BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 10

Arquitectura nativa de la nube y de infraestructura

Dónde y cómo se ejecuta el software es hoy una decisión arquitectónica. Este capítulo trata los modelos de servicio y de despliegue en la nube y el modelo de responsabilidad compartida, los contenedores y Kubernetes, la aplicación de doce factores, la infraestructura como código y GitOps, los pipelines de CI/CD y las estrategias de despliegue, el diseño multirregión, la recuperación ante desastres y el edge computing, la ingeniería de plataformas, y FinOps y la dependencia del proveedor (lock-in). Está dirigido a desarrolladores y arquitectos que conocen las cualidades operativas del Capítulo 9 y quieren elegir, justificar y operar las plataformas de las que dependen sus sistemas, sin que nadie les venda una pila tecnológica.

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

DE UN VISTAZO

Aprenderás a

- Comparar IaaS, CaaS, PaaS, FaaS y SaaS y los modelos de despliegue, y asignar responsabilidades para una carga de trabajo según el modelo de responsabilidad compartida.
- Explicar cómo Kubernetes mantiene un estado deseado declarado mediante su plano de control, sus nodos y sus bucles de reconciliación, configurar requests, limits y sondas, y juzgar cuándo Kubernetes es excesivo.
- Auditar una aplicación frente a los doce factores y los principios que van más allá de ellos, y priorizar las correcciones según el riesgo.
- Diseñar código de infraestructura con estado remoto, bloqueo, módulos, entornos, comprobaciones de políticas, pruebas y reconciliación con GitOps.
- Elegir una estrategia de despliegue (gradual, azul-verde, canario o sombra) y separar el despliegue del lanzamiento con feature flags.
- Seleccionar una estrategia de recuperación ante desastres a partir del RTO, el RPO y el costo, y explicar las consecuencias de replicar datos entre regiones y de llevar trabajo al edge.
- Explicar cómo una plataforma interna para desarrolladores con caminos dorados reduce la carga cognitiva, y cómo medir si tiene éxito.
- Calcular costos unitarios como el costo por pedido, identificar los impulsores arquitectónicos del costo en la nube y evaluar la dependencia del proveedor (lock-in) como un compromiso con precio.

Antes de empezar

Capítulo 0, sección 0.7 (servidores y la nube) y sección 0.8 (cómo se construye el software); Capítulo 9, en especial el escalado, la ausencia de estado, los SLO y la observabilidad. El Capítulo 6, sobre sistemas distribuidos, ayuda.

Tiempo

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

Términos clave

nativo de la nube (cloud native) · modelo de responsabilidad compartida · zona de disponibilidad · imagen de contenedor · Kubernetes · bucle de reconciliación · aplicación de doce factores (twelve-factor app) · infraestructura como código · GitOps · canary release (lanzamiento canario) · RTO y RPO · camino dorado (golden path) · FinOps · dependencia del proveedor (lock-in)

Por qué la infraestructura se ha convertido en arquitectura

Durante la mayor parte de la historia del software, la infraestructura era algo que ocurría después de la arquitectura: un equipo de operaciones recibía una aplicación terminada y buscaba servidores donde ejecutarla. Esa separación ha desaparecido. En un sistema moderno, la elección entre máquinas virtuales, contenedores, plataformas y funciones decide buena parte del costo; el número de zonas de disponibilidad y de regiones decide la disponibilidad; el

pipeline de despliegue decide con qué rapidez y seguridad puede un equipo cambiar cualquier cosa; y la propia infraestructura se escribe como código, se revisa en pull requests y se versiona junto con la aplicación. Las decisiones sobre dónde y cómo se ejecuta el software están hoy entre las más significativas que toma un arquitecto, exactamente en el sentido del Capítulo 1: son costosas de revertir, atraviesan a todos los equipos y fijan los atributos de calidad que el Capítulo 2 te pidió especificar.

Este capítulo se apoya en dos anteriores. El Capítulo 0 presentó los servidores, las máquinas virtuales, los contenedores, los modelos de servicio en la nube, las regiones y las zonas de disponibilidad (servidores y la nube). El Capítulo 9 explicó cómo hacer que los sistemas escalen, se degraden de forma elegante y sigan siendo observables (de estrategias de escalado en adelante). Aquí esas ideas se encuentran con las plataformas en las que hoy se ejecutan la mayoría de los sistemas. El ejemplo recurrente es la plataforma de reparto de comida que se usa a lo largo de todo el curso: las apps de clientes y de restaurantes, un servicio de pedidos, las cartas, los pagos, el despacho de repartidores y una intensa hora punta de la cena.

Cómo encajan las secciones

Las cuatro primeras secciones tratan de los fundamentos. Modelos de servicio y de despliegue en la nube expone quién gestiona qué con IaaS, contenedores como servicio, PaaS, FaaS y SaaS, el modelo de responsabilidad compartida, las razones reales de la nube privada, la híbrida y la multinube, qué entiende la CNCF por nativo de la nube, y cuándo alquilar no sale más barato que poseer. Contenedores y orquestación explica las imágenes y los estándares OCI, el plano de control y los nodos de Kubernetes, sus objetos principales y el bucle de reconciliación que está en su núcleo, con YAML real, y es honesta sobre cuándo Kubernetes es excesivo. La aplicación de doce factores recorre los doce factores con las salvedades actuales, los principios que van más allá de ellos y una auditoría de un servicio existente. Infraestructura como código trata las herramientas declarativas (Terraform, OpenTofu, CloudFormation, Bicep, Pulumi), el estado, la deriva, los módulos, la infraestructura inmutable, GitOps, la política como código y las pruebas.

Las cuatro secciones siguientes tratan del cambio, la escala, las personas y el dinero. Pipelines de CI/CD y estrategias de despliegue sigue un cambio desde el commit hasta producción, compara los despliegues gradual, azul-verde, canario y en sombra, y separa el despliegue del lanzamiento con feature flags. Multirregión, recuperación ante desastres y edge computing compara las estrategias de recuperación ante desastres por RTO, RPO y costo, explica las consecuencias para la consistencia de replicar datos entre regiones, y trata la arquitectura basada en células y las funciones en el borde. Ingeniería de plataformas trata la plataforma interna para desarrolladores como un producto con caminos dorados y salvaguardas, y muestra

cómo medir si ayuda. FinOps, arquitectura de costos y dependencia del proveedor convierte el costo en una cualidad arquitectónica, medida por pedido, y trata la dependencia del proveedor como un compromiso al que hay que poner precio y no como un pecado que haya que evitar.

Cada sección puede leerse por separado, pero el orden es deliberado: no se puede razonar sobre las estrategias de despliegue sin saber qué hace Kubernetes durante un despliegue progresivo, ni sobre la ingeniería de plataformas sin el código de infraestructura y los pipelines con los que se construye la plataforma.

Una idea recorre todas las secciones

La idea más importante de este capítulo es **el estado deseado declarado, reconciliado continuamente por la automatización**. Kubernetes mantiene el número de copias en ejecución igual al número que declaraste. El código de infraestructura describe redes y bases de datos, y una herramienta calcula y aplica la diferencia con lo que existe. GitOps mantiene el clúster igual a lo que hay en un repositorio de Git. Un pipeline de despliegue convierte cada cambio en la misma secuencia de pasos probada y repetible. La infraestructura inmutable sustituye en lugar de reparar. En cada caso, una persona cambia una descripción, que se revisa, se versiona y es auditable, y las máquinas hacen el resto. Esto es lo que permite a un equipo pequeño operar un sistema grande de forma segura: el cambio se vuelve rutinario y la recuperación pasa a ser volver a ejecutar algo en lugar de un acto de heroísmo.

La segunda idea es que cada opción del menú de este capítulo es un compromiso, y la elección correcta depende de la carga de trabajo y de la organización. Kubernetes es excelente para muchos servicios y muchos equipos, y un despilfarro para tres servicios y dos desarrolladores. Una configuración multirregión activo-activo compra minutos de tiempo de recuperación a un múltiplo del costo. Un equipo de plataforma solo ayuda si los desarrolladores quieren lo que construye. La nube es lo más barato para una carga variable, y el hardware propio puede superarla para una carga grande y estable. La dependencia del proveedor es un costo que hay que sopesar frente al costo de la portabilidad. Por eso el capítulo te da guías de decisión, ejemplos prácticos con cifras y los argumentos de ambos lados, en lugar de una única pila recomendada.

Cómo usar este capítulo

Quienes sean nuevos en las operaciones deberían leer primero los recuadros para principiantes y la sección del Capítulo 0 sobre servidores y la nube, y no preocuparse por memorizar nombres de herramientas: los productos cambian cada pocos años, mientras que los mecanismos (reconciliación, inmutabilidad, estado y deriva, entrega progresiva, objetivos de recuperación, costos unitarios) perduran. Los ingenieros con experiencia quizá prefieran empezar por los ejemplos prácticos y las guías de decisión, y después leer las secciones con

cuyos compromisos no estén de acuerdo. A lo largo del capítulo, proveedores como AWS, Microsoft Azure, Google Cloud y Cloudflare se mencionan como ejemplos, no como recomendaciones.

Le sigue el Capítulo 11, con la evaluación, la gobernanza y la modernización: cómo juzgar una arquitectura, incluida su infraestructura, y cómo cambiarla de forma segura a lo largo del tiempo.

SECCIÓN 10.1 DE 10.8 · INTERMEDIO · 34 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Modelos de servicio y de despliegue en la nube

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.7 Servidores y la nube
- 2.5 Disponibilidad y fiabilidad

Términos anteriores que utiliza: Máquina virtual (VM), Contenedor, Elasticidad, Failover (conmutación por error)

LA IDEA EN BREVE

Piensa en cómo consigue la gente un lugar donde vivir. Puedes comprar un terreno y construir una casa, alquilar un apartamento vacío y amueblarlo, alquilar uno amueblado o alojarte en un hotel. Cada paso deja más trabajo en manos de otro y te deja menos opciones; el hotel es estupendo para una semana y caro para diez años. Ejecutar software es igual: puedes tener tus propios servidores, alquilar máquinas vacías, alquilar una plataforma lista para usar o alquilar software terminado. Elijas lo que elijas, sigues cerrando tu propia puerta con llave: tus datos y quién puede acceder a ellos siguen siendo responsabilidad tuya.

LO QUE NECESITAS SABER

- NIST SP 800-145 define tres modelos de servicio (IaaS, PaaS, SaaS) y cuatro modelos de despliegue (público, privado, comunitario, híbrido); los contenedores y las funciones como servicio cubren los huecos entre ellos.
- Cada paso de IaaS hacia SaaS elimina trabajo operativo y control; la pregunta útil es qué tareas concretas (parches, escalado, copias de seguridad, failover, actualizaciones) siguen siendo tuyas.
- Según el modelo de responsabilidad compartida, el proveedor protege la nube en sí y el cliente protege lo que pone en ella; los datos, las identidades y la configuración son siempre responsabilidad del cliente.
- La nube privada, la híbrida y la multinube tienen razones legítimas (regulación, carga estable, migraciones, servicios distintivos, planes de salida), pero ejecutar cada carga de trabajo de forma portable entre nubes es costoso.
- Distribuye las cargas de trabajo en al menos dos zonas de disponibilidad, preferiblemente tres; averigua si cada dependencia es zonal, regional o global.

- La CNCF define nativo de la nube como un conjunto de prácticas para construir de forma programática y repetible sistemas poco acoplados, seguros, resilientes, gestionables, sostenibles y observables; se trata de cómo, no de dónde.
- La nube es lo más barato para una carga variable; para una carga grande y estable, el hardware propio puede salir más barato, como argumentó 37signals, una vez que se cuentan con honestidad el margen de capacidad, el personal y los descuentos por compromiso de uso.

De una definición a muchos modelos

El Capítulo 0 presentó servidores y la nube: la definición del NIST (Special Publication 800-145, 2011), con sus cinco características esenciales de autoservicio bajo demanda, amplio acceso a la red, agrupación de recursos, elasticidad rápida y servicio medido, y la idea de que los modelos de servicio delegan en un proveedor una parte mayor o menor de la pila. Esta sección lleva esas ideas a la profundidad que necesita un arquitecto para elegir dónde se ejecuta un sistema y defender esa elección.

El NIST define tres modelos de servicio (infraestructura, plataforma y software como servicio) y cuatro modelos de despliegue: **público** (abierto a cualquiera, propiedad de un proveedor), **privado** (para una sola organización, esté donde esté y lo opere quien lo opere), **comunitario** (compartido por organizaciones con inquietudes comunes, como los organismos públicos) e **híbrido** (dos o más de los anteriores, unidos de forma que los datos y las aplicaciones puedan moverse entre ellos). Desde entonces, el sector ha añadido las funciones y los contenedores como servicio a los modelos de servicio, y la multinube al vocabulario del despliegue.

