pruebas de carga, lo que te permite predecir dónde dejará de compensar el escalado antes de comprar el hardware.

**Escalado lineal, de Amdahl y de la USL**




El escalado ideal es lineal; la ley de Amdahl se aplana por el trabajo serie; la USL incluye además los costos de coherencia, así que la capacidad alcanza un máximo y luego disminuye.

## Tácticas de rendimiento: una visión general

Bass, Clements y Kazman agrupan las tácticas de rendimiento en dos familias. La primera es controlar la demanda de recursos: reducir cuánto trabajo llega o cuánto cuesta cada unidad. La segunda es gestionar los recursos: hacer que los recursos disponibles trabajen más o añadir más. La mayoría de los diseños reales combinan varias tácticas; cada una tiene también costos en otros ámbitos, y por eso son decisiones arquitectónicas.

### Tácticas de rendimiento y dónde las trata el curso

| Táctica | Qué hace | Costo o riesgo típico | Más en |
|---|---|---|---|
| **Reducir la demanda: limitación de tasa (rate limiting), control de admisión, salas de espera** | Limita o suaviza la carga entrante para que el núcleo se mantenga por debajo de su codo | Algunos usuarios sufren retrasos o son rechazados | Capítulo 9: colas y back pressure |
| **Reducir el trabajo por solicitud: algoritmos eficientes, cargas útiles más pequeñas, menos viajes de ida y vuelta** | Reduce el tiempo de servicio | Esfuerzo de desarrollo; a veces, código menos flexible | Capítulo 0: datos y algoritmos |
| **Priorizar y planificar** | Atiende primero el trabajo importante (los pedidos pagados antes que los informes) | El trabajo de baja prioridad puede quedarse sin atender | Capítulo 9: patrones de resiliencia |
| **Caché** | Mantiene los resultados cerca del usuario para evitar trabajo repetido | Datos desactualizados, complejidad de invalidación, costo de memoria | Capítulo 7: caché |
| **Concurrencia y asincronía** | Procesa en paralelo el trabajo independiente; saca el trabajo lento de la ruta de la solicitud | Razonamiento más difícil, problemas de orden y de consistencia | Capítulo 6: comunicación |
| **Replicación** | Añade copias de servicios o datos para repartir las lecturas y la carga | Consistencia entre copias; costo | Capítulo 7: replicación y particionamiento |
| **Particionamiento (sharding)** | Divide los datos y el trabajo para que cada nodo se ocupe de una parte | Las consultas entre particiones y el reequilibrado son difíciles | Capítulo 7: replicación y particionamiento |
| **Añadir recursos: escalar vertical u horizontalmente** | Aumenta la capacidad | Costo; limitado por el trabajo serie y de coherencia | Capítulo 9: estrategias de escalado |

**ATENCIÓN**

### Errores comunes

- Especificar o monitorizar medias en lugar de percentiles, y promediar percentiles entre servidores.
- Hacer pruebas de carga a un ritmo constante cuando el tráfico real llega en ráfagas; las colas se forman en las ráfagas.
- Planificar el funcionamiento de un recurso crítico al 90% de utilización o más, sin dejar margen para los picos.
- Suponer que más servidores siempre significan más capacidad, sin medir los costos de serie y de coherencia.
- Optimizar el tiempo de servicio ignorando el tiempo de espera, que suele ser la parte mayor con carga.
- Medir solo en el lado del servidor, cuando los usuarios también perciben el tiempo de red y el renderizado en el cliente.

**COMPRUEBA LO QUE HAS ENTENDIDO**

**▼ Un servicio recibe 500 solicitudes por segundo y cada una tarda 80 ms. ¿Cuántas hay en curso en promedio?**

Por la ley de Little, L = 500 × 0.08 = 40 solicitudes en curso.

**▼ Con la fórmula M/M/1, ¿cuál es el tiempo de respuesta medio de un servidor con un tiempo de servicio de 10 ms al 75% de utilización?**

W = 10 ÷ (1 − 0.75) = 40 ms, de los cuales 30 ms son de espera.

**▼ ¿Por qué p99 suele ser una medida mejor que la media para un requisito?**

Las distribuciones de los tiempos de respuesta son asimétricas; la media puede parecer saludable mientras una parte significativa de los usuarios sufre respuestas muy lentas, y p99 deja al descubierto esa cola.

**▼ Si el 20% de un trabajo es serie, ¿cuál es la aceleración máxima que se obtiene añadiendo procesadores?**

1 ÷ 0.2 = 5 veces, por muchos procesadores que se añadan.

▼ **Inténtalo: Un servicio de pedidos atiende 2400 solicitudes por segundo, y cada una mantiene ocupada una conexión a la base de datos durante 30 ms. Tiene 8 instancias con 8 conexiones cada una. ¿Es suficiente? ¿Qué tamaño de pool fijarías por instancia, con un objetivo de uso del 70%?**

Por la ley de Little, 2400 x 0.030 = 72 conexiones ocupadas de media, y solo existen 8 x 8 = 64. Así que las solicitudes hacen cola para obtener conexiones y el tiempo de espera se dispara en el pico. Para un uso del 70% necesitas 72 / 0.7 = unas 103 conexiones, o 103 / 8 = unas 13 por instancia, 104 en total. Comprueba que la base de datos pueda manejar tantas, y reduce los 30 ms si no puede.

---

**FUENTES**


- Dean and Barroso, The Tail at Scale (Google Research): https://research.google/pubs/pub40801/
- Neil Gunther: How to quantify scalability (the USL): https://www.perfdynamics.com/Manifesto/USLscalability.html
- Google SRE book: Monitoring distributed systems (latency and percentiles): https://sre.google/sre-book/monitoring-distributed-systems/
- Brendan Gregg: The USE method: https://brendangregg.com/usemethod.html

SECCIÓN 2.5 DE 2.8 · INTERMEDIO · 22 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

# Disponibilidad y fiabilidad: nueves, MTBF, MTTR y tolerancia a fallos

## ANTES DE LEER ESTO

Esta sección se basa en:

- 0.7 Servidores y la nube
- 2.2 Modelos de calidad
- 2.3 Escenarios de calidad

Términos anteriores que utiliza: Servidor, Balanceador de carga, Componente, Base de datos, Centro de datos

## LA IDEA EN BREVE

Piensa en una tienda de barrio. La disponibilidad es la frecuencia con la que la puerta está abierta cuando llegas. La fiabilidad es si, una vez dentro, la caja registradora funciona durante toda tu compra. Las tiendas se mantienen abiertas teniendo una caja de repuesto, un segundo empleado y una forma rápida de detectar y resolver los problemas. Con el software ocurre lo mismo: los arquitectos deciden cuánta capacidad de reserva mantener, cómo detectar rápidamente los fallos y con qué rapidez recuperarse, y expresan el objetivo como un número, por ejemplo el 99.9% del tiempo.

## LO QUE NECESITAS SABER

- La disponibilidad es la fracción del tiempo o de las solicitudes en la que el servicio está listo; la fiabilidad es la probabilidad de funcionar sin fallos durante un periodo determinado; son cosas distintas y un sistema puede puntuar bien en una y mal en la otra.
- Un defecto es una causa, un error es un estado interno incorrecto y un fallo es una desviación que los usuarios pueden observar; las tácticas previenen los defectos, detectan los errores o impiden que los errores se conviertan en fallos.
- Tiempo de inactividad permitido = (1 - disponibilidad) x periodo: el 99.9% equivale a unas 8.76 horas al año o 43.2 minutos al mes; el 99.99%, a unos 52.6 minutos al año.
- A = MTBF / (MTBF + MTTR), así que reducir el tiempo de reparación mediante automatización y redundancia suele ser más barato y más eficaz que hacer que los fallos sean menos frecuentes.

- En serie, las disponibilidades se multiplican; en paralelo, se multiplican las indisponibilidades, pero solo si las copias fallan de forma independiente y el failover funciona.
- El RTO (cuánto se tarda en restaurar) y el RPO (cuántos datos puedes perder) los fija el negocio, y determinan el diseño y el costo de la recuperación ante desastres.
- Los SLI miden, los SLO fijan objetivos internos y los SLA son contratos; un SLO convierte la disponibilidad en un presupuesto de errores que los equipos pueden gastar en cambios.

## Cuatro palabras que a menudo se confunden

Los usuarios perciben un sistema como algo que funciona o que no funciona. Los arquitectos necesitan palabras más precisas, porque cada palabra apunta a una respuesta de diseño distinta. La disponibilidad es la proporción del tiempo (o de las solicitudes) en la que el sistema está listo para prestar su servicio cuando se le pide. La fiabilidad es la probabilidad de que el sistema funcione correctamente, sin fallos, durante un periodo determinado y en unas condiciones determinadas. La tolerancia a fallos es la capacidad de seguir prestando el servicio aunque alguna parte haya fallado. La resiliencia es la capacidad más amplia de absorber una perturbación, mantener en marcha el servicio esencial (quizá degradado) y volver después a la normalidad.

ISO/IEC 25010:2023 sitúa la disponibilidad, la tolerancia a fallos y la recuperabilidad, junto con la ausencia de defectos (faultlessness, que sustituyó al término madurez de 2011), como subcaracterísticas de la fiabilidad. Bass, Clements y Kazman, en *Software Architecture in Practice*, tratan la disponibilidad como un atributo de calidad propio porque impulsa muchísimas decisiones arquitectónicas. Ninguna de las dos visiones es incorrecta; usa la que ayude a la conversación, pero deja claro a qué te refieres con cada palabra.

La diferencia entre disponibilidad y fiabilidad importa en la práctica. Imagina un servicio que se cae una vez por hora y se reinicia en un segundo. Su disponibilidad es 3599 / 3600 = 99.97%, lo que parece excelente. Su fiabilidad durante un trabajo por lotes de dos horas es mala: si los fallos llegan al azar a un ritmo de uno por hora, la probabilidad de funcionar dos horas sin ninguno es $e^{-2}$, alrededor del 13.5%. A un terminal de pago le importa la disponibilidad; a un cálculo científico de larga duración, a un robot quirúrgico o a un avión les importa la fiabilidad durante toda la misión.

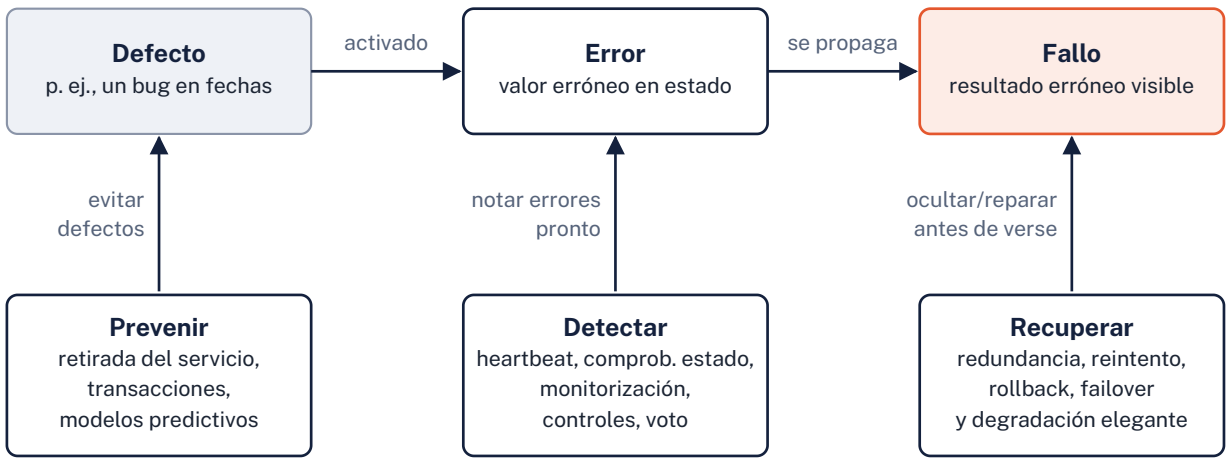
## Defecto, error y fallo

La comunidad de la confiabilidad (Avižienis, Laprie, Randell y Landwehr, 2004) separa tres ideas que el lenguaje cotidiano confunde. Un defecto es la causa: un bug, un disco averiado, una configuración errónea, un certificado caducado. Un defecto puede permanecer latente

durante años. Cuando se activa, produce un error: una parte del estado interno del sistema que es incorrecta, como un registro dañado o un valor incorrecto en memoria. Si el error llega al límite del servicio, el resultado es un fallo: el servicio que se presta a los usuarios se desvía de lo que debería ser.

Bass, Clements y Kazman usan la misma idea de forma más sencilla: un fallo es una desviación de la especificación que los usuarios del sistema pueden observar, y la causa de un fallo es un defecto. Lo importante para los arquitectos es que cada eslabón de la cadena es un lugar donde intervenir. Puedes impedir que surjan defectos (revisión de código, retirar un componente del servicio antes de que se desgaste), detectar los errores antes de que se propaguen (validación, comprobaciones de estado) y contener o enmascarar los errores para que nunca se conviertan en fallos (redundancia, reintentos, transacciones).

**La cadena y dónde actúan las tácticas**




Familias de tácticas de Bass, Clements y Kazman, Software Architecture in Practice (4.ª edición, 2021)

Las tácticas de prevención impiden que surjan defectos, las de detección advierten los errores y las de recuperación impiden que los errores se conviertan en fallos visibles.

## Contar los nueves

Los objetivos de disponibilidad suelen escribirse como un porcentaje con un número de nueves: el 99.9% son tres nueves. Conviene convertir cada objetivo en tiempo de inactividad permitido, porque un porcentaje oculta el poco tiempo del que dispones realmente. El cálculo es sencillo: inactividad = (1 - disponibilidad) x periodo. Un año de 365 días tiene 525 600 minutos, así que el 99.9% permite 0.001 x 525 600 = 525.6 minutos, unas 8.76 horas. Para un mes de 30 días (43 200 minutos) permite 43.2 minutos.

Cada nueve adicional divide el margen entre diez, mientras que el costo de ingeniería suele crecer mucho más deprisa. Con el 99.99% tienes unos 4.3 minutos al mes: menos de lo que tarda una persona en recibir el aviso, despertarse, iniciar sesión y entender qué falla. Por eso, por encima de tres o cuatro nueves, la recuperación debe ser automática; las personas son demasiado lentas para formar parte del proceso.

**Tiempo de inactividad permitido para objetivos de disponibilidad habituales (año de 365 días, mes de 30 días)**

| Disponibilidad | Por año | Por mes | Por semana | Por día |
|---|---|---|---|---|
| **99% (dos nueves)** | 3.65 días (87.6 h) | 7.2 horas | 1.68 horas (100.8 min) | 14.4 minutos |
| **99.9% (tres nueves)** | 8.76 horas | 43.2 minutos | 10.1 minutos | 1.44 minutos |
| **99.95%** | 4.38 horas | 21.6 minutos | 5.04 minutos | 43.2 segundos |
| **99.99% (cuatro nueves)** | 52.6 minutos | 4.32 minutos | 1.01 minutos | 8.64 segundos |
| **99.999% (cinco nueves)** | 5.26 minutos | 25.9 segundos | 6.05 segundos | 0.86 segundos |

**ATENCIÓN**

### ¿Basada en el tiempo o en las solicitudes?

La tabla supone que la disponibilidad se mide como tiempo de actividad sobre el tiempo de reloj. Muchos servicios en línea la miden en cambio como la proporción de solicitudes con éxito: 999 000 respuestas correctas de 1 000 000 es un 99.9%. La visión basada en solicitudes trata con naturalidad las caídas parciales (una región caída, o que solo falle el proceso de pago) y da más peso a las horas de mucha actividad que a las tranquilas. Elijas lo que elijas, anota exactamente qué cuenta como éxito (¿códigos de estado, un límite de latencia?), dónde se mide (en el balanceador de carga o en el navegador del usuario) y en qué ventana de tiempo. Dos equipos pueden afirmar ambos un 99.9% y referirse a cosas muy distintas.

## MTBF, MTTF y MTTR

Tres promedios relacionan los fallos y las reparaciones. El tiempo medio hasta el fallo (MTTF) es el tiempo medio que funciona un componente antes de fallar. El tiempo medio de reparación (MTTR) es el tiempo medio desde el fallo hasta que se restablece el servicio (algunas organizaciones leen la R como recover o restore, es decir, recuperar o restablecer; la idea es la misma, pero di qué reloj usas: desde el fallo, desde la detección o desde la primera alerta). El tiempo medio entre fallos (MTBF) es el tiempo medio desde un fallo hasta el siguiente.

La fórmula clásica de estado estacionario es **A = MTBF / (MTBF + MTTR)**. En rigor, para un sistema reparable MTBF = MTTF + MTTR, y la expresión exacta es MTTF / (MTTF + MTTR): el tiempo de actividad dividido entre el tiempo de actividad más el de inactividad. Los libros de texto escriben a menudo MTBF cuando se refieren al tiempo medio de actividad; cuando las reparaciones son breves en comparación con el tiempo entre fallos, como suele ocurrir, la diferencia es despreciable. La fórmula supone un promedio a largo plazo y no dice nada sobre cómo se distribuyen las caídas, por lo que combina bien con una medida de fiabilidad y con percentiles del tiempo de reparación.

EJEMPLO PRÁCTICO

### Ejemplo práctico: ¿es más barato fallar menos a menudo o recuperarse más rápido?

Un servicio de pedidos de una app de reparto de comida falla de media una vez cada 30 días (MTBF = 720 horas). Cada fallo tarda 4 horas en repararse, porque hay que avisar a un ingeniero, que debe diagnosticar el problema y restaurar una base de datos a mano. El negocio quiere superar el 99.9%.

| | |
|---|---|
| **Disponibilidad actual** | 720 / (720 + 4) = 720 / 724 = 0.99448, es decir, 99.45% |
| **Tiempo de inactividad actual por año** | (1 - 0.99448) x 8760 h = unas 48.4 horas |
| **Opción 1: duplicar el MTBF** | Mejor hardware y más pruebas; fallos cada 60 días. A = 1440 / 1444 = 0.99723 (99.72%), unas 24.3 horas al año |
| **Opción 2: reducir el MTTR a 15 minutos** | Detección automatizada, una reserva en caliente (hot standby) y failover mediante scripts. A = 720 / 720.25 = 0.99965 (99.97%), unas 3.0 horas al año |
| **Comprobación del objetivo** | Solo la opción 2 supera el 99.9% (8.76 horas al año) |

Reducir a la mitad la frecuencia de los fallos redujo a la mitad el tiempo de inactividad. Reducir el tiempo de reparación en un factor de 16 redujo el tiempo de inactividad aproximadamente en el mismo factor, y superó el objetivo.