Dónde se ejecuta el software forma parte de lo que el Capítulo 1 llamó la estructura de asignación de una arquitectura (definir la arquitectura): la correspondencia entre el software y el hardware, las redes y las organizaciones. Es significativo en el sentido del Capítulo 1 porque es costoso de revertir, y porque fija buena parte del costo de un sistema, su disponibilidad, su frontera de seguridad y la rapidez con la que sus equipos pueden lanzar cambios.

Modelos de servicio: quién gestiona qué

Los modelos de servicio forman un espectro. En un extremo alquilas capacidad desnuda y haces tú todo lo que hay por encima; en el otro alquilas una aplicación terminada y solo la configuras. Cada paso hacia la derecha elimina trabajo operativo y elimina opciones. La tabla añade los dos modelos que son posteriores al documento del NIST: los contenedores como servicio y las funciones como servicio.

Comparación de los modelos de servicio

Modelo	Tú despliegas y operas	El proveedor opera	Ejemplos	Encaje típico
IaaS	Máquinas virtuales: sistema operativo, parches, runtime, aplicación, reglas de escalado	Hardware, red, hipervisor, centro de datos	Amazon EC2, Azure Virtual Machines, Google Compute Engine	Software que necesita un sistema operativo o un kernel concretos, aplicaciones heredadas trasladadas tal cual, control total
CaaS (contenedores como servicio)	Imágenes de contenedor y su configuración; con Kubernetes gestionado, también la elección y las actualizaciones de los nodos de trabajo	El plano de control de contenedores, a menudo también los nodos; con contenedores serverless, todos los servidores	Amazon EKS y ECS, Azure Kubernetes Service y Container Apps, Google Kubernetes Engine y Cloud Run	Muchos servicios construidos como contenedores, equipos que quieren un empaquetado portable
PaaS	El código de la aplicación y su configuración	Sistema operativo, runtime, escalado, balanceo de carga, a menudo el logging	Heroku, Azure App Service, Google App Engine, AWS Elastic Beanstalk	Aplicaciones web y API estándar que se ajustan a las convenciones de la plataforma
FaaS	Funciones individuales y los eventos que las activan	Todo lo demás, incluido el escalado hasta cero; facturación por invocación y por duración	AWS Lambda, Azure Functions, Google Cloud Run functions, Cloudflare Workers	Trabajo dirigido por eventos, irregular o de poco volumen, en pasos cortos
SaaS	Configuración, usuarios, permisos y tus datos	La aplicación completa	Microsoft 365, Salesforce, Workday, Zendesk	Capacidades genéricas que no diferencian al negocio

En la práctica, las fronteras se difuminan. Las plataformas de contenedores serverless, como Google Cloud Run y Azure Container Apps, ejecutan una imagen de contenedor igual que un PaaS ejecuta código, y escalan hasta cero como una función. Kubernetes gestionado puede ser

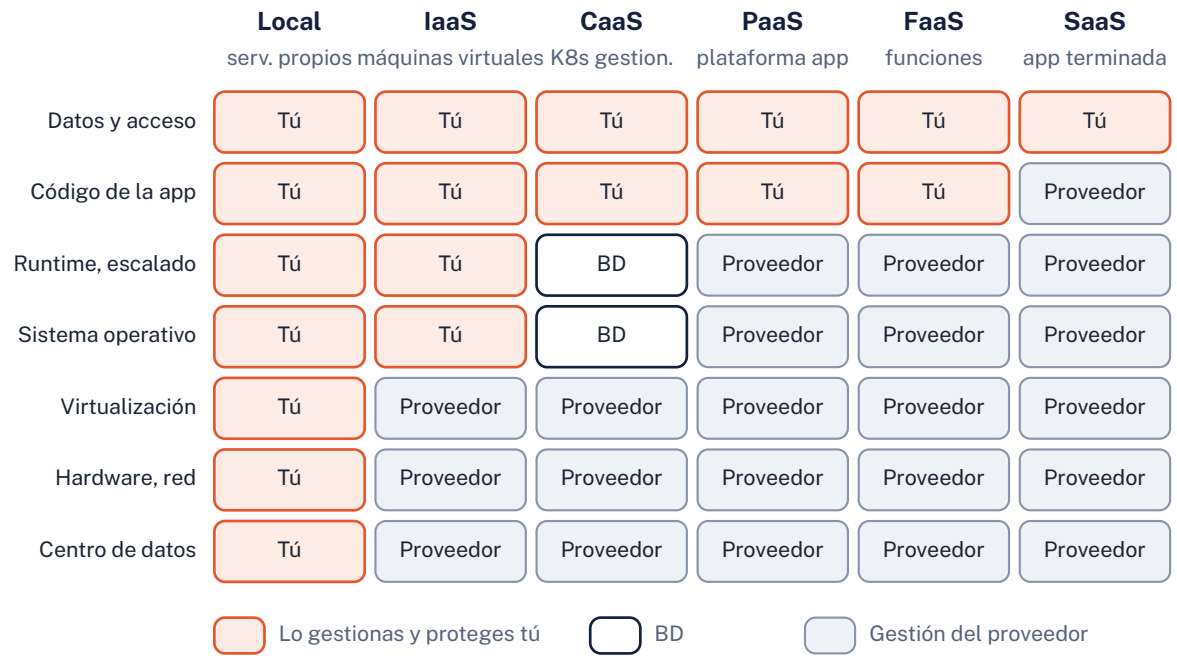
cualquier cosa, desde un plano de control alojado, en el que tú sigues aplicando parches y actualizando las máquinas de trabajo, hasta un modo totalmente gestionado en el que el proveedor también opera los nodos. La pregunta útil nunca es qué etiqueta se aplica, sino qué tareas concretas siguen siendo tuyas: los parches, las decisiones de escalado, la planificación de capacidad, las copias de seguridad, el failover, las actualizaciones y la respuesta de guardia.

Una segunda distinción útil es la unidad de despliegue y de facturación. IaaS factura las máquinas por segundo o por hora, estén ocupadas o no; FaaS factura las invocaciones y el tiempo de ejecución y no cuesta nada cuando está inactivo; PaaS y los contenedores serverless se sitúan en medio. El Capítulo 6 explica por qué el precio por solicitud es más barato que la capacidad siempre encendida por debajo de un volumen de cruce, y más caro por encima de él (arquitecturas serverless). Es la forma de la carga de una carga de trabajo, no su tamaño, lo que decide en qué lado del punto de cruce se encuentra.

El modelo de responsabilidad compartida

Todos los grandes proveedores publican un modelo de responsabilidad compartida. AWS lo resume diciendo que el proveedor es responsable de la seguridad *de* la nube y el cliente de la seguridad *en* la nube. El proveedor protege los edificios, el hardware, la red, el hipervisor y los servicios que opera; el cliente protege lo que construye y configura por encima. Dónde cae la línea depende del modelo de servicio, como muestra el diagrama.

Quién es responsable de qué, según el modelo de servicio



Al avanzar hacia la derecha se ceden capas inferiores al proveedor, pero los datos, las identidades y la configuración del acceso se quedan con el cliente en todos los modelos.

Hay tres cosas que nunca se transfieren. El cliente siempre es dueño de sus datos (qué se almacena, dónde, durante cuánto tiempo, quién puede leerlo), de sus identidades y permisos, y de la configuración de los servicios que usa. Muchas brechas en la nube han sido fallos de configuración del cliente y no fallos del proveedor: almacenamiento de objetos que cualquiera podía leer, claves de administrador incluidas en repositorios públicos, permisos demasiado amplios. En 2019, por ejemplo, un atacante aprovechó un firewall de aplicaciones web mal configurado en Capital One para obtener credenciales de la nube y copiar datos relativos a más de 100 millones de personas en Estados Unidos y Canadá; la infraestructura del proveedor se comportó tal como estaba diseñada.

Para un arquitecto, el modelo es una lista de comprobación de trabajo que hay que asignar a alguien. Si un equipo traslada la base de datos de pedidos de la plataforma de reparto de comida de una máquina virtual a un servicio de base de datos gestionada, la aplicación de parches al motor de base de datos pasa al proveedor, pero no así elegir una ventana de mantenimiento, probar las actualizaciones de versión mayor, conceder accesos con mínimo privilegio, activar el cifrado y comprobar que las copias de seguridad se pueden restaurar. El Capítulo 8 trata en profundidad los principios de seguridad que hay detrás de estas obligaciones (principios de seguridad para arquitectos).

Los modelos de despliegue y las razones reales que hay detrás

Las organizaciones rara vez eligen un modelo de despliegue por una sola razón, y la razón declarada no siempre es la real. Ayuda separar los motivos que resisten un examen atento de los que no.

Pública, privada, híbrida y multinube

1. **Nube pública**
   Se elige por la elasticidad, la velocidad (un entorno nuevo en minutos, no una compra de hardware en meses), un amplio catálogo de servicios gestionados y el alcance global. Encaja con cargas variables, productos nuevos cuya demanda se desconoce y equipos que quieren dedicar su esfuerzo al producto y no a los servidores.

2. **Nube privada**
   Se elige por el control regulatorio o contractual, una carga estable y predecible, la inversión ya hecha en un centro de datos o una latencia muy baja hacia equipos instalados in situ. Una nube privada significa aprovisionamiento de autoservicio basado en API sobre infraestructura dedicada (por ejemplo, con OpenStack, VMware o la edición para instalaciones propias de la nube pública de un proveedor), no simplemente una sala de servidores virtualizada en la que las solicitudes siguen pasando por una cola de tickets.

3 **Nube híbrida**

Suele ser el resultado de una migración gradual, de datos que deben quedarse en las propias instalaciones (un libro mayor en un mainframe, sistemas de fábricas u hospitales) o de ubicaciones en el borde, como tiendas y almacenes. Desbordar desde un centro de datos privado hacia la nube pública en los momentos de máxima demanda se promete a menudo, pero rara vez es práctico, porque los datos que necesita la carga de trabajo se quedan atrás y copiarlos cuesta tiempo y dinero.

4 **Multinube**

Usar dos o más proveedores públicos. Entre las razones defendibles están usar un servicio distintivo de cada proveedor, las adquisiciones que llegan en otra nube, la capacidad de negociación y una regulación que exija planes de salida creíbles. En la UE, el Reglamento de Resiliencia Operativa Digital (DORA), aplicable desde enero de 2025, exige a las entidades financieras tener estrategias de salida para los proveedores críticos de TIC. Ejecutar cada carga de trabajo de forma portable en varias nubes es un objetivo distinto y mucho más caro.

El error habitual es adoptar la multinube como una cobertura abstracta. Hacer que una carga de trabajo se ejecute de forma idéntica en dos proveedores significa usar solo los servicios que ofrecen ambos, construir y probar dos conjuntos de código de infraestructura, contar con personal con conocimientos de ambos y, a menudo, pagar cargos de transferencia de datos entre ellos. Esa es la trampa del mínimo común denominador: pagas cada día el costo de la portabilidad a cambio de una salida que quizá nunca uses. La sección 10.8 trata la dependencia del proveedor como un compromiso al que hay que poner precio, no como un pecado que haya que evitar a toda costa (FinOps, arquitectura de costos y dependencia del proveedor).

Las regiones y las zonas de disponibilidad como datos de entrada del diseño

Una región es un área geográfica con su propio conjunto independiente de centros de datos; una zona de disponibilidad es uno o varios centros de datos dentro de una región con alimentación eléctrica, refrigeración y red independientes, situados lo bastante lejos de los demás como para que un incendio, una inundación o un corte de luz local afecte solo a uno, y lo bastante cerca como para que los viajes de ida y vuelta duren como mucho unos pocos milisegundos. El Capítulo 0 explicó la idea. Para el diseño, importan tres puntos más.

Primero, cada servicio gestionado tiene un ámbito de fallo: los recursos *zonales* (una sola máquina virtual, un disco) mueren con su zona; los servicios *regionales* (la mayoría de las bases de datos gestionadas en modo multizona, el almacenamiento de objetos, las colas) los replica

el proveedor entre zonas; unos pocos servicios *globales* (DNS, identidad, algunos servicios de distribución de contenidos y de balanceo de carga) abarcan varias regiones, lo que hace que sus raras caídas tengan un gran alcance. Saber qué ámbito tiene cada dependencia es el primer paso de cualquier análisis de disponibilidad (consulta disponibilidad y fiabilidad).

Segundo, distribuirse entre zonas es la opción por defecto barata y de gran valor. Ejecutar los servicios sin estado de la plataforma de reparto de comida en tres zonas detrás de un balanceador de carga, con una base de datos que mantiene una réplica de reserva síncrona en una segunda zona, sobrevive a la pérdida de un centro de datos con failover automático y un pequeño costo en latencia. Tres zonas en lugar de dos importan para todo lo que use quórums, como etcd en Kubernetes o una base de datos basada en consenso, porque debe sobrevivir una mayoría. Sin embargo, el tráfico entre zonas no siempre es gratuito: la mayoría de los grandes proveedores cobran por los datos que se mueven entre zonas, así que los servicios muy conversadores repartidos entre zonas añaden una línea visible a la factura.

Tercero, las regiones no son idénticas. Las regiones más nuevas o más pequeñas pueden carecer de algunos servicios, algunas tienen menos zonas, los precios varían según la región y la jurisdicción legal sigue a la región. La sección 10.6 trata las múltiples regiones, la recuperación ante desastres y la residencia de datos (multirregión, recuperación ante desastres y edge computing).

SI ESTO ES NUEVO PARA TI

Paso a paso: qué se detiene cuando falla una zona

La idea más difícil de esta sección es el ámbito de fallo: cada parte de un sistema vive en una zona, en una región o en todo el mundo, y eso decide qué se detiene cuando algo falla. Sigue un fallo de zona a través de la plataforma de reparto de comida.

- La plataforma ejecuta sus servicios sin estado en tres zonas de disponibilidad detrás de un balanceador de carga, con una base de datos de pedidos gestionada que mantiene una réplica de reserva síncrona en una segunda zona. Los clientes la encuentran a través del DNS.
- Cada pieza tiene un ámbito. Una sola máquina virtual es zonal: vive en una zona. La base de datos multizona es regional: el proveedor la mantiene copiada entre zonas. El DNS es global: abarca varias regiones.
- Una inundación deja una zona fuera de servicio durante la hora punta de la cena. Las copias de los servicios en esa zona se detienen con ella. El balanceador de carga deja de enviarles solicitudes, y las copias de las otras dos zonas siguen funcionando y escalan si lo necesitan.
- Si la copia principal de la base de datos estaba en la zona inundada, el proveedor cambia a la réplica de reserva tras una breve pausa: un failover automático. Como la réplica de reserva es síncrona, cada escritura le llegó antes de que la base de datos la confirmara, como explicó el Capítulo 7, así que los pedidos confirmados sobreviven. Los pedidos que estaban a medias durante la pausa pueden fallar y tener que reintentarse.
- Los servicios globales fallan de otra manera. Sus caídas son raras, pero distribuirse entre zonas, o incluso entre regiones, no te protege de una de ellas, porque todas las zonas y regiones dependen del mismo servicio.
- **La lección:** conoce el ámbito de cada dependencia antes de confiar en el diseño. Las piezas zonales necesitan copias en más de una zona; las regionales sobreviven a la pérdida de una zona, pero no a la de toda su región; las globales fallan rara vez, pero en todas partes a la vez.

Servicios gestionados frente a autogestionados

Un servicio gestionado es un componente (una base de datos, una cola, una caché, un motor de búsqueda, un plano de control de Kubernetes) que el proveedor instala, opera, actualiza con parches, respalda con copias de seguridad y escala por ti. La alternativa es ejecutar tú mismo el mismo software de código abierto en máquinas virtuales o en contenedores. El compromiso es trabajo y riesgo frente a control, costo y portabilidad.

Los servicios gestionados ganan cuando el componente no es diferenciador: toda empresa de reparto de comida necesita una base de datos PostgreSQL fiable, pero ninguna gana clientes por cómo aplica los parches a su base de datos. En términos de diseño guiado por el dominio, son subdominios genéricos, los que conviene comprar en lugar de construir. Una base de datos gestionada ofrece failover multizona, recuperación a un momento dado y parches de versiones menores que, de otro modo, necesitarían un equipo cualificado de guardia. La autogestión gana cuando necesitas una versión, una extensión o un parámetro de ajuste que el proveedor no ofrece, cuando los límites o cuotas del proveedor no encajan, cuando el sobreprecio de lo gestionado se vuelve grande a escala, o cuando la portabilidad entre entornos importa más que la comodidad.

Gestionado no significa libre de mantenimiento. El proveedor impondrá actualizaciones de versión mayor según su propio calendario cuando las versiones antiguas lleguen al fin del soporte, cambiará valores por defecto, impondrá cuotas que descubrirás en el peor momento y tendrá caídas que no puedes arreglar, solo esperar a que pasen. Tú sigues siendo responsable del diseño del esquema, del rendimiento de las consultas, de las decisiones de capacidad, del costo y de la prueba periódica que demuestra que una restauración funciona. Trata cada servicio gestionado como una dependencia con sus propios modos de fallo y su propio costo de salida.

Fundamentos de la nube: landing zones y la jerarquía de cuentas

Antes de que entre la primera carga de trabajo, un entorno en la nube necesita fundamentos, y es mucho más fácil ponerlos pronto que añadirlos después. Una **landing zone** (zona de aterrizaje) es el entorno preparado en el que se despliegan las cargas de trabajo. AWS la describe como un entorno multicuenta bien diseñado, escalable y seguro. La landing zone de Azure de Microsoft separa una *landing zone de plataforma*, la base central de gobernanza, seguridad y recursos compartidos, de las *landing zones de carga de trabajo*, donde los equipos de producto despliegan dentro de esos guardrails (salvaguardas). En cualquier caso, debe crearse y cambiarse como código (Sección 10.4), y suele ser lo primero de lo que se hace cargo un equipo de plataforma (Sección 10.7).

La decisión más difícil de cambiar más adelante es la jerarquía. Cada proveedor tiene una unidad básica de aislamiento y un árbol que agrupa esas unidades: las cuentas de AWS en unidades organizativas; las suscripciones de Azure en grupos de administración; los proyectos de Google Cloud, opcionalmente en carpetas, bajo una organización. El árbol importa porque las políticas se vinculan a él. Las políticas de Google Cloud se heredan hacia abajo por el árbol, y las suscripciones de Azure heredan las políticas establecidas en sus grupos de administración. Las políticas de control de servicios de AWS fijan los permisos máximos disponibles en las cuentas miembro; no conceden nada por sí mismas y no afectan a la cuenta de administración, que es un motivo para mantener las cargas de trabajo fuera de ella.

Una cuenta o suscripción es un límite mucho más fuerte que una etiqueta o una convención de nombres. En AWS, una cuenta no tiene acceso a otra por defecto, las cuotas se aplican por cuenta y los costos se asignan directamente a ella. Así que separa por carga de trabajo y por entorno desde el principio, porque mover más adelante un sistema en vivo entre cuentas es un proyecto de migración. AWS cobra por los recursos usados, no por el número de cuentas, aunque muchas cuentas sí requieren automatización.

Para la plataforma de reparto de comida, un árbol inicial sensato, adaptado de la estructura de ejemplo de AWS, tiene cuatro ramas. *Seguridad* contiene una cuenta de archivo de logs que los equipos de producto no pueden alterar y una cuenta de auditoría. *Infraestructura* contiene las redes y los servicios compartidos. *Cargas de trabajo* contiene cuentas separadas de producción y de no producción para pedidos, despacho y pagos, con políticas más estrictas en la rama de pagos. *Sandbox* es para experimentos seguros. Los guardrails (salvaguardas) en la parte superior del árbol podrían denegar las regiones que no estén aprobadas e impedir que alguien desactive el logging.

La lista de áreas de diseño de Azure es una lista de comprobación útil incluso en otras nubes: facturación y tenencia, identidad, organización de recursos, red, seguridad, gestión, gobernanza y automatización. No necesitas construirlo todo de una vez; AWS señala que puedes empezar con un subconjunto de sus unidades de ejemplo y añadir más después. El fallo habitual es lo contrario de tener poca estructura: guardrails tan estrictos, y cuentas nuevas tan lentas de obtener, que los equipos los esquivan. Ofrece las cuentas nuevas como autoservicio a través de la plataforma, con los guardrails ya aplicados.

Fundamentos de red: VPC, conectividad privada y tráfico de salida

Una **nube privada virtual** (VPC; una red virtual en Azure) es tu red privada dentro de un proveedor. Planifica primero sus direcciones. En AWS, el rango con el que se crea una VPC no se puede redimensionar ni eliminar, solo ampliar, y dos VPC con rangos solapados no pueden conectarse por peering. Azure da el mismo consejo para sus redes y para los sitios locales. Así que asigna a cada red un rango que no se solape a partir de un único plan central, con margen para crecer.

Las subredes dividen una VPC y, en AWS, cada una está en una sola zona de disponibilidad, de modo que la resiliencia implica repetirlas por zona. Lo que hace pública o privada una subred es su enrutamiento: una subred pública tiene una ruta a una puerta de enlace de internet, una subred privada no la tiene y necesita un dispositivo NAT para salir, y una subred aislada no tiene ninguna ruta más allá de la VPC. La plataforma de reparto de comida podría ejecutar tres niveles en tres zonas: balanceadores de carga en subredes públicas, servicios en privadas y bases de datos en aisladas. Las subredes son solo zonas de grano grueso; el control más fino viene de los métodos de identidad y segmentación de la Sección 8.5.

A continuación, la conectividad. El peering une dos redes, pero no es transitivo: si A tiene peering con B y con C, B sigue sin poder llegar a C a través de A. A medida que se multiplican las redes, los equipos pasan a un hub como AWS Transit Gateway o al diseño hub-spoke de Azure, en el que una red central contiene las puertas de enlace, el firewall y el DNS compartidos. El hub se convierte entonces en una dependencia compartida, por lo que Microsoft recomienda uno por región. Para llegar a un centro de datos, una VPN de sitio a sitio tiende túneles cifrados sobre internet, mientras que AWS Direct Connect o Azure ExpressRoute proporcionan un enlace dedicado; AWS señala que Direct Connect no está cifrado por defecto. Para llegar a un servicio en lugar de a toda una red, usa AWS PrivateLink, Azure Private Link o Google Cloud Private Service Connect, que lo exponen en una dirección privada sin peering ni acceso a internet.

Por último, controla lo que sale. Muchos parques de sistemas filtran con cuidado el tráfico entrante y dejan salir cualquier cosa, y sin embargo el acceso saliente es la vía por la que salen los datos robados, también mediante instrucciones de IA inyectadas (Sección 12.6). Una pasarela NAT permite que los servicios privados inicien conexiones salientes e impide que las inicien desde fuera, pero solo traduce direcciones y no elige destinos. Para elegirlos hace falta un firewall de salida, como AWS Network Firewall o Azure Firewall con reglas de dominio, situado de modo que todo el tráfico saliente pase por él. Así, el servicio de pedidos de la plataforma solo llega a sus proveedores de pagos, mapas y mensajería, y todo lo demás se rechaza y se registra.

Las rutas de salida también tienen su lado de fiabilidad y de costos. Una pasarela NAT estándar de AWS reside en una zona, de modo que, si las cargas de trabajo de otras zonas la comparten, pierden el acceso a internet cuando esa zona falla; AWS aconseja una por zona y ha añadido una opción regional, así que comprueba la versión vigente. Las pasarelas NAT se cobran por hora y por gigabyte, mientras que los gateway endpoints de AWS para S3 no tienen cargo por hora ni por procesamiento de datos, de modo que enviar por ahí el tráfico de almacenamiento es un ahorro fácil (Sección 10.8).

Qué significa nativo de la nube

La Cloud Native Computing Foundation (CNCF), que forma parte de la Linux Foundation y es la casa de Kubernetes, publica la definición más citada de nativo de la nube. La versión 1.0 (2018) describía tecnologías que permiten a las organizaciones construir y ejecutar aplicaciones escalables en entornos dinámicos como las nubes públicas, privadas e híbridas, y citaba como ejemplos típicos los contenedores, las service meshes, los microservicios, la infraestructura inmutable y las API declarativas. La versión 1.1, aprobada en 2024, desplaza el énfasis de las tecnologías a las prácticas: desarrollar, construir y desplegar cargas de trabajo de forma programática y repetible a escala, caracterizadas por “sistemas poco acoplados que interoperan de una manera segura, resiliente, gestionable, sostenible y observable”.

De ahí se siguen dos consecuencias. Nativo de la nube describe *cómo* se construye y se opera un sistema, no *dónde*: un clúster de Kubernetes en un centro de datos privado, aprovisionado a partir de código y desplegado muchas veces al día, es nativo de la nube; una máquina virtual configurada a mano y copiada sin cambios en una nube pública no lo es. Y nativo de la nube es un medio, no un fin. Sus prácticas (automatización, componentes desechables, configuración declarativa, elasticidad, observabilidad) compensan cuando un sistema necesita cambios frecuentes, escala variable y alta disponibilidad; añaden costo y complejidad a una herramienta interna estable que usan veinte personas.

Las migraciones se describen a menudo con las siete R (consulta la Sección 11.6), que vale la pena conocer porque marcan las expectativas: *rehost* (realojar, lift and shift, sin cambios), *relocate* (reubicar máquinas virtuales o plataformas enteras en la nube sin cambiarlas), *replatform* (cambios pequeños, como pasar a una base de datos gestionada), *refactor* o rediseñar la arquitectura (rediseño para operar de forma nativa en la nube), *repurchase* (sustituir por SaaS), *retain* (dejarlo donde está) y *retire* (apagarlo). Realojar es rápido, pero suele aportar pocos de los beneficios de la nube y, ejecutado en máquinas siempre encendidas dimensionadas para el pico, puede costar más que el centro de datos que se dejó.