Aumentar el MTBF es luchar contra la física y la complejidad: no puedes hacer perfectos los discos, las redes ni las personas. Reducir el MTTR está en gran parte bajo el control del arquitecto: detección, automatización, redundancia, despliegues pequeños y reversibles. Por eso la práctica moderna de operaciones se centra en la recuperación.

El MTTR es un promedio y oculta las colas largas. Una caída de ocho horas entre muchas de cinco minutos domina el año, así que sigue la distribución de la duración de los incidentes, no solo la media.

## Componentes en serie y en paralelo

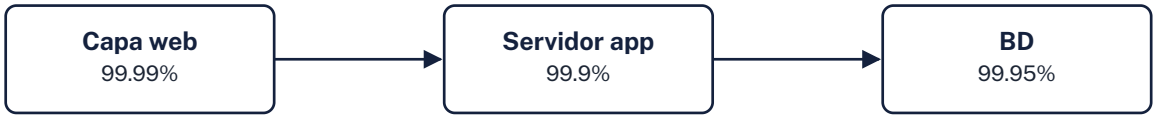
Los sistemas reales son cadenas de piezas. Si una solicitud necesita la capa web, el servidor de aplicaciones y la base de datos, los tres deben funcionar: los componentes están *en serie*, y sus disponibilidades se multiplican. Como cada factor es menor que uno, el producto siempre es

inferior al del eslabón más débil. Diez componentes al 99.9% cada uno dan $0.999^{10} = 0.990$, solo un 99.0%.

Si, en cambio, basta con una cualquiera de varias copias idénticas, las copias están *en paralelo* (redundancia). El conjunto solo falla cuando fallan todas las copias a la vez, así que se multiplican las indisponibilidades: $A_{paralelo} = 1 - (1 - A)^n$ para n copias idénticas. Dos servidores al 99% cada uno dan 1 - 0.01 x 0.01 = 99.99%.

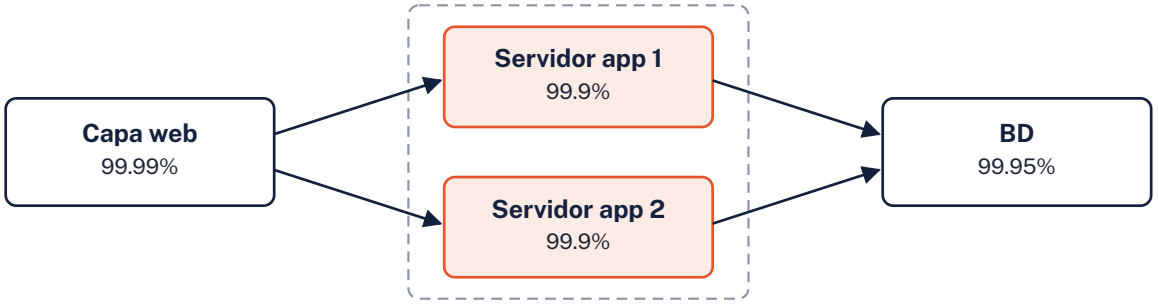
**Serie y paralelo con números**

**A. Tres componentes en serie: todos deben funcionar**




**0.9999 x 0.999 x 0.9995 = 0.9984 (99.84%, unas 14 horas de caída al año)**

**B. La misma cadena con dos servidores app en paralelo: basta con uno**




**par: 1 - (0.001 x 0.001) = 0.999999; cadena: 0.9994 (99.94%, unas 5.3 horas al año)**

Duplicar la etapa más débil lleva toda la cadena de unas 14 horas de inactividad al año a unas 5.3.

EJEMPLO PRÁCTICO

### Ejemplo práctico: ¿dónde conviene gastar la próxima libra?

Seguimos con el diagrama. La cadena de la parte A está en el 99.84%. ¿Qué cambio la acerca más al 99.99%?

| | |
|---|---|
| **Añadir un segundo servidor de aplicaciones** | 0.9999 x 0.999999 x 0.9995 = 0.99940 (99.94%): unas 5.3 horas al año |
| **Después, añadir una réplica de la base de datos con failover** | Par de bases de datos: $1 - 0.0005^2 = 0.99999975$. Cadena: 0.9999 x 0.999999 x 0.99999975 = 0.99990 (99.99%): unos 53 minutos al año |
| **Qué limita ahora el sistema** | La única capa web (99.99%) aporta casi todo el tiempo de inactividad restante |

La aritmética supone que las copias fallan de forma independiente. Dos servidores en el mismo rack, con la misma alimentación eléctrica, que ejecutan la misma versión con bugs o que dependen ambos de un único proveedor de DNS comparten defectos de modo común y pueden fallar juntos. Las ganancias reales siempre son menores de lo que sugiere la fórmula.

También supone que el failover es instantáneo y siempre funciona. Una reserva que nunca se ha probado en un failover real es una esperanza, no un componente.

Una regla útil de los autores de SRE de Google (Treynor, Dahlin, Rau y Beyer, 2017): cada dependencia crítica debería ofrecer aproximadamente un nueve más que el servicio que depende de ella, o bien el servicio necesita una forma de arreglárselas sin ella.

## Tácticas de disponibilidad: detectar, recuperar, prevenir

Bass, Clements y Kazman agrupan las tácticas de disponibilidad en tres familias. Son opciones de diseño, no una lista de comprobación que haya que aplicar entera; cada una tiene un costo.

**Las tres familias de tácticas de disponibilidad**

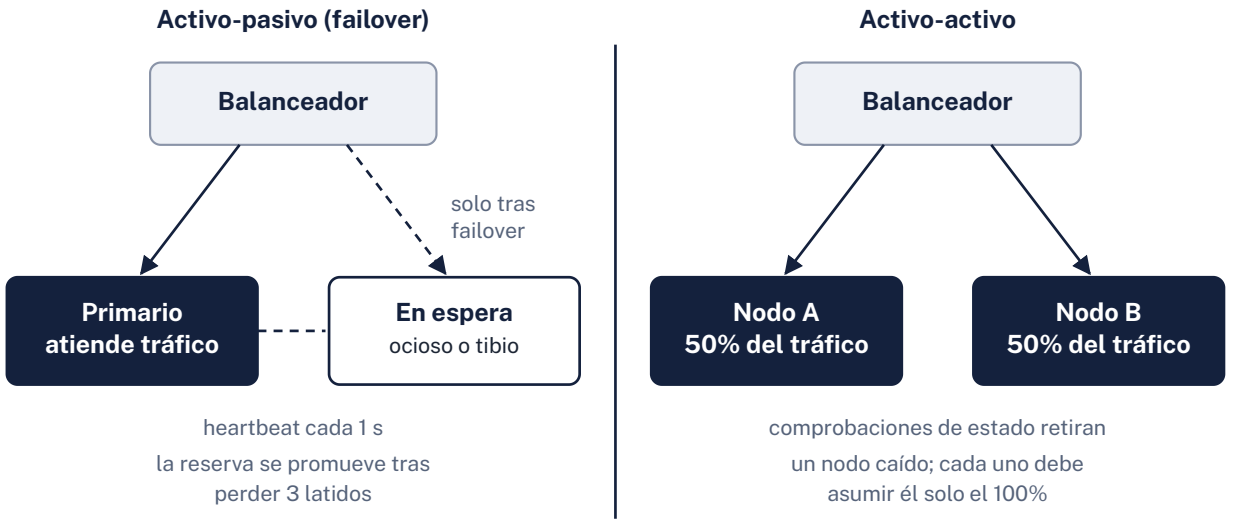
1. **Detectar defectos**
   Heartbeat (latido)
   : un componente envía periódicamente un mensaje de "estoy vivo", y el silencio desencadena una acción. Ping/eco: un monitor pregunta y espera una respuesta. Comprobaciones de estado (health checks): un endpoint como `/health` que un balanceador de carga o un orquestador consulta periódicamente (una comprobación de actividad, liveness, pregunta "¿se está ejecutando el proceso?"; una comprobación de preparación, readiness, pregunta "¿puede atender tráfico ahora?"). Monitorización de las tasas de error y de la latencia, comprobaciones de coherencia de los valores y votación entre réplicas que calculan el mismo resultado.

2. **Recuperar: preparación y reparación**
   Reservas redundantes en tres temperaturas: *activo-activo* (todas las copias atienden tráfico real), *hot standby o warm standby* (reserva activa: una copia que se mantiene sincronizada y lista) y *reserva en frío* (cold spare: solo se arranca cuando hace falta). El failover (conmutación por error)
   desvía el tráfico a una copia en buen estado. El reintento gestiona los defectos transitorios (con límites y back-off, o los reintentos se convierten en una sobrecarga autoinfligida). El rollback vuelve a un estado o a una versión que se sabe que funciona. La degradación elegante
   mantiene en marcha el servicio principal sin una función que ha fallado: la tienda sigue aceptando pedidos aunque las recomendaciones no funcionen.

3. **Recuperar: reintroducción**
   Volver a poner en servicio de forma segura un componente reparado: ejecutarlo en modo sombra, resincronizar su estado y reiniciar por pasos escalonados (un hilo, luego un proceso, luego una máquina).

4. **Prevenir defectos**
   Retirada del servicio (sacar un nodo de la rotación para mantenimiento o para un reinicio programado antes de que una fuga de memoria conocida cause problemas), transacciones (actualizaciones de todo o nada para que una caída no pueda dejar un estado escrito a medias), modelos predictivos (sustituir un disco cuando suben sus contadores de errores) y prevención de excepciones (tipos más estrictos, validación de entradas).

**Dos formas de redundancia**




Activo-pasivo paga por capacidad ociosa y por un intervalo de failover; activo-activo usa toda la capacidad, pero cada nodo debe sobrevivir a la pérdida del otro.

## Elegir entre activo-pasivo y activo-activo

Activo-pasivo es más fácil de razonar, sobre todo para componentes con estado, como una base de datos relacional con un único primario en el que se puede escribir. Sus costos son la reserva ociosa, un intervalo de failover (detección más promoción, a menudo decenas de segundos) y el riesgo de que la reserva se haya desincronizado sin que nadie lo note. Una trampa clásica es el *split brain* (cerebro dividido): la red entre los dos se rompe, cada uno cree que el otro ha muerto y ambos aceptan escrituras. Protegerse contra ello requiere un quórum o un árbitro externo.

Activo-activo usa toda la capacidad y no tiene paso de promoción, pero cada nodo debe dimensionarse para soportar toda la carga si su pareja falla. Dos nodos que funcionan cada uno al 70% de utilización recibirán una demanda del 140% cuando uno muera, y el superviviente se hunde: un diseño redundante que falla de todos modos. Activo-activo también te obliga a resolver la consistencia de los datos entre las copias, que es un gran tema en sí mismo (consulta consistencia, CAP y PACELC).

## Recuperación ante desastres: RTO y RPO

La redundancia dentro de un centro de datos no te protege de perder todo el centro de datos, una región, una cuenta en la nube o tus datos por un ransomware. La planificación de la recuperación ante desastres usa dos números que deberían venir del negocio, no de los ingenieros. El objetivo de tiempo de recuperación (RTO) es el tiempo máximo aceptable para

restaurar el servicio después de un desastre. El objetivo de punto de recuperación (RPO) es la mayor pérdida de datos aceptable, medida en tiempo: un RPO de 15 minutos significa que puedes perder como máximo los cambios de los últimos 15 minutos.

Los dos números llevan a diseños y costos muy distintos. Las copias de seguridad nocturnas copiadas a otra región dan un RPO de hasta 24 horas y un RTO de muchas horas (restaurar terabytes lleva tiempo). La replicación asíncrona continua a un warm standby (espera semiactiva) en otra región podría dar un RPO de segundos y un RTO de minutos, quizá con el doble de costo de infraestructura. La replicación síncrona puede acercar el RPO a cero, pero añade a cada escritura el tiempo de ida y vuelta entre sedes. Los proveedores de nube describen una escalera de estrategias: copia de seguridad y restauración, pilot light (piloto encendido), warm standby y activo-activo multisitio, cada una más rápida y más cara que la anterior. Los detalles corresponden a multirregión y recuperación ante desastres.

Sea cual sea la estrategia, una copia de seguridad que nunca has restaurado no es una copia de seguridad. Prueba las restauraciones de forma programada y mide cuánto tardan; el tiempo de restauración medido es tu RTO real.

## De los objetivos a los SLI, SLO y SLA

La práctica moderna de operaciones expresa la disponibilidad con tres términos relacionados, que se tratan a fondo en SLI, SLO y presupuestos de errores. Un indicador de nivel de servicio (SLI) es la medición, por ejemplo la proporción de solicitudes de pago que tienen éxito en menos de 800 ms. Un objetivo de nivel de servicio (SLO) es el objetivo interno para esa medición en una ventana de tiempo, por ejemplo el 99.9% en 30 días. Un acuerdo de nivel de servicio (SLA) es un contrato con los clientes que establece consecuencias, como créditos de servicio, si no se alcanza un nivel; normalmente se fija más laxo que el SLO para que el equipo reciba un aviso antes de que se incumpla el contrato.

El SLO convierte la disponibilidad en un presupuesto. Un SLO mensual del 99.9% permite que falle el 0.1% de las solicitudes; con 2 millones de solicitudes al mes, son 2000 solicitudes fallidas, o unos 43 minutos de caída total. Los equipos gastan este presupuesto de errores en riesgo: lanzamientos, experimentos, migraciones. Cuando se agota, frenan e invierten en fiabilidad. Esto conecta los nueves abstractos de esta sección con las decisiones del día a día.

> **SI ESTO ES NUEVO PARA TI**
>
> ### Si esto es nuevo para ti
>
> No necesitas memorizar la tabla de los nueves. Recuerda el método: uno menos el porcentaje, por el periodo. El 99.9% deja el 0.1% de un mes de 30 días, que es 0.001 x 43 200 minutos = 43.2 minutos. Todas las demás filas son la misma cuenta.

## Errores comunes

Prometer nueves sin contar las dependencias: un servicio del 99.99% construido sobre una única base de datos del 99.9% no puede cumplir su objetivo. Añadir redundancia sin eliminar las causas compartidas (la misma versión, el mismo cambio de configuración, el mismo certificado que caduca en todos los nodos a medianoche). Medir la disponibilidad desde dentro del sistema, donde todo se ve en verde, mientras los usuarios no pueden acceder por culpa del DNS o de una red de distribución de contenidos. Planificar el failover pero no ensayarlo nunca; la ingeniería del caos existe precisamente para ejercitar estos caminos de forma deliberada (ingeniería del caos). Por último, perseguir un número por sí mismo: un sistema de turnos de personal que se usa en horario de oficina no necesita cinco nueves, y el dinero se gasta mejor en otra cosa.

Para que la disponibilidad se pueda probar, escríbela como un escenario (Sección 2.3): “Cuando el servidor primario de la base de datos falla en las horas de máxima demanda, el sistema lo detecta en menos de 10 segundos y vuelve a aceptar pedidos en menos de 60 segundos, sin perder ningún pedido confirmado.” Cada parte de esa frase se convierte en una decisión de diseño y en una prueba.

**COMPRUEBA LO QUE HAS ENTENDIDO**

▼ **¿Cuántos minutos de inactividad permite un objetivo del 99.95% en un mes de 30 días?**

0.0005 x 43 200 = 21.6 minutos.

▼ **Dos balanceadores de carga independientes tienen cada uno un 99.5% de disponibilidad y cualquiera de los dos puede soportar todo el tráfico. ¿Cuál es la disponibilidad del par?**

1 - (0.005 x 0.005) = 1 - 0.000025 = 99.9975%.

▼ **¿Por qué añadir un segundo servidor idéntico podría aportar mucho menos beneficio del que predice la fórmula en paralelo?**

Porque las copias pueden compartir causas comunes de fallo, como la misma versión, la configuración, la alimentación eléctrica, la red o una dependencia, así que no fallan de forma independiente; además, el propio failover puede ser lento o no funcionar.

▼ **Las copias de seguridad de un equipo se ejecutan cada 6 horas y una restauración completa tarda 3 horas. ¿Cuáles son su RPO y su RTO efectivos en el peor caso?**

Un RPO de hasta 6 horas de datos perdidos y un RTO de al menos 3 horas, más el tiempo necesario para detectar el desastre y decidir restaurar.

▼ **Inténtalo: Una cadena pasa por el gateway 99.95%, el servicio de pedidos 99.9% y el de pagos 99.9%. ¿Cuál es su disponibilidad y su tiempo de inactividad anual? Luego añade una segunda instancia de pedidos, independiente, con un failover que funcione. Recalcula y nombra la suposición que podría estropear el resultado.**

En serie: 0.9995 x 0.999 x 0.999 = 0.99750, alrededor del 99.75%, o 21.9 horas al año. Dos instancias de pedidos dan 1 - 0.001 x 0.001 = 0.999999, de modo que la cadena es 0.9995 x 0.999999 x 0.999 = 0.99850, unas 13.1 horas. El pago y la pasarela pasan a dominar. La suposición es la independencia: un mal lanzamiento compartido, una alimentación eléctrica o una base de datos comunes hacen que ambas copias fallen a la vez, y la conmutación por error debe funcionar de verdad.