Cuando la nube no es más barata

El contraejemplo más conocido es 37signals, la empresa que está detrás de Basecamp y HEY. En octubre de 2022, su director de tecnología, David Heinemeier Hansson, anunció que dejaba la nube, alegando una factura anual de AWS de más de 3 millones de USD para cargas de trabajo con una carga estable y predecible. Compró servidores Dell por unos 700 000 USD, los instaló en un espacio de colocation alquilado y sacó sus aplicaciones de la computación de AWS durante 2023; informó de que esto redujo su factura de nube en unos 2 millones de USD al año. En 2025 anunció la segunda etapa: trasladar unos 18 petabytes de archivos de Amazon S3 a cabinas de almacenamiento por valor de unos 1.5 millones de USD para sustituir una factura de S3 de unos 1.5 millones de USD al año, con el objetivo declarado de cerrar su cuenta de AWS. La empresa ha proyectado un ahorro de unos 10 millones de USD en cinco años. Sacar así cargas de trabajo de la nube se llama repatriación desde la nube.

Ambos lados del argumento merecen ser escuchados. A favor del razonamiento de 37signals: su carga es estable, así que ganaba poco con la elasticidad; ya contaba con ingenieros de operaciones experimentados e informó de que no necesitó personal adicional; los proveedores de colocation se ocupan de la alimentación eléctrica, la refrigeración y la seguridad física; y los servidores pueden usarse durante cinco años o más, así que su costo por año de computación queda muy por debajo del alquiler bajo demanda. En el otro lado: 37signals es un negocio maduro con una demanda predecible, cosa que la mayoría de los productos nuevos no son; la comparación depende mucho de cuál habría sido la factura de la nube con descuentos por compromiso de uso y ajuste de tamaño; poseer hardware implica gasto de capital, plazos de

entrega de semanas o meses y capacidad comprada por adelantado; y un equipo más pequeño sin conocimientos profundos de operaciones tendría que contratarlos, lo que puede borrar el ahorro. Los críticos también señalan que las cifras de ahorro proceden de la propia empresa.

EJEMPLO PRÁCTICO

Ejemplo práctico: alquilar o poseer la capacidad de cómputo para la plataforma de reparto de comida

Todos los precios son supuestos ilustrativos elegidos para mostrar el método, no presupuestos reales. La capacidad bajo demanda en la nube cuesta £0.040 por hora de vCPU; un compromiso de tres años cuesta £0.024 (un descuento del 40%); el hardware propio en colocation cuesta £0.012 por hora de vCPU, incluidos la electricidad y el espacio, pero necesita un 30% de margen de capacidad y tiempo de personal. La plataforma necesita 40 vCPU todo el día, que suben a 120 vCPU durante la hora punta de la cena de cuatro horas (unas 122 horas al mes). Un mes tiene 730 horas.

A: nube, dimensionada para el pico, siempre encendida	120 vCPU × 730 h × £0.040 = **£3504 al mes**
B: nube, base comprometida más pico con autoescalado	40 × 730 × £0.024 = £701; más 80 × 122 × £0.040 = £390; total **£1091 al mes**
C: hardware propio dimensionado para el pico	120 × 1.3 × 730 × £0.012 = £1367; más una cuarta parte de un ingeniero a £80 000 al año = £1667; total **£3034 al mes**
Ahora a mayor escala: 2000 vCPU constantes, sin hora punta	Nube con compromiso: 2000 × 730 × £0.024 = £35 040. Propio: 2000 × 1.3 × 730 × £0.012 = £22 776 más un ingeniero (£6667) = **£29 443**, alrededor de un 16% más barato

Para una carga de trabajo pequeña e irregular, domina la elasticidad: la opción B cuesta menos de un tercio de la opción A y alrededor de un tercio de la opción C, porque solo paga la capacidad de pico durante la hora punta.

Para una carga de trabajo grande y plana, los costos fijos de la propiedad se reparten entre más capacidad y la propiedad empieza a ganar, que es la situación de 37signals. El ahorro es real, pero menor de lo que sugieren las comparaciones de titular una vez que se cuentan el margen de capacidad, el personal y el esfuerzo de migración.

El método importa más que las cifras: modela la forma de la carga, incluye a las personas y el margen de capacidad, compara con el mejor precio de la nube (con compromisos), no con el precio de lista, y repite el cálculo a medida que cambia el negocio.

GUÍA DE DECISIÓN

Guía de decisión: elegir un modelo de alojamiento para una carga de trabajo

Aplícala a cada carga de trabajo, no una sola vez para toda la organización: una misma empresa usa con buen criterio SaaS para las nóminas, funciones para redimensionar imágenes y contenedores para sus servicios centrales.

Q1 **¿La capacidad es genérica, con un producto en el mercado que encaja lo suficiente (correo electrónico, nóminas, CRM, pagos)?**

- **Sí** Compra SaaS e intégralo a través de su API; dedica el esfuerzo de diseño a la integración y al plan de salida.
- **No** Continúa: necesitas ejecutar tu propio software.

Q2 **¿La ley, un contrato o la ubicación física descartan la nube pública para esta carga de trabajo (normas estrictas de soberanía, equipos in situ, bucles de control de milisegundos)?**

- **Sí** Usa una nube privada, colocation o un despliegue en el borde in situ, conectado a servicios de la nube pública como un modelo híbrido donde esté permitido.
- **No** Continúa con las opciones de nube pública.

Q3 **¿El trabajo está dirigido por eventos, es de corta duración e irregular o de poco volumen (webhooks, procesamiento de archivos, tareas programadas)?**

- **Sí** Usa funciones como servicio o una plataforma de contenedores serverless; comprueba los límites de arranque en frío, de duración y de costo a gran volumen.
- **No** Continúa: se trata de un servicio de larga duración.

Q4 **¿Encaja el modelo de un PaaS o de una plataforma de contenedores serverless (servicio HTTP sin estado, runtime estándar, pocas necesidades especiales)?**

- **Sí** Úsalo. Es lo que más trabajo operativo le quita a un equipo pequeño.
- **No** Continúa: necesitas más control.

Q5 **¿Ejecutas muchos servicios en varios equipos, y tienes (o financiarás) un equipo capaz de operar Kubernetes?**

- **Sí** Usa Kubernetes gestionado (CaaS), idealmente detrás de una plataforma interna (sección 10.7).
- **No** Usa máquinas virtuales con bases de datos gestionadas y grupos de autoescalado, construidos a partir de código de infraestructura.

Q6 **¿La carga es grande, estable y predecible durante años, con conocimientos de operaciones en la propia empresa?**

Sí Compara el precio del hardware propio o en colocation con el de la capacidad comprometida en la nube; la repatriación puede compensar.

No Sigue con capacidad alquilada y usa descuentos por compromiso de uso para la base estable.

ATENCIÓN

Errores habituales al elegir modelos de nube

- Realojar todo sin cambios en máquinas virtuales siempre encendidas dimensionadas para el pico, y luego concluir que la nube es cara.
- Adoptar la multinube como cobertura general, pagando el costo de la portabilidad en cada carga de trabajo en lugar de planificar una salida para las pocas que la necesitan.
- Suponer que el proveedor hace copias de seguridad, cifra y restringe el acceso a tus datos por defecto; en todos los modelos son decisiones tuyas.
- Elegir Kubernetes antes de que haya un equipo para operarlo o suficientes servicios para justificarlo.
- Ignorar los cargos por transferencia de datos, sobre todo entre zonas, regiones y proveedores, al dibujar el diagrama de despliegue.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ Un equipo traslada su base de datos de una VM que gestionaba él mismo a un servicio de base de datos gestionada. Nombra dos responsabilidades que pasan al proveedor y dos que se quedan con el equipo.

El proveedor asume la aplicación de parches al motor, el hardware y, normalmente, las copias de seguridad automáticas y el failover entre zonas. El equipo conserva el diseño del esquema y de las consultas, los permisos de acceso, las decisiones de cifrado, la capacidad y el costo, las pruebas de restauración y la planificación de las actualizaciones de versión mayor.

▼ ¿Por qué un sistema basado en quórum, como etcd, prefiere tres zonas de disponibilidad a dos?

Un quórum necesita una mayoría. Con dos zonas, perder la zona que tiene la mayoría detiene el sistema; con tres, puede fallar cualquier zona y sobrevive una mayoría.

▼ **¿Es nativa de la nube una máquina virtual construida a mano y copiada sin cambios en una nube pública?**

No. Nativo de la nube describe prácticas como la automatización, la configuración declarativa, los componentes desechables y la observabilidad, no una ubicación. Realojar traslada el sistema sin adoptar esas prácticas.

▼ **¿Por qué podría una misma organización usar con buen criterio SaaS, funciones y Kubernetes a la vez?**

El modelo de alojamiento debe elegirse para cada carga de trabajo: las capacidades genéricas se compran como SaaS, las tareas irregulares dirigidas por eventos encajan con las funciones, y los servicios centrales de larga duración pueden justificar contenedores en Kubernetes.

▼ **Inténtalo: Ilustrativo: bajo demanda cuesta £0.05 por vCPU-hora, y con compromiso £0.03. Necesitas 20 vCPU siempre, más otras 60 durante cinco horas al día, en un mes de 30 días. Compara todo bajo demanda, comprometer la base y comprometer todo para el pico.**

Base: 20 x 720 = 14 400 horas de vCPU; extra de pico: 60 x 150 = 9000. Todo bajo demanda: 23 400 x £0.05 = £1170. Comprometer la base: 14 400 x £0.03 + 9000 x £0.05 = £882. Comprometer 80 vCPU todo el mes: 80 x 720 x £0.03 = £1728. Comprométete solo con la base segura y deja que la capacidad bajo demanda cubra la hora punta. El hardware propio solo tendría sentido con una carga grande y constante.

FUENTES

- NIST SP 800-145: The NIST Definition of Cloud Computing (2011): https://csrc.nist.gov/pubs/sp/800/145/final
- CNCF Technical Oversight Committee: Cloud Native Definition: https://github.com/cncf/toc/blob/main/DEFINITION.md
- AWS: Shared Responsibility Model: https://aws.amazon.com/compliance/shared-responsibility-model/
- Microsoft Learn: Shared responsibility in the cloud: https://learn.microsoft.com/en-us/azure/security/fundamentals/shared-responsibility
- The Register (2025): 37signals is completing its on-prem move: https://www.theregister.com/2025/05/09/37signals_cloud_repatriation_storage_savings
- Designing an AWS Control Tower landing zone (AWS Prescriptive Guidance): https://docs.aws.amazon.com/prescriptive-guidance/latest/designing-control-tower-landing-zone/introduction.html

- Sample account structure and OUs (AWS Prescriptive Guidance): https://docs.aws.amazon.com/prescriptive-guidance/latest/designing-control-tower-landing-zone/account-structure.html
- What is an Azure landing zone? (Microsoft Cloud Adoption Framework): https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/
- Azure landing zone design areas (Microsoft): https://learn.microsoft.com/en-us/azure/cloud-adoption-framework/ready/landing-zone/design-areas
- Organizing your AWS environment using multiple accounts (AWS whitepaper): https://docs.aws.amazon.com/whitepapers/latest/organizing-your-aws-environment/organizing-your-aws-environment.html
- Service control policies (AWS Organizations): https://docs.aws.amazon.com/organizations/latest/userguide/orgs_manage_policies_scps.html
- Resource hierarchy (Google Cloud): https://docs.cloud.google.com/resource-manager/docs/cloud-platform-resource-hierarchy
- VPC CIDR blocks (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/vpc-cidr-blocks.html
- Subnets for your VPC (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/configure-subnets.html
- VPC peering basics (AWS): https://docs.aws.amazon.com/vpc/latest/peering/vpc-peering-basics.html
- What is AWS Transit Gateway?: https://docs.aws.amazon.com/vpc/latest/tgw/what-is-transit-gateway.html
- Hub-spoke network topology in Azure (Microsoft): https://learn.microsoft.com/en-us/azure/architecture/networking/architecture/hub-spoke
- AWS Direct Connect (Amazon VPC connectivity options whitepaper): https://docs.aws.amazon.com/whitepapers/latest/aws-vpc-connectivity-options/aws-direct-connect.html
- What is AWS PrivateLink?: https://docs.aws.amazon.com/vpc/latest/privatelink/what-is-privatelink.html
- Private Service Connect (Google Cloud): https://docs.cloud.google.com/vpc/docs/private-service-connect
- NAT gateways (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/vpc-nat-gateway.html
- NAT gateway basics (AWS): https://docs.aws.amazon.com/vpc/latest/userguide/nat-gateway-basics.html
- Amazon VPC pricing: https://aws.amazon.com/vpc/pricing/
- What is AWS Network Firewall?: https://docs.aws.amazon.com/network-firewall/latest/developerguide/what-is-aws-network-firewall.html

SECCIÓN 10.2 DE 10.8 · AVANZADO · 26 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Contenedores y orquestación con Kubernetes

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.7 Servidores y la nube
- 9.1 Escalado
- 9.2 Balanceo de carga

Términos anteriores que utiliza: Contenedor, Autoescalado, Zona de disponibilidad, PaaS (plataforma como servicio)

LA IDEA EN BREVE

Piensa en un gran puerto. Las mercancías llegan en contenedores de transporte estándar, así que cualquier grúa, barco o camión puede manejarlos sin saber qué hay dentro. El capitán del puerto no mueve las cajas en persona; tiene un tablero que indica qué debe estar dónde, y su personal compara constantemente el muelle con el tablero y corrige cualquier diferencia: una caja que falta, una grúa averiada, un barco con retraso. Los contenedores de software son las cajas estándar, y Kubernetes es la oficina del capitán del puerto: anotas lo que debería estar en ejecución, y él sigue comprobando y corrigiendo hasta que la realidad coincide.