**FUENTES**


- Google SRE book: Embracing Risk (time-based and request-based availability): https://sre.google/sre-book/embracing-risk/
- Google SRE book: Service Level Objectives: https://sre.google/sre-book/service-level-objectives/
- SEI blog: Tactics and Patterns for Software Robustness (availability tactics): https://www.sei.cmu.edu/blog/tactics-and-patterns-for-software-robustness/
- AWS whitepaper: Disaster recovery options in the cloud: https://docs.aws.amazon.com/whitepapers/latest/disaster-recovery-workloads-on-aws/disaster-recovery-options-in-the-cloud.html
- Avižienis, Laprie, Randell and Landwehr: Basic Concepts and Taxonomy of Dependable and Secure Computing (IEEE TDSC, 2004): https://doi.org/10.1109/TDSC.2004.2

SECCIÓN 2.6 DE 2.8 · INTERMEDIO · 22 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

# Modificabilidad, capacidad de prueba y desplegabilidad: diseñar para el cambio

### ANTES DE LEER ESTO

Esta sección se basa en:

- 0.3 Organizar el código
- 0.8 Cómo se hace el software
- 2.3 Escenarios de calidad

Términos anteriores que utiliza: Feature flag, Interfaz, Módulo, Prueba unitaria, Producción

### LA IDEA EN BREVE

Piensa en una cocina. Si el horno está empotrado en la pared con las tuberías enterradas en hormigón, cambiarlo supone derribar la pared. Si se conecta a un enchufe estándar, lo cambias en una tarde. Con el software ocurre lo mismo: algunas partes cambiarán con toda seguridad, así que los arquitectos les dan conexiones estándar, facilitan comprobar que todo sigue funcionando después de un cambio y hacen que entregar la nueva versión sea una rutina y no un acontecimiento arriesgado.

### LO QUE NECESITAS SABER

- La modificabilidad es el costo y el riesgo del cambio; especifícala diciendo qué es probable que cambie, quién hace el cambio, cuándo se vincula y cuánto puede costar el cambio.
- Las tácticas de modificabilidad aumentan la cohesión, reducen el acoplamiento (encapsular, usar un intermediario, restringir las dependencias) y difieren la vinculación, pero cada momento de vinculación más tardío añade maquinaria.
- La capacidad de prueba depende de la controlabilidad y la observabilidad; aísla las dependencias detrás de interfaces e inyéctalas para que las pruebas puedan usar dobles de prueba.
- La desplegabilidad se basa en lotes pequeños, automatización, unidades desplegables de forma independiente, cambios compatibles con versiones anteriores, despliegues progresivos y un rollback rápido.
- Las métricas actuales de entrega de software de DORA son el tiempo de entrega de cambios, la frecuencia de despliegue, el tiempo de recuperación de despliegues fallidos, la tasa de fallos de cambios y la tasa de retrabajo de despliegues.

- La mantenibilidad de ISO/IEC 25010:2023 tiene cinco subcaracterísticas: modularidad, reusabilidad, analizabilidad, modificabilidad y capacidad de prueba.

## Tres cualidades relacionadas con el cambio

La mayor parte del costo de un sistema de éxito se produce después de su primer lanzamiento, a medida que se cambia una y otra vez. Tres cualidades relacionadas describen lo bien que una arquitectura lo facilita. La modificabilidad trata del costo y el riesgo de hacer un cambio. La capacidad de prueba (testabilidad) trata de la facilidad con la que puedes demostrar, mediante pruebas, que el sistema (o una parte de él) se comporta como se pretende, y de la facilidad con la que los defectos se ponen de manifiesto. La desplegabilidad trata de la rapidez, la seguridad y la previsibilidad con que un cambio puede ponerse en producción, y retirarse de nuevo.

Están unidas en un ciclo. Un cambio solo es barato si puedes hacerlo en un único lugar (modificabilidad), saber rápidamente que no has roto nada (capacidad de prueba) y ponerlo en producción sin ceremonias (desplegabilidad). Un sistema modular pero sin pruebas rápidas seguirá cambiando despacio, porque cada cambio necesita semanas de pruebas de regresión manuales. Un sistema con buenas pruebas pero con una ventana de lanzamiento trimestral seguirá entregando valor despacio.

## Modificabilidad: cuatro preguntas sobre cualquier cambio

Bass, Clements y Kazman sostienen que la modificabilidad no tiene sentido hasta que dices qué se espera que cambie. Planificar para todos los cambios posibles produce un sistema sobrediseñado, lleno de puntos de extensión que no se usan; no planificar para ninguno produce un sistema en el que cada cambio es una reescritura. Cuatro preguntas precisan el requisito.

**¿Qué puede cambiar?** Funciones, plataformas (una nueva versión de la base de datos, el paso a otra nube), el entorno (una nueva API de un socio, una nueva normativa), las cualidades (el doble de carga) o la capacidad. **¿Qué probabilidad tiene el cambio?** Dedica el esfuerzo de diseño a donde el cambio es probable: los proveedores de pagos, las normas fiscales y las interfaces de usuario cambian a menudo; el concepto de pedido rara vez lo hace. **¿Quién hace el cambio y cuándo?** Un desarrollador al programar, un ingeniero de lanzamientos en el despliegue, un administrador mediante la configuración o un usuario final mediante los ajustes. El momento en que se fija una decisión se llama su momento de vinculación (binding time). **¿Cuánto cuesta el cambio?** El esfuerzo, el tiempo transcurrido y el dinero necesarios para hacerlo, probarlo y desplegarlo, más el número de otros elementos afectados y el riesgo de introducir defectos.

Responder a esas preguntas da un escenario del tipo presentado en la Sección 2.3: “Un desarrollador añade soporte para un nuevo proveedor de pagos, sin cambiar ningún módulo existente salvo la configuración, en 3 días-persona como máximo, incluidas las pruebas, sin aumento de los errores en el proceso de pago.”

**Escenarios de modificabilidad con distintas fuentes y momentos de vinculación**

| Quién cambia | Qué cambia | Cuándo se vincula | Medida de la respuesta |
|---|---|---|---|
| **Desarrollador** | Añadir un nuevo socio de reparto a una plataforma de reparto de comida | Al programar | Solo cambia el adaptador del socio; hecho, probado y desplegado en 5 días |
| **Responsable de producto** | Activar un nuevo flujo de pago para el 10% de los usuarios | En ejecución (feature flag) | Surte efecto en 1 minuto, sin despliegue |
| **Ingeniero de operaciones** | Apuntar el sistema a un nuevo host de base de datos | En el despliegue (configuración) | Sin cambios de código; una ejecución del pipeline de menos de 15 minutos |
| **Administrador del cliente** | Añadir un campo personalizado a una factura | En ejecución (ajustes) | Sin intervención del proveedor; disponible de inmediato |
| **Equipo de plataforma** | Actualizar el framework web a una nueva versión principal | En la compilación | Cambian menos de 20 archivos fuera de una capa anticorrupción; menos de 2 semanas |

## Tácticas de modificabilidad

Las tácticas de *Software Architecture in Practice* reducen el costo del cambio limitando hasta dónde se propaga. Se basan en dos ideas antiguas, tratadas a fondo en el Capítulo 3. La cohesión mide hasta qué punto las responsabilidades de un módulo van juntas; el acoplamiento mide cuánto depende un módulo de otro. Un cambio probable debería afectar a un único módulo cohesivo, y ese módulo debería estar débilmente acoplado al resto.

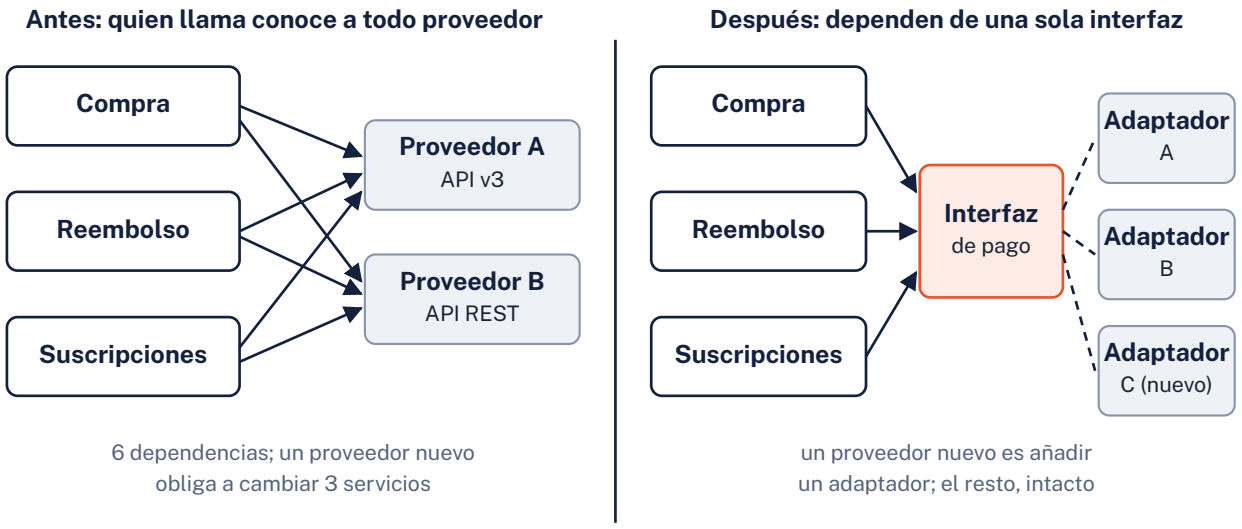
**Aumentar la cohesión.** Divide un módulo que hace dos tareas sin relación (un `CustomerService` que gestiona tanto el inicio de sesión como los puntos de fidelidad) y mueve las responsabilidades a donde corresponden, para que un cambio probable caiga en un solo lugar.

**Reducir el acoplamiento.** *Encapsular*: oculta el funcionamiento interno de un módulo detrás de una interfaz explícita, siguiendo el principio de ocultación de información de David Parnas (1972), para que quienes lo llaman no puedan depender de lo que puede cambiar. *Usar un*

*intermediario*: pon algo entre dos partes para que ninguna conozca directamente a la otra: un adaptador, un intermediario de mensajes (message broker), un canal de publicación-suscripción, un repositorio delante de una base de datos o un API gateway. *Restringir las dependencias*: decide qué módulos pueden llamar a cuáles (reglas de capas, un módulo visible solo para su propio paquete) y hazlo cumplir con herramientas. *Abstraer los servicios comunes*: proporciona el comportamiento compartido una sola vez, detrás de una interfaz, en lugar de copiarlo.

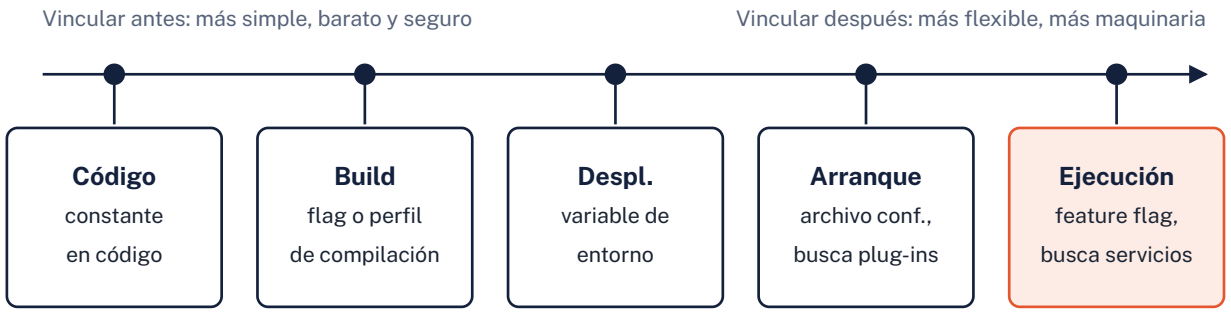
**Diferir la vinculación.** Fija una decisión tan tarde como sea útil, para que el cambio pueda hacerse más adelante y por alguien más alejado del código: archivos de configuración, variables de entorno, plug-ins descubiertos al arrancar, descubrimiento de servicios y feature flags. Cada momento de vinculación más tardío añade maquinaria (configuración que validar, flags que eliminar, interfaces de plug-in versionadas), así que difiere solo las decisiones que realmente varían.

**Usar un intermediario para contener el cambio**




Una interfaz con un adaptador por proveedor convierte un cambio que afectaba a tres servicios en la adición de un adaptador.

### Momento de vinculación (binding time)




Cada paso a la derecha permite cambiar más tarde, por alguien más lejos del código, sin un nuevo lanzamiento

Mover una decisión hacia la derecha permite cambiarla más tarde y sin un lanzamiento, pero cada paso añade maquinaria que gestionar.

GUÍA DE DECISIÓN

## Guía de decisión: ¿cuánto debería retrasarse la vinculación de esta decisión?

Úsala cuando una parte interesada diga "esto podría cambiar". Empieza por arriba y detente en el primer sí.

**Q1** **¿Es poco probable el cambio, o necesitaría código nuevo y pruebas nuevas de todos modos?**

- **Sí** Vincula al programar. Mantenlo simple y bien encapsulado para que el cambio siga siendo local cuando llegue.
- **No** Pasa a la siguiente pregunta.

**Q2** **¿El valor solo difiere entre entornos (pruebas, preproducción, producción) o entre las instalaciones de distintos clientes?**

- **Sí** Vincula en el despliegue: variables de entorno o configuración inyectada por el pipeline, guardadas en el control de versiones.
- **No** Pasa a la siguiente pregunta.

**Q3** **¿Necesitan terceros u otros equipos añadir comportamiento sin cambiar tu código?**

- **Sí** Vincula en el arranque o en ejecución con una interfaz de plug-ins o de extensión, y versiona esa interfaz con cuidado.
- **No** Pasa a la siguiente pregunta.

**Q4** **¿Deben personas que no son desarrolladores activar o desactivar un comportamiento rápidamente, o debes lanzar primero a una fracción de los usuarios?**

- **Sí** Vincula en ejecución con un feature flag, y da a cada flag un responsable y una fecha de eliminación.
- **No** Vincula en la compilación o en el despliegue; una vinculación más tardía añadiría maquinaria que nadie necesita.

EJEMPLO PRÁCTICO

## Ejemplo práctico: ¿cuánto vale el escenario de modificabilidad?

Una empresa de reparto de comida integra los sistemas de punto de venta de los restaurantes. Hoy, cada nuevo proveedor de punto de venta requiere cambios en los servicios de carta, pedidos y reembolsos: unos 15 días-persona de trabajo más un ciclo de pruebas de regresión de 5 días. Ventas prevé 8 nuevos proveedores en los próximos 12 meses. El arquitecto propone una interfaz de adaptadores para proveedores (como en el diagrama anterior). Supón un costo completo, con todos los gastos incluidos, de £600 por día-persona.

| | |
|---|---|
| **Costo por proveedor hoy** | 15 + 5 = 20 días-persona, así que 20 x £600 = £12 000 |
| **Costo de 8 proveedores hoy** | 8 x £12,000 = £96,000 |
| **Costo único de la refactorización** | Diseñar y construir la interfaz y migrar los 3 proveedores existentes: 30 días-persona = £18 000 |
| **Costo por proveedor después** | Escribir un adaptador (3 días) más pruebas de contrato contra la interfaz (1 día): 4 días-persona = £2400 |
| **Costo de 8 proveedores después** | £18,000 + 8 x £2,400 = £37,200 |
| **Ahorro en el primer año** | £96,000 - £37,200 = £58,800 |
| **Punto de equilibrio** | Cada proveedor ahorra £12 000 - £2400 = £9600, así que las £18 000 se recuperan tras £18 000 / £9600 = 1.9, es decir, con el segundo proveedor nuevo |

El cálculo solo es tan bueno como la predicción del cambio. Si ventas no consigue ningún proveedor nuevo, la refactorización es puro costo. Por eso los requisitos de modificabilidad deben decir qué es probable que cambie.

El plazo suele importar más que el costo: reducir la incorporación de un proveedor de unas 4 semanas a menos de 1 semana puede hacer ganar contratos, algo que ninguna suma de días-persona recoge.

La nueva interfaz de adaptadores es en sí misma una obligación. Cambiarla más adelante afecta a todos los adaptadores, así que merece un diseño y un versionado cuidadosos.

**SI ESTO ES NUEVO PARA TI**

### Paso a paso: decidir cuándo se fija una elección

La idea más difícil de esta sección es el momento de vinculación: el momento en que se fija una decisión, que determina quién puede cambiarla después y con qué facilidad. Sigue una decisión a través de tres momentos de vinculación, usando el diagrama de la sección, en el que el proceso de pago llega a los proveedores de pagos a través de una única interfaz Payment con un adaptador para cada proveedor.

- El Adaptador A y el Adaptador B conectan cada uno la interfaz Payment con un proveedor. Supón que la decisión es qué adaptador usa el proceso de pago.
- Al programar: un desarrollador escribe 'usar el Adaptador A' en el código. Es la opción más simple, pero cambiar a B supone un cambio de código, pruebas y un nuevo lanzamiento, y solo un desarrollador puede hacerlo.
- En el despliegue: la elección pasa a la configuración, por ejemplo una variable de entorno que proporciona el pipeline de despliegue. Ahora pruebas y producción pueden usar proveedores distintos, y un ingeniero de operaciones puede cambiar de proveedor con una ejecución del pipeline y sin cambiar código.
- En ejecución: un feature flag envía el 10% de los pagos a través del Adaptador B, y un responsable de producto puede subir o bajar esa proporción en un minuto, sin despliegue.
- Cada paso más tardío añade maquinaria. La configuración debe comprobarse para que un valor mal escrito no pueda detener los pagos; el flag necesita un responsable y una fecha de eliminación; y mientras cualquiera de los dos adaptadores pueda estar activo, ambos deben mantenerse probados.
- Añadir un nuevo Adaptador C sigue suponiendo código nuevo y pruebas nuevas, sea cual sea el momento de vinculación. Diferir la vinculación cambia cuándo, y quién, toma una decisión entre opciones que ya existen.
- **La lección:** pregunta quién necesita cambiar una decisión y con qué frecuencia, y elige el momento de vinculación más temprano que se lo permita. Vincular más tarde solo añade maquinaria que nadie usa.

## Capacidad de prueba: controlabilidad y observabilidad

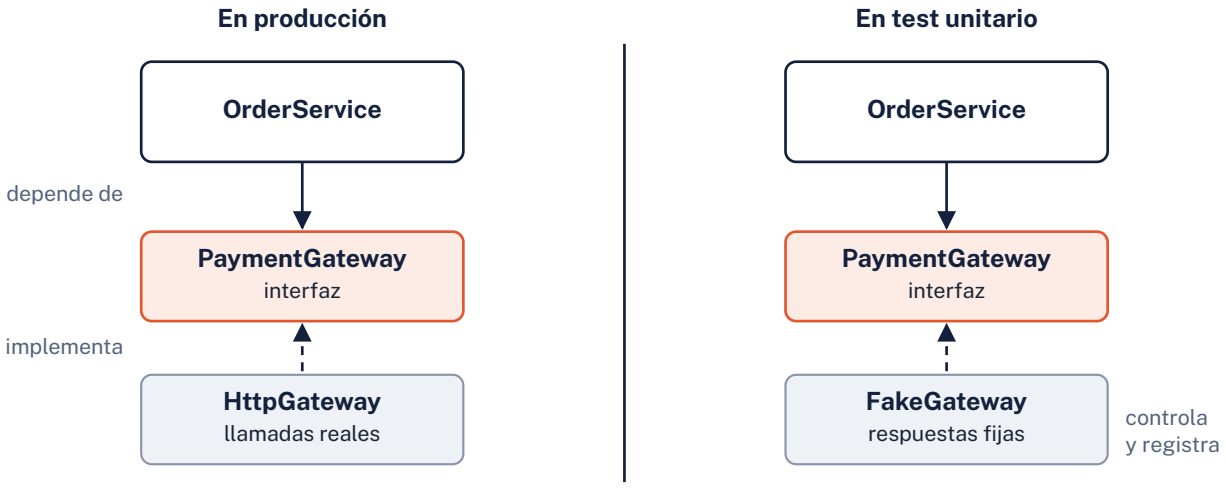
Un componente se puede probar cuando una prueba puede ponerlo en un estado conocido, darle entradas elegidas y ver qué hizo. Dos propiedades lo recogen. La **controlabilidad** es la capacidad de fijar las entradas y el estado interno de un componente desde una prueba. La **observabilidad** (en el sentido de las pruebas) es la capacidad de ver sus salidas y el estado interno relevante. El código que lee directamente el reloj del sistema, llama a un servicio de

pagos real, elige números aleatorios o escribe en una base de datos compartida es difícil de controlar; el código que oculta sus resultados en un estado privado o solo en el texto de los logs es difícil de observar.

Por eso, las tácticas arquitectónicas de capacidad de prueba buscan dar a las pruebas control y visibilidad, y limitar la complejidad. **Aísla las dependencias** detrás de interfaces y usa luego la inyección de dependencias: el componente recibe sus colaboradores (un reloj, una pasarela de pagos, un repositorio) desde fuera en lugar de crearlos, para que una prueba pueda pasarle un sustituto. Esos sustitutos son dobles de prueba, un término popularizado por Gerard Meszaros: un *stub* devuelve respuestas predefinidas, un *fake* es una implementación ligera que funciona (un repositorio en memoria), un *mock* comprueba que se hicieron las llamadas esperadas y un *spy* registra las llamadas para hacer aserciones después. **Proporciona interfaces de prueba**: ganchos para fijar el estado, grabar y reproducir tráfico o ejecutar en un entorno aislado (sandbox). **Limita el no determinismo**: evita la concurrencia oculta y el estado global, y haz que el tiempo y la aleatoriedad se puedan inyectar. **Limita la complejidad estructural**: menos dependencias y ningún ciclo significan preparaciones de prueba más pequeñas.

La arquitectura también da forma a la estrategia de pruebas en niveles superiores. En un sistema de servicios, cada límite necesita pruebas de contrato que comprueben que ambos lados están de acuerdo sobre la API, y unas pocas pruebas de extremo a extremo que recorran el camino real. Un diseño con límites limpios permite que la mayoría de las pruebas se ejecuten rápido y de forma aislada, mientras que un diseño enmarañado obliga a usar pruebas de extremo a extremo lentas y frágiles para todo.

**La inyección de dependencias hace controlable un componente**




El mismo OrderService funciona contra la pasarela real en producción y contra un fake con respuestas preparadas en las pruebas, porque solo depende de la interfaz.

```
class OrderService:
    def __init__(self, gateway, clock):
        self.gateway = gateway   # anything with charge(amount, card)
        self.clock = clock       # anything with now()

    def place(self, order):
        if self.clock.now().hour ⩾ 23:
            return "kitchen closed"
        return self.gateway.charge(order.total, order.card)

# In a test: full control, no network, no waiting until 11 pm
class FakeClock:
    def __init__(self, t): self.t = t
    def now(self): return self.t

class FakeGateway:
    def __init__(self): self.calls = []
    def charge(self, amount, card):
        self.calls.append(amount)
        return "approved"
```

Inyección por constructor: el servicio recibe su reloj y su pasarela, así que una prueba puede proporcionar fakes.

## Desplegabilidad

La desplegabilidad mide con qué previsibilidad y a qué costo pasa un cambio del commit de un desarrollador a producción, y con qué rapidez se puede deshacer. Se convirtió en una cualidad de primera clase solo en la última década, cuando la entrega continua demostró que la frecuencia de lanzamiento y la estabilidad no son opuestas. La cuarta edición de *Software Architecture in Practice* (2021) le dedicó un capítulo.

La idea central son los **lotes pequeños**. Un lanzamiento que contiene un solo cambio es fácil de probar, fácil de razonar y fácil de revertir; un lanzamiento con tres meses de trabajo contiene cientos de cambios que interactúan, así que los fallos son probables y difíciles de diagnosticar. Los lotes pequeños requieren **automatización**: un pipeline de despliegue que compila, prueba y despliega cada cambio de la misma manera, infraestructura mediante scripts y ningún paso manual que alguien pueda olvidar. Entre las tácticas arquitectónicas están las unidades desplegables de forma independiente (para que el lanzamiento de un equipo no espere al de otro), las interfaces y migraciones de base de datos compatibles con versiones anteriores, los mecanismos de rollback y de roll forward, los despliegues progresivos (lanzamientos canario primero a una pequeña parte del tráfico, conmutación azul-verde entre dos entornos) y los feature flags, que separan *desplegar* el código de *lanzar* una función a los usuarios. Los pipelines y las estrategias se tratan en CI/CD y estrategias de despliegue.

El costo de ignorar la desplegabilidad puede ser grave. En agosto de 2012, la empresa de trading Knight Capital desplegó manualmente código nuevo en sus ocho servidores de enrutamiento de órdenes, pero se olvidó de uno. Un flag de configuración reutilizado activó entonces código obsoleto en ese servidor, que inundó el mercado de órdenes, produjo unos 4 millones de ejecuciones en aproximadamente 45 minutos y le costó a la empresa unos 460 millones de dólares estadounidenses. Un despliegue automatizado y verificado, o la eliminación del código muerto, lo habrían evitado cada uno por sí solo.

## Medir la entrega: las métricas DORA

El programa de investigación DORA (DevOps Research and Assessment, descrito en *Accelerate* por Nicole Forsgren, Jez Humble y Gene Kim, 2018, y continuado en informes anuales que ahora se elaboran dentro de Google Cloud) descubrió que un pequeño conjunto de medidas predice el rendimiento de la entrega de software, y que los mejores equipos son rápidos y estables a la vez. El modelo actual tiene cinco métricas de rendimiento de la entrega de software, agrupadas en rendimiento (throughput) e inestabilidad. Sus nombres han evolucionado: el antiguo “tiempo medio de restauración” pasó a ser el tiempo de recuperación de despliegues fallidos en 2023, y la tasa de retrabajo de despliegues se añadió en 2024.

Dos advertencias. Primera: estas métricas describen el sistema de entrega de un equipo, no a las personas; usarlas para clasificar a las personas invita a hacer trampas. Segunda: son resultados. La arquitectura influye mucho en ellas: los equipos que pueden desplegar su parte de forma independiente, probarla de forma aislada y revertirla rápidamente obtienen mejores resultados, y esa es una de las razones por las que la investigación de DORA relaciona la arquitectura débilmente acoplada con un mayor rendimiento de la entrega.

**Métricas de rendimiento de la entrega de software de DORA (dora.dev, modelo actual)**

| Métrica | Grupo | Qué mide | Palancas arquitectónicas |
|---|---|---|---|
| **Tiempo de entrega de cambios** | Rendimiento (throughput) | Tiempo desde que se hace commit de un cambio en el control de versiones hasta que se ejecuta en producción | Pruebas automatizadas rápidas, unidades pequeñas desplegables de forma independiente, sin traspasos entre equipos |
| **Frecuencia de despliegue** | Rendimiento (throughput) | Con qué frecuencia se despliegan cambios en producción (o el tiempo entre despliegues) | Desplegabilidad independiente, pipeline automatizado, feature flags |
| **Tiempo de recuperación de despliegues fallidos** | Rendimiento (throughput) | Tiempo para recuperarse de un despliegue que falla y requiere intervención inmediata | Rollback automatizado, lanzamientos canario, buena monitorización, cambios pequeños |
| **Tasa de fallos de cambios** | Inestabilidad | Proporción de despliegues que requieren intervención inmediata, como un rollback o un hotfix | Capacidad de prueba, pruebas de contrato, cambios compatibles con versiones anteriores |
| **Tasa de retrabajo de despliegues** | Inestabilidad | Proporción de despliegues no planificados que se producen a causa de un incidente en producción | Calidad de las pruebas y las revisiones; observabilidad que detecta los problemas pronto |

## La mantenibilidad en ISO/IEC 25010

ISO/IEC 25010:2023 agrupa estas inquietudes bajo la característica mantenibilidad, con cinco subcaracterísticas: **modularidad** (el sistema está formado por componentes discretos, de modo que un cambio en uno tiene un impacto mínimo en los demás), **reusabilidad** (un activo puede usarse en más de un sistema o para construir otros activos), **analizabilidad** (la facilidad con la que puedes evaluar el impacto de un cambio, diagnosticar una deficiencia o encontrar las partes que hay que modificar), **modificabilidad** (el sistema puede cambiarse sin introducir defectos ni degradar la calidad) y **capacidad de prueba** (se pueden establecer criterios de prueba y realizar pruebas para determinar si se cumplen). La desplegabilidad no tiene una subcaracterística propia en la norma; la instalabilidad y la reemplazabilidad se encuentran bajo la flexibilidad. El modelo es una lista de comprobación, como explica la Sección 2.2; los escenarios son lo que hace concreto cada elemento.

**ATENCIÓN**

### Errores comunes

- Construir para cambios imaginarios: frameworks de plug-ins y capas de abstracción para variaciones que nadie ha pedido. Cada punto de extensión tiene un costo en complejidad y en pruebas. Pregunta qué probabilidad tiene el cambio antes de diferir la vinculación.
- Feature flags sin salida: cientos de flags obsoletos multiplican los caminos del código que nadie prueba. Da a cada flag un responsable y una fecha de eliminación.
- Los microservicios como remedio para una entrega lenta: dividir un sistema en servicios que aún deben lanzarse juntos (un monolito distribuido) empeora la desplegabilidad, no la mejora.
- Pruebas cargadas de mocks que verifican detalles de implementación: pasan mientras el sistema está roto y fallan cada vez que se refactoriza el código. Prefiere fakes y pruebas de contrato en los límites arquitectónicos.
- Medir el tiempo de entrega desde el inicio de un ticket en un equipo y desde el commit en otro, y luego compararlos. Define cada métrica una sola vez y aplícala de forma coherente.

**COMPRUEBA LO QUE HAS ENTENDIDO**

▼ **¿Por qué "el sistema deberá ser fácil de cambiar" no es un requisito útil?**

No dice qué cambiará, quién lo cambiará, cuándo, ni qué costo es aceptable, así que nadie puede diseñar para cumplirlo ni comprobar si se ha cumplido.

▼ **¿Cuál es la diferencia entre un stub y un mock?**

Un stub devuelve respuestas predefinidas para que la prueba pueda controlar las entradas; un mock además comprueba que se hicieron las llamadas esperadas, así que la prueba verifica las interacciones.

▼ **¿Por qué ayudan los feature flags a la desplegabilidad?**

Separan desplegar el código de lanzar una función, de modo que las funciones incompletas o arriesgadas pueden desplegarse desactivadas, lanzarse de forma gradual y volver a desactivarse sin un nuevo despliegue.

▼ **Un equipo despliega una vez al mes y cada lanzamiento contiene unos 200 cambios. ¿Qué métricas DORA es probable que se resientan, y por qué?**

El tiempo de entrega de cambios (los cambios esperan semanas al lanzamiento), la frecuencia de despliegue y, probablemente, la tasa de fallos de cambios y el tiempo de recuperación, porque los lotes grandes tienen más probabilidad de fallar y son más difíciles de diagnosticar y de revertir.

▼ **Inténtalo: Cada nuevo socio de reparto cuesta hoy 10 días-persona. Una refactorización de 15 días lo reduciría a 4 días. Se esperan cinco socios este año, a £600 al día. ¿Vale la pena la refactorización y cuándo alcanza el punto de equilibrio?**

Sin ello: 5 x 10 = 50 días, £30 000. Con ello: 15 + (5 x 4) = 35 días, £21 000, un ahorro de 15 días o £9000. Cada socio ahorra 6 días, así que se alcanza el punto de equilibrio en 15 / 6 = 2.5, es decir, con el tercer socio. Vale la pena si parece probable que haya al menos tres socios. También hace que cada integración sea menos arriesgada, pero comprueba antes la previsión, ya que el ahorro depende de ella.

**FUENTES**


- DORA: software delivery performance metrics: https://dora.dev/guides/dora-metrics/
- DORA: history of the software delivery metrics: https://dora.dev/guides/dora-metrics/history/
- Martin Fowler: Test Double: https://martinfowler.com/bliki/TestDouble.html
- Pete Hodgson: Feature Toggles (aka Feature Flags), martinfowler.com: https://martinfowler.com/articles/feature-toggles.html
- Martin Fowler: Inversion of Control Containers and the Dependency Injection pattern: https://martinfowler.com/articles/injection.html

SECCIÓN 2.7 DE 2.8 · INTERMEDIO · 22 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

# Seguridad, usabilidad, interoperabilidad, costo y sostenibilidad como cualidades

## ANTES DE LEER ESTO

Esta sección se basa en:

- 2.2 Modelos de calidad
- 2.3 Escenarios de calidad
- 2.5 Disponibilidad y fiabilidad

Términos anteriores que utiliza: ISO/IEC 25010, Escenario de atributo de calidad, Táctica, Restricción, API (interfaz de programación de aplicaciones)

## LA IDEA EN BREVE

Cuando eliges un banco, te importa que mantenga tu dinero a salvo, que su app sea fácil de usar, que funcione con otros bancos, que sus comisiones sean justas y, quizá, que no desperdicie energía. Ninguna de estas cosas es una funcionalidad que puedas señalar en una pantalla; cada una es una cualidad del servicio en su conjunto. Los arquitectos tienen que convertir cada una de ellas en algo lo bastante concreto como para diseñar en función de ello y comprobarlo, y a menudo tienen que equilibrar unas con otras.

## LO QUE NECESITAS SABER

- La seguridad parte de la confidencialidad, la integridad y la disponibilidad; ISO/IEC 25010:2023 enumera la confidencialidad, la integridad, el no repudio, la responsabilidad, la autenticidad y la resistencia, y sitúa la disponibilidad dentro de la fiabilidad.
- Los escenarios de seguridad nombran al atacante, el ataque, el objetivo y una respuesta medible, como el tiempo de detección o los datos expuestos; las tácticas resisten los ataques, los detectan, reaccionan ante ellos y permiten recuperarse.
- La capacidad de interacción (antes llamada usabilidad) afecta a la arquitectura a través de la capacidad de respuesta, deshacer, cancelar y el trabajo sin conexión; la accesibilidad conforme al nivel AA de WCAG 2.2 es cada vez más una restricción legal.
- La interoperabilidad exige acuerdo sobre el significado, no solo sobre el formato; los estándares del sector y los adaptadores reducen el costo.
- El costo y el carbono se expresan mejor por unidad de trabajo; la especificación SCI (ISO/IEC 21031:2024) calcula ((E x I) + M) por R y solo cuenta las reducciones reales.

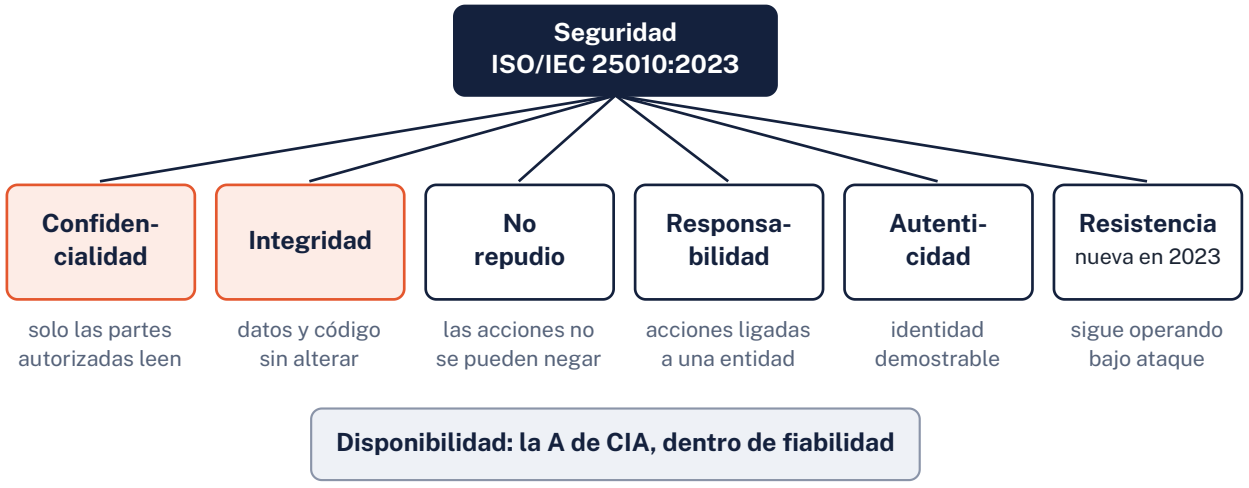
- La seguridad (safety), la observabilidad, la residencia de datos, la auditabilidad y la conservación de datos también son cualidades, y deben recogerse pronto como restricciones que se puedan probar.