LO QUE NECESITAS SABER

- Una imagen de contenedor es una pila de capas direccionadas por contenido más una configuración y un manifiesto; las etiquetas (tags) pueden moverse, los digests no, así que en producción deben ejecutarse imágenes fijadas por digest.
- Las especificaciones de runtime, de imagen y de distribución de OCI permiten que las imágenes creadas con cualquier herramienta se almacenen en cualquier registro y se ejecuten en cualquier runtime compatible.
- Kubernetes tiene un plano de control (API server, etcd, scheduler, controller manager) y nodos de trabajo (kubelet, runtime de contenedores, kube-proxy); solo el API server se comunica con etcd, y los nodos obtienen su trabajo por extracción (pull).
- Los controladores ejecutan bucles de reconciliación activados por nivel (level-triggered) que comparan el estado real con el deseado y actúan sobre la diferencia, lo que hace que el sistema resista los fallos y los eventos perdidos.

- Los pods son desechables; los Deployments los despliegan, los Services les dan nombres estables, Gateway API enruta el tráfico de entrada, los StatefulSets aportan identidad y almacenamiento, los Jobs se ejecutan hasta completarse y el HPA escala las réplicas.
- Los requests guían la planificación y el autoescalado, los limits topan el uso (la CPU se estrangula, el exceso de memoria se mata); las sondas de actividad (liveness) no deben comprobar dependencias compartidas.
- Kubernetes compensa con muchos servicios, varios equipos y un equipo de plataforma; para unos pocos servicios sin estado, un PaaS o una plataforma de contenedores serverless suele ser la mejor opción.

Imágenes, capas y registros

El Capítulo 0 presentó los contenedores como procesos aislados que comparten el kernel del host, empaquetados como imágenes creadas a partir de un Dockerfile (servidores y la nube). Un arquitecto necesita algo más de detalle, porque el diseño de las imágenes afecta a la velocidad de compilación, a la seguridad y a la fiabilidad con la que el mismo software llega a todos los entornos.

Una imagen de contenedor es una pila de *capas* de solo lectura, cada una un conjunto comprimido de cambios en archivos, más un pequeño documento de configuración (el comando que hay que ejecutar, las variables de entorno, el usuario) y un *manifiesto* que las enumera. Cada capa y cada manifiesto se identifican por un *digest*, un hash SHA-256 de su contenido. Las capas idénticas se almacenan y se descargan una sola vez, así que cien imágenes construidas sobre la misma base comparten sus capas. Como cada instrucción de un Dockerfile produce una capa y las herramientas de construcción reutilizan las capas en caché hasta que algo cambia, las instrucciones se ordenan de la que cambia con menos frecuencia a la que cambia con más: la imagen base, después las dependencias y después el código de la aplicación. Las *compilaciones multietapa* compilan en una etapa y copian solo el resultado en una imagen final pequeña, dejando atrás los compiladores y las herramientas de compilación.

Las imágenes viven en un *registro*: Docker Hub, GitHub Container Registry, Amazon ECR, Azure Container Registry, Google Artifact Registry o uno autoalojado como Harbor. Una *etiqueta* (tag) como `orders:2.4.1` es un rótulo movible; cualquiera con permisos de push puede apuntarla mañana a otra imagen. Un digest como `orders@sha256:3f1c...` nunca puede cambiar de significado. Por eso, los despliegues en producción deben resolver las etiquetas a digests, de modo que lo que se probó sea exactamente lo que se ejecuta, y nunca deben usar `latest`.

Las imágenes pequeñas se descargan más rápido y tienen menos componentes a los que aplicar parches. La práctica habitual es una base mínima (una distribución reducida o una imagen distroless que contiene solo el runtime), ejecutarse como usuario no root, reconstruir

con regularidad para incorporar las correcciones de seguridad y analizar las imágenes en el pipeline en busca de vulnerabilidades conocidas. Firmar las imágenes y adjuntarles una lista de materiales de software forma parte de la seguridad de la cadena de suministro, que trata el Capítulo 8 (seguridad de la cadena de suministro de software).

Los estándares OCI

Docker popularizó los contenedores a partir de 2013, y en 2015 Docker y otros fundaron la Open Container Initiative (OCI) bajo la Linux Foundation para evitar que el formato se fragmentara. La OCI mantiene tres especificaciones. La *especificación de runtime* dice cómo ejecutar un contenedor desempaquetado (namespaces, grupos de control, puntos de montaje, el proceso que hay que iniciar); `runc` es su implementación de referencia. La *especificación de imagen* define las capas, los manifiestos y la configuración. La *especificación de distribución* define la API HTTP que exponen los registros para subir y descargar imágenes. La versión 1.1 de las especificaciones de imagen y de distribución (2024) añadió formas estándar de almacenar otros artefactos en los registros y de vincularlos a una imagen, que es como hoy viajan las firmas y las listas de materiales de software junto a las imágenes que describen.

El resultado práctico es la intercambiabilidad. Una imagen creada con Docker, Podman, Buildah o un servicio de compilación en la nube puede almacenarse en cualquier registro compatible y ejecutarse en cualquier runtime compatible: containerd y CRI-O en un nodo de Kubernetes, una plataforma de contenedores serverless o la computadora portátil de un desarrollador. Por eso los contenedores, a diferencia de la mayoría de los formatos de empaquetado anteriores, se convirtieron en una lengua franca entre el desarrollo, las operaciones y todas las grandes nubes.

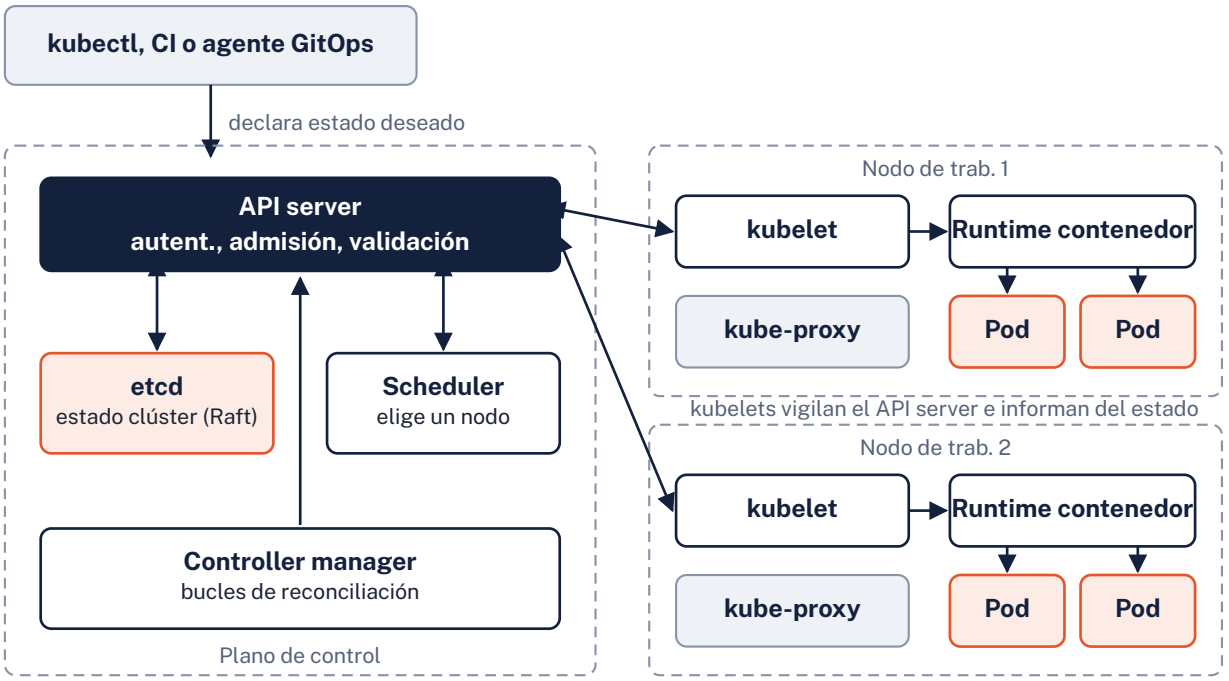
Por qué hace falta la orquestación

Ejecutar un contenedor es un solo comando. Ejecutar la plataforma de reparto de comida no lo es: cuarenta servicios, de tres a treinta copias cada uno, repartidos en decenas de máquinas en tres zonas de disponibilidad, con una nueva versión de algo saliendo cada hora. Alguien tiene que decidir en qué máquina se ejecuta cada copia, reiniciar las copias que se caen, sustituir las máquinas que mueren, añadir copias para la hora punta de la cena y retirarlas después, desplegar nuevas versiones sin perder solicitudes, permitir que los servicios se encuentren entre sí mientras las copias aparecen y desaparecen, entregarles la configuración y los secretos, y conectar almacenamiento a los pocos que lo necesitan.

La orquestación de contenedores es software que hace todo esto a partir de una declaración de lo que debería estar en ejecución. Google usó para ello durante más de una década un sistema interno llamado Borg; en 2014 anunció Kubernetes, un sistema de código abierto basado en esa experiencia, lanzó la versión 1.0 en julio de 2015 y lo donó a la recién fundada Cloud Native

Computing Foundation, donde en 2018 se convirtió en el primer proyecto en graduarse. Existen alternativas (HashiCorp Nomad, Amazon ECS, plataformas de contenedores serverless), pero Kubernetes es hoy el sustrato común que ofrecen todos los grandes proveedores y la mayoría de las plataformas para instalaciones propias.

Arquitectura de Kubernetes



Todos los componentes se comunican con el API server; solo el API server se comunica con etcd, y los nodos obtienen su trabajo por extracción en lugar de recibir órdenes.

Cómo funciona Kubernetes: plano de control y nodos

Un clúster de Kubernetes tiene un plano de control, que toma las decisiones, y nodos de trabajo, que ejecutan las cargas de trabajo. El **API server** es la puerta de entrada: todos los clientes, desde el `kubectl` de un desarrollador hasta los propios componentes del clúster, leen y escriben objetos a través de su API REST, y él autentica a quien llama, comprueba la autorización y ejecuta controles de *admisión* que pueden validar o modificar cada solicitud (por ejemplo, rechazar contenedores que se ejecutan como root). Es el único componente que se comunica con **etcd**, un almacén clave-valor con consistencia fuerte que replica el estado del clúster con el protocolo de consenso Raft entre, normalmente, tres o cinco miembros, de modo que una mayoría sobrevive a la pérdida de uno o dos (consistencia, CAP y PACELC).

El **scheduler** vigila los pods nuevos sin nodo asignado, descarta los nodos que no pueden ejecutarlos (CPU o memoria no reservadas insuficientes, zona equivocada, hardware que falta, taints que el pod no tolera), puntúa los demás (repartiendo las copias entre zonas, empaquetando de forma eficiente) y registra su elección a través del API server. El **controller**

manager ejecuta decenas de controladores, cada uno responsable de un tipo de objeto: el controlador de ReplicaSet mantiene el número correcto de pods, el controlador de Node detecta los nodos que dejan de informar, el controlador de Job ejecuta tareas hasta completarlas. En una nube, un cloud controller manager también crea balanceadores de carga y conecta discos a través de la API del proveedor.

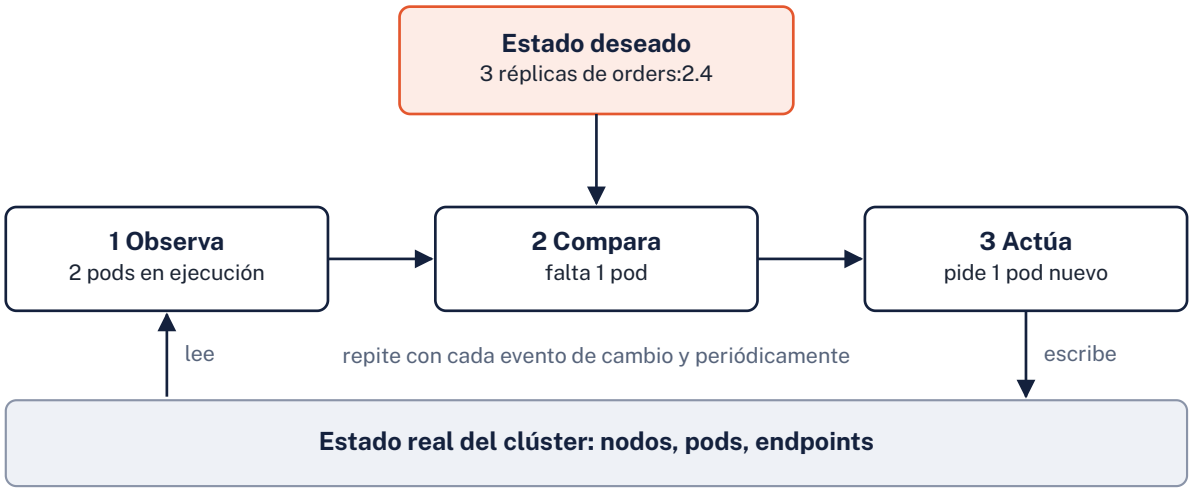
Cada nodo de trabajo ejecuta un **kubelet**, un agente que vigila el API server en busca de pods asignados a su nodo, pide al **runtime de contenedores** (normalmente containerd o CRI-O, a través de la Container Runtime Interface; el shim integrado para Docker Engine se eliminó en la versión 1.24, en 2022) que los inicie, ejecuta sus sondas de estado e informa de su estado. **kube-proxy** programa las reglas de reenvío de paquetes del nodo para que el tráfico enviado a la dirección virtual de un Service llegue a uno de sus pods; algunos plugins de red, como Cilium, lo sustituyen por un plano de datos eBPF. Un plugin de red da a cada pod su propia dirección IP, y un servicio DNS del clúster (normalmente CoreDNS) permite a los pods encontrar los Services por su nombre.

Fíjate en la dirección de las flechas. El plano de control no inicia sesión en un nodo para arrancar cargas de trabajo; los nodos vigilan el estado en el que deberían estar y convergen hacia él. (Los comandos bajo demanda son la excepción: para `kubectl exec`, `logs` y `port-forward`, el API server llama al kubelet del nodo.) Este diseño basado en extracción, de vigilar y converger, es lo que hace robusto a Kubernetes: un nodo que estuvo desconectado diez minutos simplemente se pone al día cuando vuelve. Con Kubernetes gestionado (Amazon EKS, Azure Kubernetes Service, Google Kubernetes Engine y otros), el proveedor opera y actualiza el plano de control; el cliente sigue eligiendo los tipos de nodo, actualizando los grupos de nodos y siendo responsable de todo lo que se despliega en el clúster.

Estado deseado declarativo y bucles de reconciliación

La idea central de Kubernetes es la configuración declarativa. No le dices al clúster que inicie tres contenedores; guardas un objeto que dice que deben existir tres copias de una imagen dada: el estado deseado. Después, los controladores ejecutan bucles de reconciliación: observan el estado real, lo comparan con el estado deseado y actúan para eliminar la diferencia, para siempre.

El bucle de reconciliación



Los controladores se activan por nivel: actúan sobre la diferencia actual, no sobre una secuencia de eventos recordada, así que los eventos perdidos y los reinicios no causan daños duraderos.

Sigue lo que ocurre cuando falla un nodo de la zona B durante la hora punta de la cena. El controlador de Node deja de recibir heartbeats y marca el nodo como no preparado; tras un timeout, sus pods quedan marcados para desalojo. El controlador de ReplicaSet del servicio de pedidos observa dos pods en ejecución donde se desean tres y crea un nuevo objeto pod. El scheduler ve un pod sin planificar y lo asigna a un nodo sano, prefiriendo los supervivientes de la zona B u otra zona según las reglas de reparto. El kubelet de ese nodo ve la asignación, descarga la imagen e inicia el contenedor. Cuando su sonda de preparación (readiness) se supera, se actualizan los endpoints que hay detrás del Service de pedidos y el tráfico empieza a llegarle. Ningún componente le dijo a otro qué hacer; cada uno reaccionó al estado que observaba.

Como los controladores se *activan por nivel* (reaccionan a la diferencia actual) y no se *activan por flanco* (reaccionando a eventos individuales), un controlador que se cae, se pierde una notificación o se ejecuta dos veces sigue convergiendo al resultado correcto. El mismo patrón se puede ampliar: las *definiciones de recursos personalizados* añaden nuevos tipos de objeto, y el patrón *operador*, presentado por CoreOS en 2016, empaqueta un controlador que codifica conocimiento operativo, por ejemplo crear un clúster de PostgreSQL, hacer sus copias de seguridad y gestionar su failover a partir de un único objeto declarado. La sección 10.4 muestra el mismo bucle aplicado a entornos completos mediante GitOps.

El control declarativo tiene un costo: tienes que pensar en términos de estados finales y renunciar a la secuencia exacta. Cuando dos controladores quieren estados deseados distintos para el mismo campo, se pelean. Un ejemplo clásico es un manifiesto de Deployment en Git que

dice `replicas: 3` mientras un autoescalador lo fija en nueve; cada sincronización de GitOps vuelve a reducir el servicio en plena hora punta. La solución es que cada campo tenga exactamente un dueño, aquí eliminando `replicas` del manifiesto.

Los objetos principales de Kubernetes

Objeto	Qué es	Ejemplo de reparto de comida
Pod	Uno o más contenedores planificados juntos en un nodo, que comparten una dirección IP y volúmenes. Los pods son desechables y se sustituyen, nunca se reparan.	Una copia del servicio de pedidos, quizá con un pequeño sidecar que envía los logs.
ReplicaSet	Mantiene en ejecución un número indicado de pods idénticos. Rara vez se escribe a mano.	Lo crea y lo gestiona el Deployment de pedidos.
Deployment	Declara la plantilla de pod y el número de réplicas de una aplicación sin estado, y despliega los cambios moviendo pods de un ReplicaSet antiguo a uno nuevo, dentro de unos límites de exceso y de no disponibilidad.	Los servicios de pedidos, de carta y de precios.
Service	Un nombre estable y una IP virtual para un conjunto cambiante de pods seleccionados por etiqueta, con la carga repartida entre ellos.	`orders.default.svc`, usado por el servicio de pago.
Ingress y Gateway API	Enrutan el tráfico HTTP(S) externo hacia los Services por host y ruta. Gateway API (disponible de forma general desde la versión 1.0, en octubre de 2023) es el sucesor más expresivo y orientado a roles: el equipo de plataforma es dueño del Gateway, y los equipos de aplicación, de sus HTTPRoutes.	`api.example.com/orders` enrutado al Service de pedidos, con TLS.
ConfigMap	Configuración no secreta que se entrega a los pods como variables de entorno o archivos.	Ajustes de funcionalidades, timeouts, la URL del servicio de carta.
Secret	Valores sensibles que se entregan como los ConfigMaps. En etcd se guardan solo codificados en base64, salvo que se configure el cifrado en reposo; muchos equipos los sincronizan desde un gestor de secretos externo.	La clave de API del proveedor de pagos, la contraseña de la base de datos.
StatefulSet	Pods con nombres estables, identidades de red estables y sus propios volúmenes persistentes, que se inician y se detienen en orden.	Un clúster de Kafka o de Redis autogestionado, si no se usa un servicio gestionado.
DaemonSet	Exactamente un pod en cada nodo (o en cada nodo seleccionado).	Recolector de logs o agente de monitorización del nodo.

Objeto	Qué es	Ejemplo de reparto de comida
Job y CronJob	Pods que se ejecutan hasta completarse, una vez o según un calendario, con reintentos.	Migración de la base de datos antes de un lanzamiento; informe nocturno de pagos a los restaurantes.
HorizontalPodAutoscaler	Ajusta el número de réplicas de un Deployment para mantener una métrica cerca de un objetivo.	Escalar pedidos de 3 a 12 réplicas a medida que sube el uso de CPU por la noche.

NOTA

Los controladores de Ingress y el paso a Gateway API

El objeto Ingress solo declara reglas de enrutamiento; un *controlador de Ingress* las implementa. En noviembre de 2025, el proyecto Kubernetes anunció la retirada del controlador comunitario más usado, ingress-nginx, con un mantenimiento en la medida de lo posible que terminaba en marzo de 2026 y sin correcciones de seguridad después, y recomendó Gateway API para el trabajo nuevo. La API de Ingress en sí sigue funcionando con otros controladores, pero los equipos que usaban ingress-nginx han tenido que planificar una migración, un recordatorio de que incluso un componente que es un estándar de facto de una plataforma tiene un ciclo de vida que hay que seguir.

Un Deployment y un Service en YAML

Los manifiestos siguientes declaran el servicio de pedidos: la configuración a partir de un Secret, los requests de recursos y las sondas, un Service estable delante y un autoescalador que mantiene entre tres y doce réplicas. El Deployment no tiene a propósito el campo `replicas`, para que el autoescalador sea el único dueño del número. Se aplican con `kubectl apply` o, mejor, mediante un pipeline o un agente de GitOps a partir de un repositorio.

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orders
spec:
  # no replicas field: the autoscaler below owns the count
  selector:
    matchLabels: {app: orders}
  template:
    metadata:
      labels: {app: orders}
    spec:
      containers:
      - name: orders
        image: registry.example.com/orders:2.4.1
        ports: [{containerPort: 8080}]
        env:
        - name: DATABASE_URL
          valueFrom: {secretKeyRef: {name: orders-db, key: url}}
        resources:
          requests: {cpu: 500m, memory: 512Mi}
          limits: {memory: 512Mi}  # no CPU limit: may burst
        readinessProbe:
          httpGet: {path: /ready, port: 8080}
          periodSeconds: 5
        livenessProbe:
          httpGet: {path: /live, port: 8080}
          periodSeconds: 10
          failureThreshold: 3
```

Un Deployment para el servicio de pedidos. En producción, la imagen se fijaría por digest en lugar de por etiqueta.