## La seguridad como atributo de calidad

La seguridad es la capacidad del sistema de proteger los datos y las funciones frente al acceso y el uso no autorizados sin dejar de dar servicio a quienes sí están autorizados. Los arquitectos suelen partir de la tríada CIA: **confidencialidad** (solo las partes autorizadas pueden leer los datos), **integridad** (los datos y el código no se alteran sin autorización, y las alteraciones se pueden detectar) y **disponibilidad** (los usuarios autorizados pueden usar el servicio cuando lo necesitan). Los ataques se corresponden con las tres: robar registros de clientes vulnera la confidencialidad, cambiar los datos bancarios de un beneficiario vulnera la integridad y un ataque de denegación de servicio vulnera la disponibilidad.

ISO/IEC 25010:2023 afina el panorama. Su característica de seguridad tiene seis subcaracterísticas: confidencialidad, integridad, **no repudio** (se puede demostrar que las acciones o los eventos han tenido lugar, de modo que una parte no pueda negarlos después), **responsabilidad** (accountability: las acciones se pueden atribuir de forma inequívoca a la entidad que las realizó), **autenticidad** (se puede demostrar que la identidad de un sujeto o de un recurso es la que dice ser) y, como novedad de la edición de 2023, **resistencia** (el producto sigue funcionando como está previsto mientras sufre el ataque de un actor malicioso). En ese modelo, la disponibilidad aparece dentro de la fiabilidad, y por eso se trató en la Sección 2.5.

**Las subcaracterísticas de seguridad**




La confidencialidad y la integridad proceden de la tríada CIA; las otras cuatro hacen explícitas la responsabilidad, la prueba de identidad y el comportamiento bajo ataque.

## Escribir escenarios de seguridad

"El sistema debe ser seguro" no se puede diseñar ni probar. Un escenario de seguridad nombra una fuente (quién ataca y con qué acceso), un estímulo (qué intenta), un artefacto (qué es el objetivo), un entorno (en línea, durante un incidente, con el atacante ya dentro de la red), una respuesta y una medida. Las medidas de seguridad suelen referirse al tiempo de detección, a la cantidad de daño o al tiempo de restauración, porque "ningún ataque tiene éxito nunca" no se puede verificar.

Tres ejemplos para un banco en línea. *Relleno de credenciales (credential stuffing):* un atacante externo prueba pares filtrados de nombre de usuario y contraseña contra la API de inicio de sesión, a 50 intentos por segundo y desde muchas direcciones; el sistema detecta el patrón en menos de 2 minutos, exige una verificación adicional a las direcciones de origen o las bloquea, y no se toma el control de las cuentas protegidas con autenticación multifactor. *Manipulación interna:* un empleado de soporte con acceso legítimo cambia los datos del beneficiario de un cliente; el cambio requiere la aprobación de una segunda persona, y el registro de auditoría guarda quién, qué y cuándo de una forma que el empleado no puede alterar. *Copia de seguridad robada:* un atacante obtiene una copia de la copia de seguridad de la base de datos de la semana pasada; todos los datos personales que contiene están cifrados con claves guardadas por separado, así que no se puede leer ningún dato de clientes.

Observa que cada escenario implica una arquitectura distinta: limitación de tasa (rate limiting) y detección de anomalías en el perímetro; un flujo de aprobación por cuatro ojos y registros a prueba de manipulaciones; y cifrado con una gestión de claves separada. Encontrar estos escenarios de forma sistemática es la tarea del modelado de amenazas, que se trata en el Capítulo 8.

**SI ESTO ES NUEVO PARA TI**

### Paso a paso: distinguir los términos de seguridad

La idea que requiere más atención aquí es distinguir las subcaracterísticas de seguridad que suenan parecidas, sobre todo la autenticidad, la responsabilidad y el no repudio. Toma el escenario anterior de manipulación interna, en el que un empleado de soporte cambia los datos del beneficiario de un cliente, y relaciona cada salvaguarda que contiene con una subcaracterística.

- Los datos del beneficiario no deben cambiar sin autorización, y cualquier cambio debe poder detectarse; en el escenario, una segunda persona debe aprobar el cambio. Eso protege la **integridad**.
- El registro de auditoría guarda quién hizo el cambio, en qué consistió y cuándo, de modo que se puede atribuir a ese empleado concreto. Eso es la **responsabilidad**: saber quién lo hizo.
- El registro solo puede nombrar a la persona correcta si el banco puede estar seguro de que quien inició sesión como ese empleado es realmente ese empleado, y no alguien que usa una contraseña robada. Poder demostrar que una persona o un sistema es quien o lo que dice ser es la **autenticidad**.
- El registro se guarda de una forma que el empleado no puede alterar, así que, si más adelante dice "eso no lo hice yo", el banco puede demostrar que sí. Eso es el **no repudio**: la prueba de que la acción tuvo lugar, de modo que no se puede negar.
- Las otras dos corresponden a otros escenarios. La **confidencialidad** (solo las partes autorizadas pueden leer los datos) es lo que el escenario de la copia de seguridad robada protege con cifrado. La **resistencia**, seguir funcionando como está previsto bajo ataque, es lo que pone a prueba el escenario del relleno de credenciales.
- **La lección:** la confidencialidad y la integridad protegen los datos; la autenticidad, la responsabilidad y el no repudio se refieren a las personas que actúan sobre ellos: demostrar quiénes son, rastrear lo que hicieron y demostrarlo después. Cada una exige un mecanismo distinto, así que nombra en cada escenario la que quieres decir.

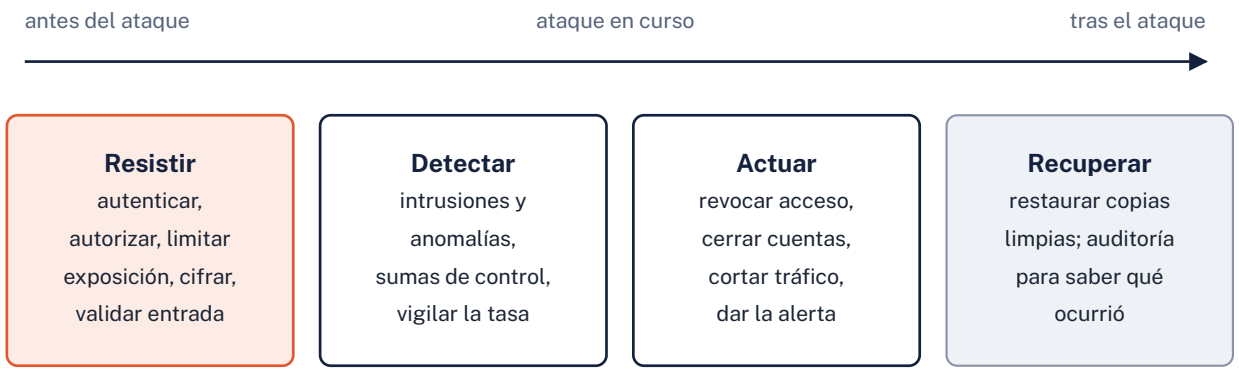
## Tácticas de seguridad: una visión general

Bass, Clements y Kazman agrupan las tácticas de seguridad según el momento en que actúan. **Resistir los ataques**: identificar, autenticar y autorizar a los actores; limitar el acceso y la exposición (menos puertos abiertos, menos endpoints públicos, mínimo privilegio); cifrar los datos en tránsito y en reposo; separar entidades (los datos sensibles en su propio almacén o segmento de red); cambiar la configuración predeterminada y validar la entrada. **Detectar los ataques**: detección de intrusiones, detección de la denegación de servicio, verificación de la

integridad de los mensajes y localización de retrasos o patrones inusuales. **Reaccionar ante los ataques**: revocar el acceso, bloquear cuentas tras varios inicios de sesión fallidos, informar a los operadores y a los usuarios. **Recuperarse de los ataques**: restaurar un estado correcto conocido (de nuevo, las tácticas de disponibilidad) y mantener una pista de auditoría para entender qué ocurrió y respaldar el no repudio.

Esta sección solo presenta las familias. Principios como la defensa en profundidad y el mínimo privilegio, los protocolos de identidad, el cifrado, zero trust y la seguridad de la cadena de suministro son el tema del Capítulo 8. Lo que importa aquí desde el punto de vista arquitectónico es que la seguridad es una propiedad de toda la estructura, no un componente que se añade al final: dónde se sitúan los límites de confianza, qué componente guarda qué datos y cómo se autentican los servicios entre sí son decisiones tempranas y costosas de cambiar.

**Tácticas antes, durante y después de un ataque**




Agrupación de Bass, Clements y Kazman; detalles en el Capítulo 8

Las tácticas de resistencia reducen la probabilidad de éxito; las de detección, reacción y recuperación limitan el daño cuando la resistencia falla.

## Usabilidad, capacidad de interacción y accesibilidad

ISO/IEC 25010:2023 cambió el nombre de la usabilidad por el de *capacidad de interacción*: el grado en que usuarios especificados pueden interactuar con un producto para intercambiar información a través de la interfaz de usuario y completar tareas concretas. Sus subcaracterísticas son la reconocibilidad de la adecuación, la capacidad de aprendizaje, la operabilidad, la protección contra errores del usuario, la implicación del usuario (que sustituyó a la estética de la interfaz de usuario), la inclusividad, la asistencia al usuario y la autodescriptividad. Gran parte de esto es diseño de interacción más que arquitectura, pero varias inquietudes de usabilidad llegan hasta lo más profundo de la estructura.

La capacidad de respuesta es la más evidente. Las pautas de factores humanos, establecidas desde hace mucho y popularizadas por Jakob Nielsen, dan tres límites: unos 0.1 segundos se perciben como instantáneos, alrededor de 1 segundo mantiene el hilo de pensamiento del usuario y unos 10 segundos son el límite de la atención. Cumplirlos puede exigir caché, procesamiento asíncrono con indicación del progreso o actualizaciones optimistas. Deshacer y cancelar son cuestiones arquitectónicas: admitir "deshacer las cinco últimas acciones" requiere comandos que se puedan revertir, y cancelar una subida larga requiere una operación que se pueda interrumpir limpiamente. Trabajar sin conexión en una app para dispositivos móviles requiere almacenamiento local y sincronización con gestión de conflictos. Todo esto es mucho más barato de incorporar en el diseño que de añadir a posteriori.

La accesibilidad, es decir, la capacidad de las personas con discapacidad de usar el producto, es cada vez más una restricción legal y no una preferencia. Las Web Content Accessibility Guidelines (pautas de accesibilidad para el contenido web) versión 2.2 (WCAG 2.2), Recomendación del W3C desde octubre de 2023, definen criterios de conformidad comprobables en los niveles A, AA y AAA; el nivel AA es la referencia habitual en el ámbito legal y en la contratación. WCAG 2.2 añadió nueve criterios, entre ellos un tamaño mínimo de los objetivos para la entrada con puntero y la autenticación accesible (ningún paso del inicio de sesión puede depender de una prueba de función cognitiva, como recordar una contraseña o transcribir un código, salvo que se ofrezca al usuario un método alternativo o una ayuda, como pegar desde un gestor de contraseñas; hay además unas pocas excepciones concretas). En la Unión Europea, el Acta Europea de Accesibilidad (Directiva (UE) 2019/882) se aplica desde el 28 de junio de 2025 a muchos productos y servicios de consumo, como el comercio electrónico, los servicios bancarios, los libros electrónicos y los servicios de transporte de pasajeros. Para un arquitecto, la accesibilidad es, por tanto, una restricción que hay que recoger pronto: afecta a la elección del framework de frontend, al sistema de diseño, al flujo de autenticación y al pipeline de pruebas (comprobaciones automáticas de accesibilidad en la integración continua, además de pruebas manuales con tecnologías de apoyo).

## Interoperabilidad

La interoperabilidad es el grado en que dos o más sistemas pueden intercambiar información y usar la información que intercambian. ISO/IEC 25010 la sitúa dentro de la compatibilidad, junto a la coexistencia. Tiene dos niveles. La interoperabilidad *sintáctica* significa que los sistemas se ponen de acuerdo en el formato y el protocolo: ambos hablan HTTP y JSON. La interoperabilidad *semántica* significa que se ponen de acuerdo en el significado: `"amount": 1050` está en peniques, no en libras; `status: "closed"` significa pagado, no cancelado. La mayoría de los fallos de integración son semánticos.

Los sectores reducen el costo del acuerdo semántico con estándares compartidos: HL7 FHIR para intercambiar historias clínicas, ISO 20022 para los mensajes financieros, GTFS para los horarios del transporte público. Entre las tácticas arquitectónicas están descubrir los servicios a través de un registro, adaptar las interfaces con adaptadores y traductores, y gestionar las versiones de forma explícita para que los socios más antiguos sigan funcionando. Un escenario de interoperabilidad podría decir: “Un nuevo hospital se une a la red e intercambia resúmenes de pacientes con recursos FHIR R4 en un plazo de 4 semanas, sin cambios en el servicio central de historias clínicas.”

## El costo como cualidad

El costo rara vez figura en los modelos de calidad, pero condiciona todas las decisiones arquitectónicas. Distingue el **costo de construcción** (el esfuerzo de crear el sistema), el **costo de operación** (infraestructura, licencias, servicios de terceros, soporte) y el **costo de cambio** (el esfuerzo de cada cambio futuro, que es de lo que trata la modificabilidad). En la nube, el costo de operación es variable y depende directamente de la arquitectura: los servicios que se comunican en exceso pagan por la transferencia de red, los clústeres sobredimensionados pagan por capacidad ociosa y un patrón de consultas descuidado puede multiplicar la factura de una base de datos.

Una forma útil de convertir el costo en un atributo de calidad es el costo unitario: el costo por transacción, por usuario activo o por pedido. A diferencia de una factura mensual total, muestra si la arquitectura se vuelve más o menos eficiente a medida que crece el negocio, y se puede escribir como escenario: “Con 3 millones de pedidos al mes, el costo de infraestructura se mantiene por debajo de 1 penique por pedido.” Las prácticas de FinOps para gestionarlo se tratan en el Capítulo 10.

EJEMPLO PRÁCTICO

## Ejemplo práctico: costo unitario y carbono por solicitud en una API de reparto de comida

Una API de pedidos se ejecuta en un clúster que cuesta £24 000 al mes y gestiona 3 000 000 de pedidos al mes. Un análisis estima su consumo de energía en 1200 kWh al mes, en una red eléctrica con una media de 250 g de $CO_2e$ por kWh, y se le asignan cada mes 60 kg de $CO_2e$ de emisiones incorporadas del hardware. La fórmula de Software Carbon Intensity es SCI = ((E x I) + M) por R.

| | |
|---|---|
| **Costo por pedido** | £24 000 / 3 000 000 = £0.008, es decir, 0.8 peniques por pedido |
| **Emisiones operativas (E x I)** | 1200 kWh x 250 g = 300 000 g = 300 kg de $CO_2e$ |
| **Sumar las emisiones incorporadas (M)** | 300 + 60 = 360 kg de $CO_2e$ |
| **Dividir entre la unidad funcional (R = un pedido)** | 360 000 g / 3 000 000 = 0.12 g de $CO_2e$ por pedido |
| **Cambio: almacenar las cartas en caché, reducir la energía un 25%** | E = 900 kWh: (900 x 250 + 60 000) / 3 000 000 = 0.095 g por pedido |
| **Cambio: ejecutar los trabajos por lotes en una región con 50 g por kWh** | Reduce I para esa parte de la carga; la planificación consciente del carbono reduce la puntuación sin cambiar el código |
| **Crecimiento hasta 6 000 000 de pedidos en el mismo clúster** | El costo por pedido se reduce a la mitad, a 0.4 peniques; el carbono por pedido también baja, siempre que la energía crezca menos que proporcionalmente (los servidores ociosos siguen consumiendo electricidad, así que una mayor utilización es más eficiente) |

Ambas métricas son tasas por unidad de trabajo útil, así que premian la eficiencia en lugar de penalizar el crecimiento.

La especificación SCI solo cuenta las reducciones reales: comprar compensaciones de carbono o certificados de energía renovable no reduce la puntuación.

Las cifras de energía y de la red eléctrica son estimaciones; lo que importa es medirlas siempre de la misma manera para poder comparar los cambios.

## Eficiencia energética y sostenibilidad

El software no emite carbono por sí mismo, pero el hardware en el que se ejecuta consume electricidad e incorpora las emisiones de su fabricación. La especificación Software Carbon Intensity (SCI) de la Green Software Foundation, publicada por ISO como ISO/IEC 21031:2024, define una tasa: las emisiones operativas (la energía E multiplicada por la intensidad de carbono I de la electricidad) más las emisiones incorporadas M, divididas entre una unidad funcional R, como un usuario, una llamada a la API o un pedido. Identifica tres palancas: la **eficiencia energética** (usar menos electricidad para el mismo trabajo), la **eficiencia del hardware** (usar menos dispositivos y aprovecharlos más y durante más tiempo) y la **conciencia del carbono** (ejecutar el trabajo flexible cuando y donde la electricidad es más limpia).

La arquitectura influye en las tres. El ajuste del tamaño de los recursos (right-sizing) y una mayor utilización mejoran la eficiencia del hardware; la caché, los formatos de datos eficientes y evitar el trabajo innecesario mejoran la eficiencia energética; las colas y el procesamiento por lotes permiten desplazar el trabajo en el tiempo o en el espacio. La sostenibilidad suele ir de la mano del costo, pero no siempre: mantener una réplica en espera activa (warm standby) en una segunda región mejora la disponibilidad y duplica las emisiones incorporadas. Los grandes modelos de IA hacen la cuestión más acuciante, como explica el Capítulo 12.

## Seguridad (safety)

La seguridad (safety) es la ausencia de un riesgo inaceptable de daño a las personas, los bienes o el medio ambiente. Se distingue de la seguridad frente a ataques (security): esta protege el sistema de las personas con intenciones dañinas; la safety protege a las personas del sistema, incluso cuando nadie actúa con malicia. ISO/IEC 25010:2023 añadió la safety como novena característica, con las subcaracterísticas restricción operativa, identificación de riesgos, a prueba de fallos (fail safe), advertencia de peligro e integración segura.