```
apiVersion: v1
kind: Service
metadata:
  name: orders
spec:
  selector: {app: orders}
  ports: [{port: 80, targetPort: 8080}]
---
apiVersion: autoscaling/v2
kind: HorizontalPodAutoscaler
metadata:
  name: orders
spec:
  scaleTargetRef: {apiVersion: apps/v1, kind: Deployment, name: orders}
  minReplicas: 3
  maxReplicas: 12
  metrics:
  - type: Resource
    resource:
      name: cpu
      target: {type: Utilization, averageUtilization: 60}
```

Un Service que da a los pods una única dirección estable, y un autoescalador que apunta a un uso medio de CPU del 60% de la CPU solicitada.

Requests, limits y calidad de servicio de los recursos

Los requests y limits son las líneas con más consecuencias de un manifiesto. Un *request* es lo que reserva el scheduler: un pod que solicita 500 milicores (media CPU) y 512 MiB de memoria solo se coloca en un nodo con esa capacidad sin reservar, y tiene garantizada esa parte cuando hay contención. Un *limit* es un techo que impone el kernel: un contenedor que supera su límite de CPU se estrangula (se ralentiza), y uno que supera su límite de memoria se mata y se reinicia, el conocido estado `OOMKilled`. Kubernetes deriva de estos ajustes una clase de calidad de servicio: *Guaranteed* (los requests son iguales a los limits para todos los recursos), *Burstable* (requests definidos, limits más altos o ausentes) y *BestEffort* (nada definido; son los primeros en ser desalojados cuando un nodo se queda corto).

Los profesionales discrepan sobre los límites de CPU. Un grupo los define en todas partes por previsibilidad y por equidad entre equipos; otro define requests de CPU precisos y ningún límite de CPU, porque el estrangulamiento provoca picos de latencia incluso cuando el nodo tiene CPU libre, y los requests ya garantizan a cada pod su parte. Con la memoria hay poco debate: fija el límite igual al request, porque la memoria no se puede recuperar de un proceso sin matarlo, y un pod que usa más de lo que solicitó es el primer candidato al desalojo. En cualquier

caso, los requests deben basarse en mediciones; los requests inflados desperdician dinero (el nodo está reservado pero inactivo), y los subestimados causan problemas de vecinos ruidosos y un autoescalado engañoso.

El HorizontalPodAutoscaler calcula *réplicas deseadas = techo(réplicas actuales × métrica actual ÷ objetivo)*, midiendo el uso de CPU como porcentaje del request. Escala hacia arriba con rapidez pero, por defecto, espera una ventana de estabilización de cinco minutos antes de escalar hacia abajo, para evitar el flapping. Cuando los pods nuevos no caben en los nodos existentes, quedan en estado *Pending*, y un autoescalador de nodos (el Cluster Autoscaler, o Karpenter en algunos proveedores) añade máquinas, lo que lleva minutos en lugar de segundos. El autoescalado de pods y el de nodos son bucles separados con retrasos separados, y hay que planificar ambos para la hora punta.

EJEMPLO PRÁCTICO

Ejemplo práctico: dimensionar el servicio de pedidos para la hora punta de la cena

Cada pod de pedidos solicita 500m de CPU y 512 MiB, con un límite de memoria de 512 MiB y sin límite de CPU. Los nodos tienen 4 vCPU y 16 GiB, de los que unos 3.8 vCPU y 14.5 GiB son asignables tras las reservas del sistema (cifras ilustrativas). El HPA apunta a un 60% de CPU, con entre 3 y 12 réplicas.

Pods por nodo	CPU: 3.8 ÷ 0.5 = 7 pods. Memoria: 14.5 GiB ÷ 0.5 GiB = 29 pods. La CPU es la restricción limitante: **7 pods de pedidos por nodo** si no se ejecuta nada más en él.
18:00, empieza la hora punta	4 réplicas con una media del 90% de su request de CPU. Deseadas = techo(4 × 90 ÷ 60) = **6 réplicas**.
18:20, sigue subiendo	6 réplicas al 85%. Deseadas = techo(6 × 85 ÷ 60) = techo(8.5) = **9 réplicas**, que solicitan 4.5 vCPU en total.
Capacidad de los nodos	Si otros servicios ya llenan los nodos, tres de los pods nuevos quedan en Pending; el autoescalador de nodos añade un nodo, que tarda quizá de dos a cuatro minutos en unirse y descargar la imagen.
Respuesta de diseño	Sube `minReplicas` antes de la hora punta predecible (escalado programado), mantén algo de capacidad de nodos libre y mantén la imagen pequeña para que los pods nuevos arranquen rápido.
Efecto de un request equivocado	Si cada pod en realidad solo necesitara 250m pero el request se quedara en 500m, la misma carga de la hora punta (225m por pod) aparecería como un 45% en lugar de un 90%: el HPA no escalaría hasta que el uso superara los 300m por pod, y el clúster reservaría el doble de CPU de la que necesita.

El autoescalado reacciona a una carga que ya ha llegado; para un pico diario predecible, el escalado previo programado es más barato que los pedidos perdidos.

Los requests guían tanto al scheduler como al autoescalador, así que medirlos es una tarea arquitectónica de primer orden, no un detalle menor de ajuste.

Sondas de actividad, de preparación y de arranque

Las sondas son la forma en que Kubernetes se entera del estado de salud. Una sonda de **preparación** (readiness) responde a si este pod debe recibir tráfico ahora; mientras falla, el pod se retira de los endpoints de su Service, pero sigue en ejecución. Una sonda de **actividad**

(liveness) responde a si el proceso está atascado sin posibilidad de recuperarse; cuando falla repetidamente, el kubelet reinicia el contenedor. Una sonda de **arranque** (startup) retiene las otras dos hasta que una aplicación de arranque lento ha terminado de inicializarse.

El error más dañino es una sonda de actividad que comprueba las dependencias. Si el endpoint de actividad del servicio de pedidos consulta la base de datos, un failover de la base de datos de treinta segundos hace que todos los pods de pedidos fallen la sonda de actividad a la vez, Kubernetes los reinicia todos, todos se vuelven a conectar en el mismo momento, y un breve tropiezo se convierte en una caída. La sonda de actividad solo debe comprobar que el propio proceso puede responder; la de preparación puede tener en cuenta si el pod puede hacer trabajo útil, aunque incluso ahí muchos equipos evitan dejar todos los pods como no preparados por una dependencia compartida, y prefieren hacer fallar rápido las solicitudes con un circuit breaker (consulta patrones de resiliencia).

Las sondas funcionan junto con el apagado ordenado. Cuando se elimina un pod durante un despliegue progresivo o una reducción de escala, Kubernetes lo retira de los endpoints y envía al proceso `SIGTERM`; el proceso debe dejar de aceptar trabajo nuevo, terminar las solicitudes en curso y salir dentro del periodo de gracia (30 segundos por defecto) antes de que lo maten. Como la retirada de los endpoints se propaga de forma asíncrona, un breve retardo `preStop` evita enviar solicitudes a un pod que ya ha empezado a detenerse. La sección 10.3 vuelve sobre esto como el principio de desechabilidad de los doce factores.

Cuándo Kubernetes es excesivo

Kubernetes es una plataforma de propósito general para construir plataformas, y cobra por esa generalidad. Alguien tiene que diseñar la red, la entrada de tráfico, la identidad, los secretos, las políticas, la observabilidad y la multitenencia; seguir el ritmo de tres versiones menores al año, cada una con soporte durante unos catorce meses, además de los complementos que dependen de ellas; y estar de guardia tanto para el clúster como para las aplicaciones. Los planos de control gestionados eliminan parte de este trabajo, no la mayor parte.

Para un equipo pequeño que ejecuta un puñado de servicios web sin estado, un PaaS o una plataforma de contenedores serverless (Google Cloud Run, Azure Container Apps, Amazon ECS con Fargate y similares) ofrece la mayoría de los beneficios (contenedores, autoescalado, despliegues graduales, TLS gestionado) con una fracción de la carga operativa. Kubernetes empieza a compensar cuando hay muchos servicios y varios equipos, un equipo de plataforma que lo opere, cargas de trabajo con necesidades poco habituales (sistemas con estado, GPU, trabajos por lotes, controladores personalizados), el requisito de ejecutar la misma plataforma en instalaciones propias y en varias nubes, o un ecosistema de herramientas que la

organización quiere usar. Muchas organizaciones lo han adoptado porque estaba de moda y después han pasado años construyendo, en la práctica, un PaaS peor encima de él. La sección 10.7 explica cómo evitan ese destino los equipos de plataforma (ingeniería de plataformas).

SI ESTO ES NUEVO PARA TI

Si esto es nuevo para ti

No necesitas memorizar todos los objetos de Kubernetes. Recuerda tres ideas: describes en archivos lo que quieres y el sistema lo hace realidad; las unidades pequeñas y desechables (los pods) se sustituyen en lugar de repararse; y los nombres estables (los Services) ocultan el hecho de que las copias que hay detrás cambian sin parar. Todo lo demás es detalle construido sobre esas tres ideas.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué un despliegue en producción debe referirse a una imagen por digest y no por etiqueta?

Una etiqueta es un rótulo movible que se puede volver a apuntar a otro contenido; un digest es un hash del contenido y siempre identifica exactamente la imagen que se probó.

▼ Un nodo muere. ¿Qué componentes actúan, y en qué orden, para restaurar la tercera réplica?

El controlador de Node marca el nodo como no preparado y sus pods se desalojan; el controlador de ReplicaSet ve que hay muy pocos pods y crea uno; el scheduler lo asigna a un nodo sano; el kubelet de ese nodo lo inicia a través del runtime; cuando supera la sonda de preparación, se une a los endpoints del Service.

▼ ¿Qué le ocurre a un contenedor que supera su límite de memoria, y a uno que supera su límite de CPU?

Superar el límite de memoria hace que el contenedor se mate (OOMKilled) y se reinicie; superar el límite de CPU solo lo estrangula.

▼ ¿Por qué es peligroso comprobar la base de datos en una sonda de actividad?

Un breve problema de la base de datos hace que todos los pods fallen la sonda de actividad a la vez, así que Kubernetes los reinicia todos de golpe, y un corto tropiezo de una dependencia se convierte en una caída total.

▼ **Inténtalo: Cada pod solicita 750m de CPU y 1 GiB. Los nodos ofrecen 7.6 vCPU y 28 GiB asignables. La avalancha necesita 24 pods. ¿Cuántos pods caben por nodo y cuántos nodos necesitas si debes sobrevivir a la pérdida de uno?**

La CPU permite 7.6 / 0.75 = 10.1, es decir, 10 pods; la memoria permite 28, así que la CPU es el límite efectivo. Veinticuatro pods necesitan 2.4, es decir, 3 nodos. Sobrevivir al fallo de un nodo requiere uno más: 4 nodos. Las solicitudes de recursos determinan la programación, así que mantenlas cerca del uso real. Reparte los pods entre zonas. Comprueba que el máximo del autoescalador permita 24 réplicas y que el clúster pueda añadir nodos antes de la hora punta.

FUENTES

- Kubernetes documentation: Cluster components: https://kubernetes.io/docs/concepts/overview/components/
- Kubernetes documentation: Resource management for pods and containers: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
- Kubernetes documentation: Configure liveness, readiness and startup probes: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
- Open Container Initiative: https://opencontainers.org/
- Kubernetes blog (2025): Ingress NGINX retirement: https://kubernetes.io/blog/2025/11/11/ingress-nginx-retirement/

SECCIÓN 10.3 DE 10.8 · INTERMEDIO · 25 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

La aplicación de doce factores y los principios nativos de la nube

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.8 Cómo se hace el software
- 9.2 Balanceo de carga
- 10.2 Contenedores y Kubernetes

Términos anteriores que utiliza: Imagen de contenedor, Entrega continua, Idempotente, Escalado horizontal (scaling out), Observabilidad

LA IDEA EN BREVE

Piensa en una cadena de cocinas de comida para llevar. Una buena ficha de receta sirve en cualquier sucursal: enumera todos los ingredientes, no dice nada sobre la sucursal en la que está y cualquier cocinero puede retomarla a mitad de turno, porque nada importante se guarda en la cabeza de un solo cocinero. Los suministros llegan en cajas etiquetadas y, cuando cambia la carta, se imprime una ficha nueva y numerada en lugar de garabatear sobre la anterior. La aplicación de doce factores es ese tipo de ficha de receta para el software: reglas que permiten que las máquinas arranquen, copien, muevan, sustituyan y escalen la misma aplicación sin que nadie tenga que conocer sus manías.

LO QUE NECESITAS SABER

- The Twelve-Factor App (Adam Wiggins, Heroku, 2011, liberada como código abierto en 2024) describe el contrato entre una aplicación y una plataforma moderna, y se aplica por igual a monolitos y a microservicios.
- La configuración (III) y los servicios de respaldo (IV) mantienen fuera del artefacto todo lo que varía entre despliegues, de modo que un único artefacto se ejecuta en todos los entornos; hoy los secretos deben estar en un gestor de secretos o sustituirse por la identidad de carga de trabajo, y no en simples variables de entorno.
- Build, release, run (V) hace que las versiones sean inmutables y estén numeradas, de modo que hacer rollback significa ejecutar una versión anterior y nadie edita un servidor en ejecución.

- Los procesos sin estado (VI), la concurrencia por tipo de proceso (VIII) y la desechabilidad (IX) son lo que hace seguros el escalado horizontal, las actualizaciones graduales y la sustitución de instancias que han fallado.
- La paridad entre desarrollo y producción (X), los logs como flujos (XI) y los procesos de administración ejecutados como procesos puntuales separados (XII) reducen las sorpresas en producción; las migraciones deben ser compatibles con el código antiguo y con el nuevo.