Los ámbitos críticos para la seguridad (safety) tienen sus propios estándares, que imponen arquitectura y procesos: IEC 61508 para la seguridad funcional en general, ISO 26262 para los vehículos de carretera, IEC 62304 para el software de dispositivos médicos y DO-178C para el software embarcado en aeronaves. Las tácticas típicas son un comportamiento predeterminado a prueba de fallos (un tren se detiene cuando pierde las señales), canales de monitorización independientes que pueden anular al controlador principal, redundancia con votación y una separación estricta entre los componentes críticos para la seguridad y todo lo demás, para que un defecto en el sistema de infoentretenimiento no pueda llegar a los frenos. Ten en cuenta que "a prueba de fallos" y "altamente disponible" pueden entrar en conflicto: un sistema a prueba de fallos a menudo se detiene, lo cual es seguro pero no está disponible.

## La observabilidad, una cualidad emergente

La observabilidad es el grado en que puedes entender el estado interno de un sistema desde fuera, a partir de los datos que emite, lo bastante bien como para responder preguntas que no se previeron cuando se construyó. No figura en ISO/IEC 25010, aunque la analizabilidad y la responsabilidad se acercan a ella. En los sistemas distribuidos se ha convertido en una cualidad por derecho propio, porque el tiempo de recuperación, y por tanto la disponibilidad, depende de lo rápido que las personas puedan averiguar qué falla. Desde el punto de vista arquitectónico requiere logs estructurados, métricas y trazas distribuidas con identificadores de correlación que sigan a una solicitud a través de los servicios, diseñados desde el principio. Un escenario podría ser: “Ante cualquier pago fallido, un ingeniero de guardia puede identificar el servicio que falla y el recorrido de la solicitud en menos de 5 minutos.” Consulta la observabilidad en el Capítulo 9.

## Cualidades legales y regulatorias

Las leyes y la normativa crean cualidades que es fácil pasar por alto hasta que llega una auditoría. La residencia de datos exige que ciertos datos se almacenen o se procesen en un país o una región concretos; puede venir de normas sectoriales, de contratos del sector público o de exigencias de los clientes. El RGPD (GDPR) de la UE no exige por sí mismo que los datos permanezcan en la UE, pero las transferencias fuera de ella necesitan una base jurídica y garantías, lo que en la práctica determina dónde se ejecutan los servicios y las copias de seguridad. La **auditabilidad** significa poder mostrar, más adelante, quién hizo qué y cuándo: registros a prueba de manipulaciones, conservados durante un periodo definido y con los relojes sincronizados. Las normas de **conservación y eliminación** exigen que los datos se conserven durante un tiempo mínimo (registros financieros) y se eliminen tras un tiempo máximo o cuando se solicite (datos personales), lo que resulta difícil en sistemas con muchas copias, cachés y copias de seguridad. La **explicabilidad** y la supervisión humana se exigen para algunas decisiones automatizadas, como explica el capítulo sobre el Reglamento de IA de la UE.

Trátalas como restricciones con criterios de aceptación: "los datos de los clientes alemanes del sector público se almacenan y procesan solo en regiones de la UE, incluidas las copias de seguridad y los logs" se puede probar, y descarta de inmediato algunos servicios gestionados. Los requisitos de privacidad se tratan en privacidad desde el diseño y el RGPD.

**Resumen: las cualidades de esta sección y medidas de ejemplo**

| Calidad | Medida de la respuesta de ejemplo | Palanca arquitectónica típica |
|---|---|---|
| **Seguridad (confidencialidad)** | Ningún dato personal legible a partir de una copia de seguridad robada | Cifrado en reposo con claves gestionadas por separado |
| **Seguridad (responsabilidad)** | Cada cambio en los datos de un beneficiario atribuible a una persona, registros a prueba de manipulaciones | Registro de auditoría central, almacenamiento de escritura única |
| **Capacidad de interacción** | El 95% de los usuarios nuevos completa el pago sin ayuda en menos de 3 minutos | Flujos sencillos, respuestas rápidas, deshacer |
| **Accesibilidad** | Todos los recorridos del cliente cumplen el nivel AA de WCAG 2.2 | Sistema de diseño accesible, comprobaciones automáticas en CI |
| **Interoperabilidad** | Nuevo socio integrado con el estándar de API publicado en un plazo de 4 semanas | Formatos estándar, adaptadores, API versionadas |
| **Costo** | Infraestructura por debajo de 1 penique por pedido con 3 millones de pedidos al mes | Ajuste del tamaño (right-sizing), caché, autoescalado |
| **Sostenibilidad** | SCI por debajo de 0.1 g de $CO_2$e por pedido | Eficiencia, utilización, planificación consciente del carbono |
| **Observabilidad** | Servicio que falla identificado en menos de 5 minutos para cualquier solicitud fallida | Logs estructurados, métricas, trazado distribuido |
| **Residencia de datos** | Todos los datos de los clientes de la UE, incluidas las copias de seguridad, permanecen en regiones de la UE | Almacenamiento y procesamiento fijados a una región |

**COMPRUEBA LO QUE HAS ENTENDIDO**

▼ **¿Cuál es la diferencia entre la responsabilidad y el no repudio?**

La responsabilidad significa que las acciones se pueden atribuir a la entidad que las realizó; el no repudio significa que hay una prueba lo bastante sólida como para que la entidad no pueda negar después la acción.

▼ **¿Por qué "deshacer las cinco últimas acciones" cuenta como un requisito arquitectónico y no solo de la interfaz de usuario?**

Porque exige modelar las operaciones como comandos reversibles con un historial almacenado, lo que afecta a cómo se estructuran los cambios de estado en todo el sistema.

▼ **Una API consume 800 kWh al mes en una red eléctrica con 200 g de CO2e por kWh, tiene asignados 40 kg de emisiones incorporadas y atiende 2 000 000 de solicitudes. ¿Cuál es su SCI por solicitud?**

(800 x 200 g + 40 000 g) / 2 000 000 = 200 000 / 2 000 000 = 0.1 g de CO2e por solicitud.

▼ **¿En qué se diferencia la seguridad (safety) de la seguridad frente a ataques (security)?**

La security protege el sistema y sus datos de las personas con intenciones dañinas; la safety protege a las personas y el medio ambiente del daño que el sistema podría causar, incluso sin ningún actor malicioso.

▼ **Inténtalo: Una API de venta de entradas cuesta £18 000 al mes para 1 500 000 entradas y usa 800 kWh a 300 g CO2e por kWh, con 40 kg de emisiones incorporadas. Calcula el costo y el SCI por entrada, y luego el SCI si la caché reduce la energía un 25%.**

Costo: £18 000 / 1 500 000 = 1.2 peniques por entrada. Emisiones operativas: 800 x 300 g = 240 kg; sumando 40 kg da 280 kg, y 280 000 g / 1 500 000 = unos 0.187 g por entrada. Con caché: 600 kWh dan 180 kg, más 40 kg son 220 kg, o unos 0.147 g por entrada. Las cifras por unidad siguen teniendo sentido a medida que crece el volumen, así que haz seguimiento de ellas en lugar de la factura total.

**FUENTES**


- W3C: Web Content Accessibility Guidelines (WCAG) 2.2: https://www.w3.org/TR/WCAG22/
- W3C WAI: What's new in WCAG 2.2: https://www.w3.org/WAI/standards-guidelines/wcag/new-in-22/
- Green Software Foundation: Software Carbon Intensity (SCI) specification: https://sci.greensoftware.foundation/
- Nielsen Norman Group: Response Times, the 3 Important Limits: https://www.nngroup.com/articles/response-times-3-important-limits/
- OWASP Application Security Verification Standard: https://owasp.org/www-project-application-security-verification-standard/

SECCIÓN 2.8 DE 2.8 · INTERMEDIO · 22 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

# Elicitación, priorización y compromisos entre atributos de calidad

## ANTES DE LEER ESTO

Esta sección se basa en:

- 1.5 Partes interesadas e inquietudes
- 1.6 Decisiones y compromisos
- 2.3 Escenarios de calidad

Términos anteriores que utiliza: Parte interesada, Inquietud, Escenario de atributo de calidad, Percentil (p95, p99), Registro de decisiones de arquitectura (ADR)

## LA IDEA EN BREVE

Al planificar unas vacaciones en familia, cada uno quiere algo distinto: uno las quiere baratas, otro quiere piscina, otro quiere trayectos cortos. No se puede tener todo, así que toca sentarse juntos, anotar lo que cada persona necesita de verdad, votar lo que más importa y acordar a qué se renuncia. Los arquitectos hacen lo mismo con las cualidades de un sistema: reúnen a todos los interesados, convierten los deseos en objetivos concretos y medibles, los clasifican y dejan por escrito los compromisos que eligen.

## LO QUE NECESITAS SABER

- Los requisitos de calidad deben elicitarse a partir de las partes interesadas, los objetivos de negocio, la normativa, los incidentes pasados, la competencia y las mediciones del sistema existente, porque rara vez se formulan sin que nadie los pida.
- El Quality Attribute Workshop del SEI tiene ocho pasos; sus resultados son los impulsores arquitectónicos y unos escenarios refinados y priorizados, clasificados mediante una votación con un presupuesto fijo de votos.
- Un árbol de utilidad refina la utilidad en atributos de calidad, refinamientos y escenarios concretos, cada uno valorado (importancia para el negocio, dificultad técnica); los escenarios (H,H) reciben antes que ningún otro la atención del diseño.
- Las cualidades entran en conflicto (rendimiento y seguridad, disponibilidad y consistencia, modificabilidad y rendimiento, costo y casi todo lo demás, usabilidad y seguridad), y la resolución adecuada depende del contexto y del valor para el negocio.

- Registra los compromisos significativos en registros de decisiones y haz que cada objetivo sea medible con una escala, un medidor y unos niveles; después, haz su seguimiento a lo largo del tiempo.
- Las funciones de aptitud convierten las medidas de la respuesta de los escenarios en comprobaciones automáticas que se ejecutan a medida que el sistema evoluciona.

## De dónde salen los requisitos de calidad

Los requisitos funcionales suelen llegar en un backlog. Los requisitos de calidad rara vez lo hacen: las partes interesadas los dan por supuestos ("por supuesto que tiene que ser rápido y seguro") o no saben que tienen que formularlos. Por eso los arquitectos tienen que elicitarlos activamente, a partir de varias fuentes.

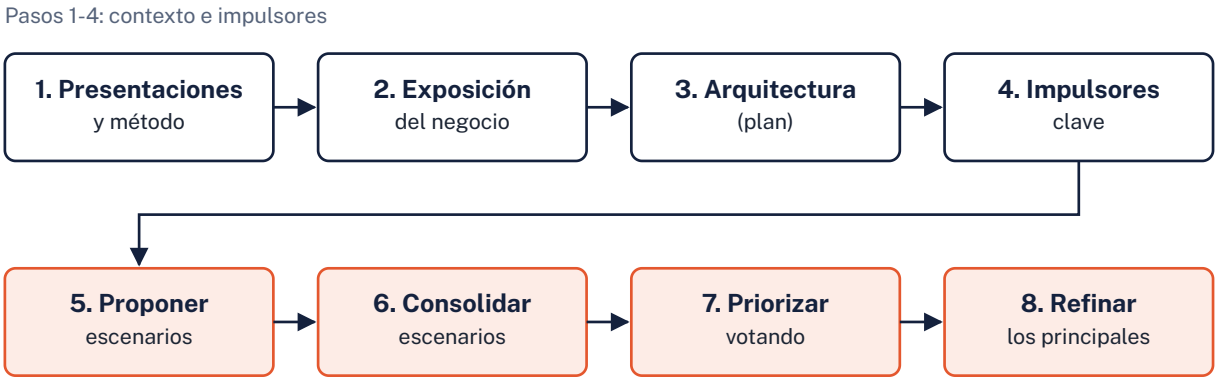
**Partes interesadas.** Cada grupo tiene inquietudes distintas: a los clientes les importan la velocidad y la facilidad de uso; a operaciones, la monitorización y la recuperación; a finanzas, el costo de operación; a soporte, el diagnóstico de problemas; a cumplimiento normativo, las pistas de auditoría. El mapa de partes interesadas del Capítulo 1 es el punto de partida. **Objetivos de negocio.** "Expandirnos a tres países nuevos el año que viene" implica localización, residencia de datos y escalabilidad; "reducir un 20% los costos de atención al cliente" implica usabilidad y autoservicio. **Normativa y contratos.** La protección de datos, la legislación de accesibilidad, las normas sectoriales y los SLA con los clientes crean restricciones innegociables. **Incidentes.** Cada caída, brecha o día lento del pasado revela una cualidad que estaba poco especificada; las revisiones posteriores a los incidentes son una fuente muy rica. **La competencia y el mercado.** Si los rivales cargan en un segundo, dos segundos son un problema de calidad. **El sistema existente.** Su comportamiento medido hoy (la carga real, los percentiles de latencia reales, el tiempo de entrega de cambios real) da una línea base que convierte "más rápido" en un número.

Las entrevistas y los cuestionarios funcionan, pero los atributos de calidad salen a la luz mejor cuando las partes interesadas se escuchan unas a otras. Un product owner que oye del responsable de operaciones lo que cuesta una hora de caída a la hora de la cena de un viernes priorizará de otra manera. Esa es la idea que hay detrás de los talleres estructurados.

## El Quality Attribute Workshop

El Quality Attribute Workshop (QAW), desarrollado en el Software Engineering Institute (Barbacci y sus colegas; la tercera edición del informe del método se publicó en 2003), es una sesión guiada por facilitadores, normalmente de un día, que se celebra pronto, antes de que la arquitectura quede fijada. Reúne a entre 5 y 30 partes interesadas para producir escenarios de atributos de calidad refinados y priorizados. Tiene ocho pasos.

### Los ocho pasos del QAW




Los cuatro primeros pasos establecen el contexto y los impulsores; los cuatro últimos producen, fusionan, clasifican y refinan los escenarios.

### Qué ocurre en cada paso

1. **1. Presentación e introducciones**
   Los facilitadores explican el propósito y el método; los participantes se presentan y exponen sus inquietudes.

2. **2. Exposición del negocio o de la misión**
   Un representante del negocio describe el contexto, los objetivos y los requisitos de alto nivel: por qué existe el sistema y qué es lo que más importa.

3. **3. Exposición del plan arquitectónico**
   Un responsable técnico presenta los planes actuales, los sistemas existentes, las restricciones conocidas y los compromisos tecnológicos, por incipientes que sean.

4. **4. Identificación de los impulsores arquitectónicos**
   Los facilitadores extraen los principales impulsores arquitectónicos oídos hasta ese momento (objetivos, cualidades, restricciones) y los contrastan con el grupo.

5. **5. Lluvia de ideas de escenarios**
   Los participantes proponen escenarios por turnos rotatorios; los facilitadores se aseguran de que cada impulsor tenga al menos uno y de que estén representados los escenarios de uso normal, de crecimiento y exploratorios (de estrés).

6 **6. Consolidación de escenarios**
Los escenarios casi duplicados se fusionan, pero solo cuando quienes los propusieron están de acuerdo, para que los votos no se dividan.

7 **7. Priorización de escenarios**
Cada participante recibe un presupuesto de votos de alrededor del 30% del número de escenarios y lo gasta en dos rondas, con cualquier número de votos por escenario. El recuento clasifica los escenarios.

8 **8. Refinamiento de escenarios**
Los cuatro o cinco escenarios principales se escriben completos: fuente, estímulo, artefacto, entorno, respuesta y medida de la respuesta, junto con los objetivos de negocio a los que sirven y las preguntas abiertas.

Los equipos que no pueden dedicar un día entero suelen celebrar un "mini-QAW" más corto, una variante más ligera descrita por profesionales como Michael Keeling, Will Chaparro y Thijmen de Gooijer, que usa una taxonomía de atributos de calidad como guía y dura unas pocas horas con un grupo más reducido. Lo esencial se mantiene: las partes interesadas juntas, escenarios en lugar de adjetivos, una votación explícita y el refinamiento de los ganadores.

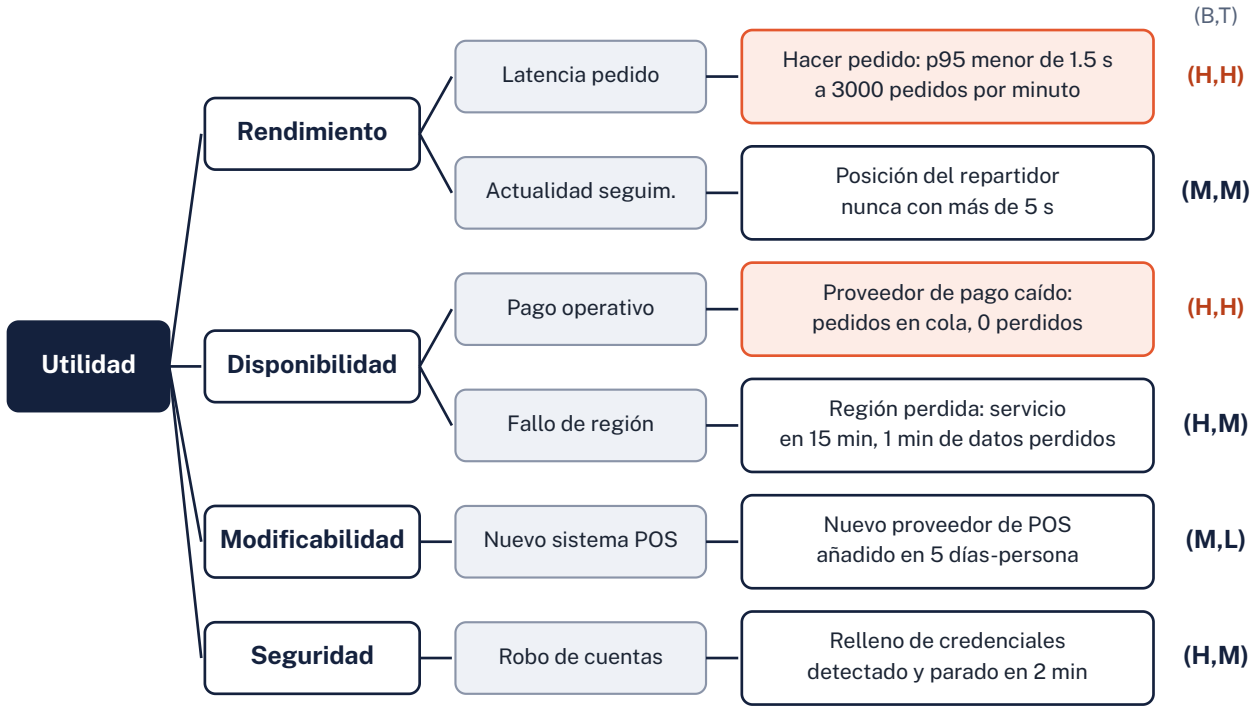
## El árbol de utilidad

Un árbol de utilidad es una forma de organizar y priorizar los requisitos de calidad, introducida como parte del Architecture Tradeoff Analysis Method (ATAM), que Rick Kazman, Mark Klein y Paul Clements desarrollaron en el SEI; el método de evaluación en sí se trata en el Capítulo 11. La raíz es la *utilidad*, la bondad global del sistema. El segundo nivel contiene los atributos de calidad (rendimiento, disponibilidad, etc.). El tercer nivel refina cada uno en inquietudes más concretas (para el rendimiento: la latencia de los pedidos, la actualidad del seguimiento). Las hojas son escenarios concretos y medibles.

Cada hoja se valora con un par: la **importancia para el negocio** (cuánto le importa el escenario al negocio: alta (High), media (Medium) o baja (Low)) y la **dificultad técnica** o el riesgo (lo difícil que es lograrlo con la arquitectura: alta, media o baja). El par se escribe (B,T), por ejemplo (H,M). Los escenarios valorados (H,H) son las máximas prioridades de la arquitectura: son los que más importan y los más difíciles, así que merecen un esfuerzo de diseño temprano, prototipos y análisis. Los escenarios (H,L) importan, pero son fáciles, así que basta con un enfoque estándar; los escenarios (L,H) son candidatos a descartarse o aplazarse, porque son difíciles pero valen poco.

El árbol es útil incluso fuera de una evaluación formal. Hace visibles los compromisos en una sola página, muestra las lagunas (una cualidad sin escenarios) e impide que la parte interesada que más alza la voz fije las prioridades por su cuenta.

**Un árbol de utilidad para una plataforma de reparto de comida**




Los dos escenarios valorados (H,H), resaltados, son las máximas prioridades de la arquitectura; el escenario de modificabilidad (M,L) se puede cumplir con un buen diseño corriente.

EJEMPLO PRÁCTICO

## Ejemplo práctico: construir el árbol de utilidad de una plataforma de reparto de comida

Una plataforma de reparto de comida conecta a clientes, 4000 restaurantes y 9000 repartidores en un país. El negocio planea duplicar el volumen de pedidos en 18 meses. El pico del viernes por la noche es hoy de 3000 pedidos por minuto. Un QAW de dos horas produjo 26 escenarios; con 12 participantes, cada uno tuvo 8 votos (el 30% de 26 es 7.8, redondeado hacia arriba a un 8 par). Los escenarios más votados se convirtieron en hojas del árbol.

| | |
|---|---|
| **Rendimiento: latencia de los pedidos (H,H)** | Los clientes hacen pedidos al ritmo pico de 3000 por minuto; el 95% de las confirmaciones de pedido se devuelve en menos de 1.5 s y el 99% en menos de 3 s. Importancia alta: los carritos abandonados son ingresos perdidos. Dificultad alta: la solicitud pasa por los servicios de carrito, precios, pagos y restaurantes |
| **Comprobación de capacidad para ese escenario** | 3000 pedidos/min = 50 pedidos/s. Duplicar el volumen da 100 pedidos/s. La ley de Little necesita el tiempo medio, no un percentil (Sección 2.4), así que el objetivo de 1.5 s solo da una cifra de planificación prudente: si la media llegara a 1.5 s, habría unos 100 x 1.5 = 150 pedidos en curso a la vez; la cifra real es menor porque la media queda muy por debajo del p95 |
| **Disponibilidad: disponibilidad del pago (H,H)** | Cuando el proveedor de pagos agota el tiempo de espera durante el pico, los pedidos se aceptan y quedan en cola para el pago, no se pierde ningún pedido confirmado y se informa a los clientes en menos de 10 s. Es difícil porque requiere pagos asíncronos, reintentos idempotentes y una experiencia de usuario para el estado "pendiente" |
| **Disponibilidad: fallo de región (H,M)** | Si falla la región de la nube, el servicio se reanuda en una segunda región en menos de 15 minutos (RTO) con no más de 1 minuto de datos de pedidos perdidos (RPO). Importante, pero técnicamente bien conocido |
| **Seguridad: robo de cuentas (H,M)** | Un ataque de relleno de credenciales se detecta y se bloquea en menos de 2 minutos; no se toma el control de ninguna cuenta protegida con autenticación multifactor |
| **Rendimiento: actualidad del seguimiento (M,M)** | La posición de un repartidor en el mapa del cliente nunca tiene más de 5 s de antigüedad con la carga pico |
| **Modificabilidad: nuevo proveedor de POS (M,L)** | Un nuevo proveedor de sistemas de punto de venta (POS) para restaurantes se integra en 5 días-persona añadiendo un adaptador (Sección 2.6) |
| **Prioridades resultantes** | Hacer primero un prototipo y una prueba de carga del recorrido de los pedidos y del diseño de pagos en cola; usar patrones multirregión estándar para el fallo de región; mantener sencilla la interfaz del adaptador |

Las valoraciones son juicios acordados por el grupo, no cálculos. Registra quién valoró qué y por qué, para que las valoraciones se puedan cuestionar más adelante.

El árbol guía el plan de trabajo: los dos escenarios (H,H) reciben spikes arquitectónicos (investigaciones acotadas) en los primeros sprints, mientras que el escenario (M,L) no necesita una atención especial.

El seguimiento de los repartidores solo se valoró con importancia media, lo que sorprendió a los ingenieros; el product owner explicó que los clientes consultan sobre todo la hora estimada de llegada, no el mapa.

## Priorizar cuando todo es "crítico"

Cada parte interesada valorará su propia inquietud como alta. Ayudan las técnicas que obligan a elegir de verdad. Un presupuesto fijo de votos (como en el QAW) obliga a gastar votos escasos. La comparación por pares ("si solo pudieras tener una, ¿búsqueda en menos de un segundo o cero pérdida de datos en la conmutación por error?") revela el orden real. Poner dinero o tiempo junto a cada objetivo ("cuatro nueves costarán £9000 más al mes y dos meses de trabajo") cambia muchas opiniones. Y distinguir las *restricciones* (los requisitos legales de accesibilidad, la residencia de datos) de las *preferencias* evita que lo innegociable se ceda por accidente.

Las prioridades no son fijas. Una start-up que busca sus primeros clientes hace bien en anteponer el tiempo de llegada al mercado y la modificabilidad a una disponibilidad de cinco nueves; el mismo producto, tres años después, con clientes empresariales y SLA contractuales, debe volver a clasificarlas. Revisa el árbol de utilidad cuando cambie el negocio.

## Conflictos entre cualidades

Las cualidades interactúan. Algunas se refuerzan entre sí: la modularidad ayuda a la capacidad de prueba, que a su vez ayuda a la desplegabilidad. Muchas entran en conflicto, porque la táctica que ayuda a una cualidad perjudica a otra. El ATAM, descrito por Bass, Clements y Kazman, llama puntos de compromiso a los puntos en los que una decisión arquitectónica afecta a varias cualidades. Reconocer los conflictos habituales te permite plantearlos antes de que sorprendan a nadie.

**Conflictos habituales entre cualidades y cómo los resuelve el contexto**

| Conflicto | Por qué tiran en direcciones opuestas | Formas de resolverlo en contexto |
|---|---|---|
| **Rendimiento frente a seguridad** | El cifrado, las comprobaciones de autenticación, la validación de la entrada y el registro de auditoría añaden trabajo a cada solicitud | Medir la sobrecarga real (a menudo pequeña con TLS y hardware modernos); almacenar en caché brevemente las decisiones de autorización; llevar las comprobaciones costosas al perímetro; no ceder nunca en la seguridad de datos que causarían un daño grave si quedaran expuestos |
| **Disponibilidad frente a consistencia** | Durante una partición de red, un sistema replicado debe elegir entre responder (quizá con datos desactualizados) y negarse | Decidir por operación: una carta o la posición de un repartidor pueden estar algo desactualizadas; un pago o una reserva de existencias no. Consulta CAP y PACELC en el Capítulo 6 |
| **Modificabilidad frente a rendimiento** | Las capas, las interfaces, los intermediarios y la vinculación tardía añaden indirección y saltos | Mantener la indirección donde es probable que haya cambios; optimizar la ruta crítica medida; aceptar algunos atajos deliberados, documentados en un registro de decisiones |
| **Costo frente a casi todo** | La redundancia, las regiones adicionales, más pruebas y un hardware más rápido cuestan dinero | Vincular cada objetivo de calidad a su valor para el negocio; usar presupuestos de errores y el costo unitario; comprar nueves solo donde una caída cuesta más que evitarla |
| **Usabilidad frente a seguridad** | La autenticación robusta, los tiempos de expiración y las confirmaciones añaden fricción | Autenticación basada en el riesgo (pedir más verificación solo para las acciones de riesgo), passkeys e inicio de sesión único, duraciones de sesión razonables; implicar a los usuarios en las pruebas |
| **Tiempo de llegada al mercado frente a mantenibilidad** | Los atajos aceleran el primer lanzamiento y ralentizan los siguientes | Asumir deuda técnica de forma deliberada, registrarla y planificar su pago (Capítulo 1) |

La resolución adecuada siempre depende del contexto. "La seguridad está por encima del rendimiento" es un eslogan, no una decisión: 2 ms más de negociación TLS en un sitio web público son irrelevantes, mientras que las comprobaciones de fraude síncronas en cada pago con tarjeta en un estadio abarrotado pueden ser un problema real que merece una respuesta arquitectónica (un modelo de aprobar primero y comprobar después, con límites). Pregunta: ¿a qué objetivo de negocio sirve cada lado, cuánto cuesta cada opción y quién tiene que estar de acuerdo?

**SI ESTO ES NUEVO PARA TI**

### Paso a paso: un conflicto, dos respuestas

El punto en el que más gente tropieza aquí es que un conflicto entre dos cualidades no se resuelve de una vez para todo el sistema. Se resuelve operación por operación, según lo que cada una vale para el negocio. Sigue la fila de disponibilidad frente a consistencia de la tabla a través de la plataforma de reparto de comida.

- Supón que la plataforma guarda copias de algunos de sus datos en más de una máquina, para poder sobrevivir a un fallo, y transmite cada cambio de una copia a las demás.
- Supón ahora que el enlace de red entre dos copias se rompe durante un tiempo: una partición de red. Una copia que no puede enterarse de los últimos cambios puede seguir respondiendo con lo que tiene, que puede estar desactualizado, o negarse hasta que vuelva el enlace. No puede garantizar ambas cosas. El Capítulo 6 vuelve sobre esta elección.
- Posición del repartidor: un punto en el mapa del cliente con unos segundos de antigüedad hace poco daño, y los clientes consultan sobre todo la hora estimada de llegada. Gana la disponibilidad: seguir respondiendo.
- Pago: actuar con información desactualizada podría suponer cobrar dos veces a un cliente, o tratar un pedido como pagado cuando el pago nunca se completó. Gana la consistencia: retener el pago y mostrar el pedido como pendiente en lugar de adivinar.
- Así que una misma plataforma da respuestas opuestas al mismo conflicto. Ninguna sale de un eslogan como "la consistencia está por encima de la disponibilidad"; cada una se deriva de lo que costaría una respuesta errónea en esa operación.
- **La lección:** cuando dos cualidades entran en conflicto, pregunta a qué objetivo de negocio sirve cada lado en esa operación concreta, y decide ahí. Después registra la elección y sus razones, como explica la siguiente parte de esta sección, para que nadie la revierta más adelante sin saber por qué.

## Registrar las decisiones y hacer el seguimiento de las cualidades

Un compromiso que no se deja por escrito se volverá a discutir, a menudo por personas que no estaban presentes. Registra los significativos como registros de decisiones de arquitectura (Capítulo 4): el contexto, las cualidades en tensión, las opciones consideradas, la decisión y sus consecuencias, incluido aquello a lo que has renunciado conscientemente.

Después, haz que cada objetivo de calidad sea medible y visible. El Planguage de Tom Gilb ofrece un formato riguroso: una *escala* (scale: qué se mide y en qué unidad), un *medidor* (meter: cómo se mide) y niveles como *fallo* (fail: el umbral por debajo del cual no se cumple el requisito), *objetivo* (goal: la meta) y *ampliado* (stretch) o *deseo* (wish), que son niveles más

ambiciosos. Aplicado al escenario de latencia de los pedidos: escala, segundos desde el envío hasta la confirmación en el percentil 95; medidor, monitorización de usuarios reales durante el pico del viernes; fallo, por encima de 3 s; objetivo, 1.5 s; ampliado, 1 s.

Por último, haz el seguimiento de las medidas a lo largo del tiempo en un panel que las partes interesadas puedan ver, junto a los SLO (Capítulo 9), y revísalas cada vez que cambien el negocio, la carga o el equipo.

## De los escenarios a las funciones de aptitud

La forma más sólida de mantener vigente un requisito de calidad es comprobarlo automáticamente, cada vez que el sistema cambia. Neal Ford, Rebecca Parsons y Patrick Kua, en *Building Evolutionary Architectures* (2017; segunda edición con Pramod Sadalage, 2022), llaman a esa comprobación función de aptitud (fitness function) arquitectónica: una evaluación objetiva de lo bien que el sistema cumple una característica arquitectónica. Una prueba de carga en el pipeline que hace fallar la compilación si la latencia p95 de los pedidos supera 1.5 s a 100 pedidos por segundo es una función de aptitud. También lo son una prueba que falla si algún módulo fuera de `payments` importa la biblioteca del proveedor de pagos (lo que protege el escenario de modificabilidad), un simulacro programado de conmutación por error que mide el tiempo de recuperación o un análisis de dependencias que bloquea las bibliotecas con vulnerabilidades conocidas.

Las funciones de aptitud son el punto de llegada natural del trabajo de este capítulo: un deseo vago se convierte en un escenario (Sección 2.3), el escenario recibe una prioridad en el árbol de utilidad y la medida de la respuesta del escenario se convierte en una comprobación automática. No todas las cualidades se pueden automatizar a bajo costo (la usabilidad sigue necesitando personas), pero cada una que automatizas frena una erosión silenciosa. El Capítulo 11 trata su diseño en profundidad: funciones de aptitud y arquitectura evolutiva.

## Lista de comprobación: celebrar un taller de atributos de calidad

Adaptada para una sesión de medio día o de un día; un "mini-QAW" más corto puede cubrir lo mismo con un grupo más reducido.

- [ ] **Invita a la combinación adecuada**
  Entre 5 y 30 personas: responsables de negocio, usuarios o sus representantes, operaciones, seguridad, soporte, desarrolladores y cualquiera que dé el visto bueno al cumplimiento normativo. La ausencia de operaciones es la laguna más habitual.
- [ ] **Envía una breve lectura previa**
  Los objetivos de negocio, las restricciones conocidas y un borrador de lista de atributos de calidad (con ISO/IEC 25010 como guía), para que la gente llegue con las ideas en marcha.
- [ ] **Empieza por los objetivos de negocio, no por la tecnología**
  Una parte interesada de alto nivel explica por qué existe el sistema, cómo sería el éxito y cuánto costaría el fracaso.
- [ ] **Acuerda los impulsores arquitectónicos**
  Una lista breve de los objetivos, las cualidades y las restricciones que darán forma a la arquitectura, contrastada con el grupo antes de empezar con los escenarios.
- [ ] **Propón escenarios por rondas**
  Todos aportan al menos dos, por turnos, para que se oigan las voces más calladas. Incluye uso normal, crecimiento ("dentro de dos años") y estrés ("cuando el proveedor de pagos está caído").
- [ ] **Fusiona duplicados solo con consentimiento**
  La consolidación evita que los votos se dividan, pero quien propuso el escenario debe estar de acuerdo en que dos escenarios significan realmente lo mismo.
- [ ] **Vota con un presupuesto fijo**
  Cada persona recibe un número de votos igual a alrededor del 30% del número de escenarios, que gasta en dos rondas. Esto revela las prioridades reales y evita el "todo es crítico".
- [ ] **Refina por completo los principales**
  Escribe los cuatro o cinco principales como escenarios de seis partes con medidas de la respuesta medibles, y anota el objetivo de negocio al que sirve cada uno.
- [ ] **Registra las preguntas abiertas y los conflictos**
  Los desacuerdos sobre la prioridad o la viabilidad son resultados valiosos; enuméralos con un responsable.

- [ ] **Publica el resultado en pocos días**
  Comparte los escenarios priorizados, añádelos a la documentación de arquitectura y al backlog, y planifica cuándo se comprobará cada uno.

**ATENCIÓN**

### Errores comunes

- Recoger las cualidades solo del patrocinador de negocio: las inquietudes de operaciones, seguridad y soporte aparecen entonces después de la puesta en producción, cuando resultan caras.
- Aceptar adjetivos: "escalable" y "robusto" en una lista de requisitos sin un escenario detrás.
- Valorar todas las hojas (H,H): si todo es la máxima prioridad, nada lo es. Insiste hasta que las valoraciones discriminen.
- Tratar el árbol de utilidad como un artefacto de un solo uso: hay que revisarlo cuando cambien los objetivos o la carga.
- Resolver los conflictos en silencio en el código: un desarrollador que elimina discretamente el registro de auditoría para cumplir un objetivo de latencia ha tomado una decisión arquitectónica que nadie ha registrado.

**COMPRUEBA LO QUE HAS ENTENDIDO**

**▼ En un QAW con 20 escenarios consolidados, ¿aproximadamente cuántos votos recibe cada participante?**

Alrededor del 30% de 20, es decir, 6 votos, gastados en dos rondas de 3.

**▼ Una hoja de un árbol de utilidad está valorada (L,H). ¿Qué sugiere eso?**

Que importa poco al negocio, pero es difícil de lograr, así que es candidata a descartarse, aplazarse o renegociarse en lugar de guiar la arquitectura.

**▼ ¿Por qué "la seguridad siempre está por encima del rendimiento" es una mala forma de resolver un conflicto?**

Porque la respuesta correcta depende de la sobrecarga real, del daño que evita el control de seguridad y del valor del rendimiento para el negocio; el conflicto debe resolverse caso por caso, con cifras, y registrarse.

▼ **Da un ejemplo de función de aptitud para un requisito de modificabilidad.**

Una prueba automática en la compilación que falla si algún módulo distinto del adaptador de pagos importa la biblioteca del proveedor de pagos, de modo que el proveedor sigue siendo sustituible.

▼ **Inténtalo: Las partes interesadas dicen que todo es crítico. Valora estos puntos (importancia, dificultad) en un árbol de utilidad de reparto de comida: confirmación del pedido en 1.5 s en el pico; exportación del informe mensual en una hora; añadir un proveedor de pagos en un mes. ¿Cuál recibe antes atención de diseño?**

La confirmación del pedido es (H,H): los carritos perdidos son ingresos perdidos, y la solicitud cruza varios servicios bajo carga de pico, así que recibe primero la atención del diseño. Un nuevo proveedor de pagos podría ser (M,M): valioso y moderadamente difícil, salvo que ya exista un adaptador. La exportación del informe es (L,L) o (M,L): se ejecuta fácilmente de noche. Las valoraciones obligan a establecer un orden sin discutir que un atributo de calidad siempre valga más. Anota las razones y compruébalas con las partes interesadas mediante una votación.

**FUENTES**


- SEI: Quality Attribute Workshop collection: https://www.sei.cmu.edu/library/quality-attribute-workshop-collection/
- SEI: Architecture Tradeoff Analysis Method (ATAM) collection: https://www.sei.cmu.edu/library/architecture-tradeoff-analysis-method-collection/
- IREB RE Magazine: Discover Quality Requirements with the Mini-QAW: https://re-magazine.ireb.org/articles/discover-quality-requirements-with-the-mini-qaw
- Thoughtworks Technology Radar: Architectural fitness function: https://www.thoughtworks.com/radar/techniques/architectural-fitness-function

## Lecturas recomendadas

- *Software Architecture in Practice (4th edition)*, Len Bass, Paul Clements and Rick Kazman (2021), Addison-Wesley. La fuente de los escenarios de atributos de calidad y de las tácticas, con un capítulo para cada cualidad principal y otro sobre los requisitos arquitectónicamente significativos.
  https://www.sei.cmu.edu/library/software-architecture-in-practice-fourth-edition/
- *ISO/IEC 25010:2023 Systems and software engineering: Product quality model*, ISO/IEC JTC 1/SC 7 (2023), International Organization for Standardization. El modelo estándar de calidad del producto, con nueve características; su norma complementaria, ISO/IEC 25019:2023, trata la calidad en uso.
  https://www.iso.org/standard/78176.html
- *Characterizing Architecturally Significant Requirements*, Lianping Chen, Muhammad Ali Babar and Bashar Nuseibeh (2013), IEEE Software, volume 30, issue 2. Un estudio empírico sobre cómo reconocen los profesionales los requisitos que dan forma a una arquitectura.
  https://pure.itu.dk/en/publications/characterizing-architecturally-significant-requirements/
- *The Tail at Scale*, Jeffrey Dean and Luiz André Barroso (2013), Communications of the ACM, volume 56, issue 2. Un artículo breve y fácil de leer sobre por qué las respuestas lentas poco frecuentes dominan en los grandes sistemas con fan-out, y las técnicas que usa Google para controlarlas.
  https://research.google/pubs/pub40801/
- *Designing Data-Intensive Applications (2nd edition)*, Martin Kleppmann and Chris Riccomini (2026), O'Reilly Media. Un tratamiento claro de la carga, los percentiles, la fiabilidad y la escalabilidad en los sistemas de datos, con muchos ejemplos reales.
- *Fundamentals of Software Architecture (2nd edition)*, Mark Richards and Neal Ford (2025), O'Reilly Media. La visión de un profesional sobre cómo identificar y priorizar las características de la arquitectura y protegerlas con funciones de aptitud.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (eds) (2016), O'Reilly Media. Se puede leer gratis en línea; los capítulos sobre objetivos de nivel de servicio y monitorización muestran cómo se miden las cualidades en producción.
  https://sre.google/books/
- *Systems Performance: Enterprise and the Cloud (2nd edition)*, Brendan Gregg (2020), Addison-Wesley. La referencia estándar sobre análisis del rendimiento, desde el encolamiento y la utilización hasta el método USE.

# Glosario

**Accesibilidad**
El grado en que las personas con discapacidad, incluidas las que usan tecnologías de apoyo como los lectores de pantalla, pueden usar un producto.

**Acuerdo de nivel de servicio (SLA)**
Contrato con los clientes que establece un nivel de servicio y las consecuencias, como créditos de servicio, si no se cumple.

**Árbol de utilidad**
Árbol, procedente del ATAM, que refina la utilidad global en atributos de calidad y después en escenarios concretos, cada uno valorado según su importancia para el negocio y su dificultad técnica.

**Calidad en uso**
La calidad que un producto ofrece a las personas en un contexto de uso específico (si es beneficioso, está libre de riesgos y es aceptable). Desde 2023 tiene su propia norma, ISO/IEC 25019.

**Capacidad de prueba (testabilidad)**
La facilidad con que puede demostrarse, mediante pruebas, que un sistema o componente se comporta como se pretende, lo que depende de poder controlar sus entradas y su estado y observar sus salidas.

**Costo unitario**
El costo de operar un sistema dividido por una unidad de trabajo útil, como el costo por pedido o por usuario activo, para poder seguir la eficiencia a medida que crece el negocio.

**Defecto (fault)**
La causa de un problema, como un bug, un disco averiado o una configuración errónea. Un defecto puede permanecer latente hasta que algo lo activa.

**Degradación elegante**
Seguir proporcionando las funciones más importantes cuando falla parte de un sistema, en lugar de fallar por completo; por ejemplo, una tienda que sigue aceptando pedidos mientras las recomendaciones están fuera de servicio.

**Desplegabilidad**
La rapidez, la seguridad y la previsibilidad con que un cambio puede ponerse en producción, y retirarse de nuevo si es necesario.

**Disponibilidad**
La proporción del tiempo, o de las solicitudes, en la que un sistema está listo para prestar su servicio cuando se le pide, normalmente expresada como un porcentaje, por ejemplo 99.9%.

**Doble de prueba**
Sustituto de un colaborador real usado en una prueba, como un stub con respuestas predefinidas, un fake con una implementación sencilla que funciona o un mock que comprueba las llamadas que recibe.

**Elasticidad**
La capacidad de añadir y quitar recursos automáticamente a medida que la carga sube y baja, de modo que la capacidad sigue a la demanda y solo pagas por lo que usas.

**Elicitación**
Obtener activamente los requisitos de las partes interesadas, los documentos, la normativa y los sistemas existentes, en lugar de esperar a que alguien los formule.

**Encolamiento (queueing)**
Esperar a un recurso ocupado. La teoría de colas estudia cómo crece el tiempo de espera con la utilización; explica por qué el tiempo de respuesta se dispara a medida que un recurso se acerca al 100% de ocupación.

**Error**
Parte del estado interno de un sistema que es incorrecta porque se ha activado un defecto, como un registro corrupto. Un error puede provocar o no un fallo.

**Escalabilidad**
La capacidad de un sistema para soportar más carga añadiendo recursos, idealmente con un aumento predecible y aproximadamente proporcional de la capacidad y del costo.

**Escenario concreto**
Un escenario escrito para un sistema en particular, con actores, eventos, condiciones y cifras específicos, listo para diseñar a partir de él y probarlo.

**Escenario de atributo de calidad**
Enunciado breve y comprobable de un requisito de calidad con seis partes: fuente del estímulo, estímulo, artefacto, entorno, respuesta y medida de la respuesta.

**Escenario general**
Una plantilla independiente del sistema que enumera las fuentes, los estímulos, los entornos, las respuestas y las medidas típicos de un atributo de calidad.

**Estímulo**
El evento del que trata un escenario de atributo de calidad, como una ráfaga de solicitudes, la caída de un servidor, una solicitud de cambio o un ataque.

**Failover (conmutación por error)**
Trasladar el trabajo de un componente que ha fallado a una réplica o un componente de reserva sano, de forma automática o manual.

**Fallo (failure)**
Un suceso en el que el servicio que presta un sistema se desvía de lo que debería ser, de un modo que sus usuarios pueden observar.

**Fan-out (abanico)**
Un patrón en el que una solicitud entrante desencadena varias solicitudes paralelas a otros servicios o servidores, cuyas respuestas luego se combinan.

**Fiabilidad**
La probabilidad de que un sistema funcione correctamente, sin fallos, durante un periodo determinado y en unas condiciones determinadas. Un sistema puede tener una disponibilidad alta y aun así ser poco fiable si falla a menudo pero se recupera rápido.

**FURPS+**
Una clasificación de la calidad que surgió en Hewlett-Packard como FURPS (funcionalidad, usabilidad, fiabilidad, rendimiento y soportabilidad), ampliada después con "+" para restricciones como las de diseño, implementación, interfaz y límites físicos.

**Heartbeat (latido)**
Un mensaje periódico de "sigo vivo" que envía un componente; si los mensajes se detienen, un monitor supone que el componente ha fallado y actúa.

**Impulsor arquitectónico**
Un objetivo, un requisito de atributo de calidad o una restricción que da forma de manera significativa a la arquitectura.

**Indicador de nivel de servicio (SLI)**
Medición cuidadosamente definida del comportamiento de un servicio, como la proporción de solicitudes que tienen éxito en menos de 300 ms.

**Interoperabilidad**
El grado en que los sistemas pueden intercambiar información y usar correctamente lo que intercambian, lo que requiere acuerdo tanto sobre el significado como sobre el formato.

**Inyección de dependencias**
Proporcionar a un componente desde fuera los colaboradores que necesita (una base de datos, un reloj, una pasarela de pagos) en lugar de dejar que los cree, para que puedan intercambiarse, por ejemplo, por imitaciones en las pruebas.

**ISO/IEC 25010**
La norma internacional que define el modelo de calidad del producto. La edición de 2023 define nueve características, desde la adecuación funcional hasta la seguridad física (safety), cada una dividida en subcaracterísticas.

**Latencia de cola (tail latency)**
Los tiempos de respuesta de las solicitudes más lentas, descritos normalmente con percentiles altos como p99 o p99.9. A gran escala, la cola suele decidir la experiencia de los usuarios.

**Ley de Amdahl**
Una fórmula (Gene Amdahl, 1967) que muestra que la aceleración obtenida al añadir procesadores está limitada por la parte del trabajo que debe ejecutarse en serie.

**Ley de escalabilidad universal (USL)**
Modelo de Neil Gunther sobre cómo crece la capacidad al añadir nodos, teniendo en cuenta la contención por los recursos compartidos y el costo de mantener la coherencia de los datos. Predice un máximo a partir del cual añadir nodos reduce la capacidad.

**Ley de Little**
La regla $L = \lambda \times W$: el número medio de elementos en un sistema estable es igual a la tasa media de llegada multiplicada por el tiempo medio que cada elemento pasa en él.

**Mantenibilidad**
En ISO/IEC 25010, la eficacia y eficiencia con que puede modificarse un producto; abarca la modularidad, la reusabilidad, la analizabilidad, la modificabilidad y la capacidad de prueba.

**Medida de la respuesta**
El número que hace comprobable un escenario, como '99% en menos de 800 ms', 'servicio restablecido en 2 minutos' o 'cambio completado en 3 días-persona'.

**Métricas DORA**
Las medidas de rendimiento de la entrega de software del programa de investigación DORA: tiempo de entrega de cambios, frecuencia de despliegue, tiempo de recuperación de despliegues fallidos, tasa de fallos de cambios y tasa de retrabajo de despliegues.

**Modelo de calidad**
Lista estructurada de características de calidad y sus subcaracterísticas, que se usa como vocabulario compartido y como lista de comprobación al especificar o evaluar la calidad.

**Modificabilidad**
Lo barato y seguro que resulta cambiar un sistema: el costo, el esfuerzo y el riesgo de un cambio, y hasta dónde se propaga más allá del lugar donde se hace.

**Momento de vinculación (binding time)**
El momento en que se fija una decisión: al escribir el código, en la compilación, en el despliegue, en el arranque o mientras el sistema se ejecuta.

**Objetivo de nivel de servicio (SLO)**
Objetivo interno para un SLI en una ventana de tiempo, como que el 99.9% de las solicitudes tengan éxito en 30 días.

**Objetivo de punto de recuperación (RPO)**
La mayor cantidad de datos, medida como un periodo de tiempo, que el negocio puede aceptar perder en un desastre; por ejemplo, los cambios de los últimos 15 minutos.

**Objetivo de tiempo de recuperación (RTO)**
El tiempo máximo que el negocio puede aceptar entre un desastre y el restablecimiento del servicio.

**Observabilidad**
Lo bien que las personas pueden entender lo que ocurre dentro de un sistema en ejecución a partir de los datos que emite (logs, métricas, trazas), incluso ante problemas que nadie había previsto.

**Percentil (p95, p99)**
El valor por debajo del cual queda una proporción dada de las mediciones. Un p99 de 400 ms significa que el 99% de las solicitudes tardaron 400 ms o menos y el 1% tardó más.

**Punto de compromiso**
Decisión arquitectónica que afecta a más de un atributo de calidad, mejorando uno y perjudicando otro.

**Quality Attribute Workshop (QAW)**
Método guiado del SEI en el que las partes interesadas generan ideas, consolidan, votan y refinan escenarios de atributos de calidad en las primeras fases de un proyecto.

**Redundancia**
Mantener más de una copia de un componente para que otra pueda tomar el relevo si una falla.

**Rendimiento (throughput)**
La cantidad de trabajo que un sistema completa por unidad de tiempo, como 1200 solicitudes por segundo o 40 000 pedidos por hora.

**Requisito arquitectónicamente significativo (ASR)**
Un requisito que tiene un efecto medible en la arquitectura de un sistema: si cambiara, la estructura tendría que cambiar. La mayoría de los ASR son atributos de calidad o restricciones.

**Requisito funcional**
Una declaración de lo que un sistema debe hacer: un comportamiento, un cálculo o una información que debe proporcionar, como "un cliente puede cancelar un pedido antes de que se envíe".

**Requisito implícito**
Una cualidad que todas las partes interesadas dan por supuesta pero que nadie pone por escrito, como "los pedidos nunca se pierden" o "el sistema sigue funcionando durante un cambio de hora".

**Requisito no funcional (NFR)**
Nombre muy usado pero impreciso para los requisitos de calidad y las restricciones. Muchos arquitectos lo evitan porque estos requisitos califican a las funciones en lugar de existir al margen de ellas.

**Residencia de datos**
Un requisito de que ciertos datos se almacenen, y a veces se traten, solo en un país o una región determinados.

**Resiliencia**
La capacidad más amplia de un sistema para absorber perturbaciones, mantener su servicio esencial (quizá de forma degradada) y volver después a la normalidad.

**Restricción**
Una decisión de diseño que ya se ha tomado por ti, sin libertad o con muy poca libertad restante, como un lenguaje de programación obligatorio, una normativa, un plazo fijo o un presupuesto.

**Rollback**
Devolver un sistema a un estado o una versión anterior que se sabe correcto después de que un cambio cause problemas.

**Seguridad (safety)**
Ausencia de un riesgo inaceptable de daños a personas, bienes o el medio ambiente causados por el sistema, actúe o no alguien con mala intención.

**Software Carbon Intensity (SCI)**
Especificación de la Green Software Foundation, publicada como ISO/IEC 21031:2024, que expresa las emisiones de carbono del software como una tasa: ((energía x intensidad de carbono de la red eléctrica) + emisiones incorporadas) por unidad funcional.

**Subcaracterística**
Cualidad más específica dentro de una característica de un modelo de calidad; por ejemplo, la tolerancia a fallos y la capacidad de recuperación son subcaracterísticas de la fiabilidad en ISO/IEC 25010.

**Táctica**
Decisión de diseño que influye en cómo responde un sistema a un estímulo de calidad, como usar caché para reducir la latencia o redundancia para sobrevivir a una caída. Las tácticas son los bloques de construcción de los patrones.

**Tiempo de respuesta**
El tiempo total entre que un cliente envía una solicitud y recibe la respuesta: el tiempo de servicio más cualquier tiempo de espera en colas, más los retardos de red.

**Tiempo de servicio**
El tiempo que un recurso dedica realmente a trabajar en una solicitud, sin contar el tiempo que la solicitud pasa esperando a ese recurso.

**Tiempo medio de reparación (MTTR)**
El tiempo medio desde un fallo hasta que se restablece el servicio. Algunas organizaciones interpretan la R como recuperación (recover) o restablecimiento (restore); indica siempre qué puntos de inicio y fin se miden.

**Tiempo medio entre fallos (MTBF)**
El tiempo medio que transcurre entre un fallo de un componente o sistema reparable y el siguiente.

**Tiempo medio hasta el fallo (MTTF)**
El tiempo medio que un componente funciona correctamente antes de fallar.

**Tolerancia a fallos**
La capacidad de un sistema de seguir prestando su servicio aunque una de sus partes haya fallado, normalmente mediante redundancia.

**Tríada CIA**
Los tres objetivos clásicos de la seguridad de la información: confidencialidad (solo las partes autorizadas pueden leer), integridad (sin cambios no autorizados) y disponibilidad (utilizable cuando se necesita).

**Utilización**
La fracción de tiempo que un recurso está ocupado, que suele escribirse ρ (rho). Un servidor que atiende 40 solicitudes por segundo, cada una de las cuales necesita 20 ms de trabajo, tiene una utilización del 80%.

**WCAG**
Las Pautas de Accesibilidad para el Contenido Web (Web Content Accessibility Guidelines) del W3C. La versión 2.2 (2023) fija criterios de conformidad comprobables en los niveles A, AA y AAA; el nivel AA es la referencia legal habitual.