- Más allá de los doce factores, las aplicaciones nativas de la nube se diseñan pensando en la desechabilidad, la inmutabilidad, la observabilidad y la seguridad, y la ampliación de Hoffman de 2016 añade API primero, telemetría y autenticación y autorización.
- Una auditoría frente a los factores debe ordenar las carencias por riesgo: primero los secretos expuestos, después todo lo que bloquee la plataforma de destino, como el estado local, y luego la fiabilidad y la comodidad.

De dónde vienen los doce factores

En 2011, Adam Wiggins, cofundador de Heroku, publicó *The Twelve-Factor App*, un breve documento web que destilaba lo que el equipo de Heroku había aprendido al observar cómo se construía, desplegaba y escalaba en su plataforma un número muy grande de aplicaciones. La metodología de la aplicación de doce factores describe aplicaciones que se configuran a partir de pasos declarativos y automatizados, tienen un contrato limpio con la plataforma sobre la que se ejecutan para poder funcionar en cualquier lugar, mantienen el desarrollo y la producción lo bastante cerca como para permitir el despliegue continuo y pueden escalar horizontalmente sin rediseñarse.

El documento es anterior a Docker (2013) y a Kubernetes (2014), y aun así describe casi exactamente el contrato que esas plataformas impusieron después: una aplicación es un conjunto de procesos sin estado, compilados en un artefacto inmutable, configurados desde fuera, que escriben sus logs en un flujo y se comunican con todo lo demás por la red. Por eso sigue siendo el vocabulario común del diseño de aplicaciones nativas de la nube. En noviembre de 2024, Heroku anunció que la metodología pasaba a ser de código abierto, mantenida por una comunidad de colaboradores de varias empresas, con el objetivo de actualizarla a la práctica actual; sus doce factores originales siguen siendo el punto de referencia.

Los factores tratan de la relación de la aplicación con su plataforma de ejecución, no de su arquitectura interna. Un monolito modular puede cumplir perfectamente los doce factores; un conjunto de microservicios puede incumplirlos todos. Complementan las inquietudes de desplegabilidad y operabilidad del Capítulo 2 y los principios de escalado y ausencia de estado del Capítulo 9.

Los doce factores, con matices actuales

Factor	Qué significa	Por qué importa	Matiz actual
I. Código base	Un único código base en control de versiones por aplicación, desplegado en muchos entornos.	Todos los entornos ejecutan alguna versión del mismo código; no hay bifurcaciones por cliente ni por entorno.	Un monorrepo con varias aplicaciones está bien si cada aplicación tiene un límite claro; el antipatrón son las bifurcaciones de copiar y modificar.
II. Dependencias	Declara explícitamente cada dependencia y aísla la aplicación del sistema anfitrión.	Las compilaciones son reproducibles en cualquier máquina; nada depende de un paquete que alguien instaló a mano.	Los lockfiles y las imágenes lo convierten en lo habitual; las dependencias fijadas siguen necesitando actualizaciones periódicas y análisis de vulnerabilidades.
III. Configuración	Mantén todo lo que varía entre despliegues (URL, credenciales, ajustes de funcionalidades) fuera del código, en el entorno.	Un único artefacto se ejecuta en todos los entornos; los secretos nunca entran en el repositorio.	Las variables de entorno se filtran a los logs, a los informes de fallos y a los procesos hijos; es mejor leer los secretos de un gestor de secretos o de archivos montados.
IV. Servicios de respaldo	Trata las bases de datos, las colas, las cachés y las API externas como recursos conectados que se localizan mediante la configuración.	Una base de datos local puede sustituirse por una gestionada, o una que ha fallado por una copia restaurada, cambiando una URL.	Los cambios rara vez son transparentes cuando la semántica difiere; aísla las diferencias detrás de puertos y adaptadores.
V. Build, release, run	Separa la compilación de un artefacto, su combinación con la configuración para formar una versión y su ejecución.	Las versiones son inmutables y están numeradas; cualquier versión puede volver a ejecutarse; nadie edita código en un servidor.	Hoy: compila una vez en CI, guarda la imagen por su digest y registra las versiones en Git o en la herramienta de despliegue.
VI. Procesos	Ejecuta la aplicación como uno o varios procesos sin estado que no comparten nada; guarda el estado en servicios de respaldo.	Cualquier copia puede atender cualquier solicitud, así que las copias pueden añadirse, quitarse y sustituirse libremente.	Las sesiones persistentes (sticky sessions) ocultan el problema en lugar de resolverlo; el disco local es solo un área de trabajo temporal.

Factor	Qué significa	Por qué importa	Matiz actual
VII. Asignación de puertos	La aplicación es autónoma y expone su servicio escuchando en un puerto.	La plataforma le dirige el tráfico sin tener que inyectarla en un servidor web aparte.	Hoy es universal; los consumidores de colas no tienen puerto, y una service mesh puede situarse delante.
VIII. Concurrencia	Escala horizontalmente ejecutando más procesos, con distintos tipos de proceso para distintos trabajos (web, worker).	Cada tipo de trabajo escala de forma independiente según su propia carga.	Los autoescaladores lo hacen automáticamente, incluso según la longitud de la cola; los hilos dentro de un proceso siguen importando.
IX. Desechabilidad	Arranca rápido, detente de forma ordenada al recibir SIGTERM, sobrevive a una muerte repentina.	Escalado rápido, despliegues graduales seguros y robustez frente a caídas.	Los entornos de ejecución que arrancan despacio necesitan sondas de arranque (startup probes) o técnicas de arranque más rápido; los trabajos en segundo plano deben ser idempotentes y reanudables.
X. Paridad entre desarrollo y producción	Mantén el desarrollo, staging y producción tan parecidos como sea posible en tiempo, personas y herramientas.	Menos sorpresas que solo aparecen en producción; el despliegue continuo se vuelve seguro.	La paridad total es demasiado costosa; prioriza los mismos tipos y versiones de servicios de respaldo, usando contenedores en local.
XI. Logs	Escribe los logs como un flujo de eventos sin búfer en la salida estándar; deja que la plataforma los encamine.	La aplicación no gestiona archivos; los logs de todas las copias se recogen de forma centralizada.	Usa logs estructurados además de métricas y trazas (por ejemplo, mediante OpenTelemetry); vigila el volumen y el costo de los logs.
XII. Procesos de administración	Ejecuta las tareas puntuales (migraciones, scripts) como procesos separados que usan el mismo código y la misma configuración.	Las tareas de administración se ejecutan exactamente sobre la versión que está en producción, no sobre una copia desactualizada.	Ejecútalas como pasos del pipeline o como Jobs de Kubernetes; nunca dejes que todas las réplicas ejecuten migraciones al arrancar sin un bloqueo.

Configuración, secretos y servicios de respaldo

El factor de configuración tiene una prueba sencilla: ¿podría hacerse público hoy el código base sin exponer ni una sola credencial? Si no es así, la configuración se ha filtrado al código. Separar la configuración del código permite que un único artefacto pase sin cambios por todos los entornos, que es la base del principio de compilar una vez de la Sección 10.5 (CI/CD y despliegue). La configuración incluye las URL de las bases de datos y de los servicios, las credenciales y los ajustes de cada entorno; excluye el cableado interno que es igual en todas partes, como qué clase atiende qué ruta, que pertenece al código.

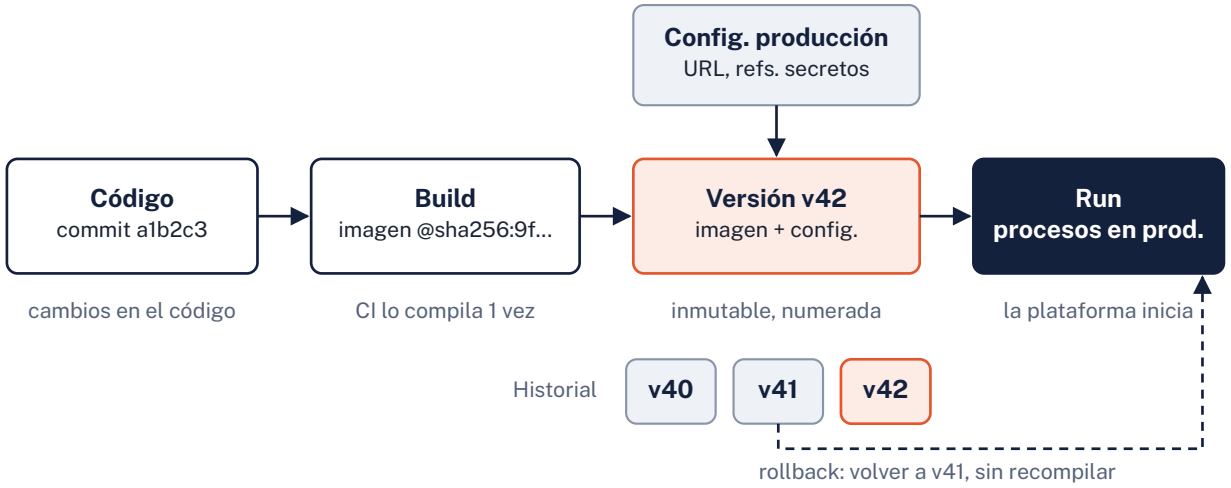
La recomendación original de usar variables de entorno ha envejecido de forma desigual. Son universales y neutrales respecto al lenguaje, y Kubernetes, las plataformas PaaS y las funciones las admiten. Pero el entorno de un proceso es visible para sus procesos hijos, a menudo se vuelca en los informes de fallos y en las páginas de depuración, y no puede rotarse sin reiniciar. La práctica actual para los secretos es obtenerlos al arrancar de un gestor de secretos (AWS Secrets Manager, Azure Key Vault, Google Secret Manager, HashiCorp Vault u OpenBao) o montarlos como archivos que la plataforma puede actualizar y, mejor aún, sustituir los secretos de larga duración por la identidad de carga de trabajo, con la que la plataforma da automáticamente a cada servicio credenciales de corta duración. La intención del factor, mantener la configuración fuera del artefacto, no ha cambiado.

Tratar un servicio de respaldo como un recurso conectado significa que el código llega a la base de datos de pedidos, al proveedor de pagos y al intermediario de mensajes de la misma manera: mediante una URL y unas credenciales suministradas en el momento del despliegue. Eso hace que los entornos sean componibles y que la recuperación sea más rápida. No hace que los servicios de respaldo sean intercambiables por arte de magia; sustituir un intermediario de mensajes por otro cambia la semántica de entrega. El estilo hexagonal del Capítulo 5 complementa el factor: la configuración dice dónde está el recurso, y un adaptador detrás de un puerto oculta cómo se comporta.

Build, release, run (construcción, lanzamiento, ejecución)

El factor build, release, run (construcción, lanzamiento, ejecución) separa tres etapas que a menudo se confunden. La etapa de *construcción* (build) convierte un commit en un artefacto ejecutable, hoy normalmente una imagen de contenedor identificada por su digest. La etapa de *lanzamiento* (release) combina ese artefacto con la configuración de un entorno y da al resultado una identidad, como la versión v42 del servicio de pedidos en producción. La etapa de *ejecución* (run) arranca procesos a partir de una versión. Las versiones son inmutables: un cambio de código o de configuración produce una versión nueva, nunca una edición de una existente.

Compilar una vez, lanzar por entorno, ejecutar en cualquier lugar



Como las versiones son inmutables y están numeradas, hacer rollback significa volver a ejecutar una versión anterior, sin recompilar ni editar nada.

Esta separación es lo que hace que desplegar sea aburrido. Nadie edita archivos en un servidor, así que los servidores no derivan; la misma imagen que superó las pruebas en staging se ejecuta en producción, así que un resultado de prueba significa algo; y el rollback es una operación rápida y ensayada en lugar de una recompilación de emergencia. También es una pista de auditoría: para cualquier momento puedes decir exactamente qué código y qué configuración se estaban ejecutando. Kubernetes guarda un historial de revisiones de los ReplicaSets de cada Deployment, las plataformas PaaS numeran las versiones y GitOps (Sección 10.4) convierte el historial de Git del repositorio de versiones en el registro.

Procesos, concurrencia y desechabilidad

Los procesos sin estado que no comparten nada son la condición previa de casi todo lo demás en este capítulo: el escalado horizontal, los despliegues graduales, el autoescalado, la sustitución de nodos que han fallado y la distribución entre zonas. Todo lo que un proceso guarda solo en su memoria o en su disco local (la sesión de un usuario, un archivo subido, un informe a medio generar) se pierde cuando se sustituye ese proceso, y es invisible para las demás copias. El Capítulo 9 trata en detalle la gestión de sesiones (balanceo de carga, ausencia de estado y sesiones). El estado no desaparece; se traslada a servicios de respaldo diseñados para guardarlo.

El factor de concurrencia añade que los distintos tipos de trabajo deben ejecutarse como distintos tipos de proceso que escalan por separado. En la plataforma de reparto de comida, el proceso web que atiende la app del cliente escala según la tasa de solicitudes, mientras que un

proceso worker que envía las confirmaciones de pedido escala según la longitud de la cola, y un consumidor de ubicaciones de los repartidores, según el throughput de eventos. Mezclarlos en un solo tipo de proceso obliga a escalarlos todos por las necesidades de uno.

La desechabilidad exige que los procesos arranquen en segundos y se detengan limpiamente. Al recibir `SIGTERM`, un proceso web deja de aceptar conexiones y termina las solicitudes en curso; un worker devuelve su mensaje actual a la cola o lo termina dentro del periodo de gracia. Como los procesos también pueden morir sin aviso (un fallo de nodo, una eliminación por falta de memoria), el trabajo debe poder repetirse sin peligro: un tratamiento idempotente de los mensajes, como en la arquitectura dirigida por eventos, y trabajos que se reanudan desde puntos de control. George Candea y Armando Fox llevaron la idea al límite en 2003 con el *software crash-only*, que tiene una sola forma de detenerse (caerse) y una sola forma de arrancar (recuperarse), de modo que la ruta de recuperación se ejercita constantemente en lugar de descubrirse durante un incidente.

SI ESTO ES NUEVO PARA TI

Paso a paso: un worker al que se le pide que se detenga

La idea que requiere más atención aquí es la desechabilidad: un proceso debe contar con que lo detengan en cualquier momento, con educación o sin ella, y no perder nada cuando ocurra. Sigue a un proceso worker a través de ambos tipos de detención.

- La plataforma de reparto de comida ejecuta un proceso worker que toma mensajes de una cola y envía confirmaciones de pedido. Durante un despliegue rutinario, cada copia antigua del worker se sustituye por una nueva, así que cada copia antigua debe detenerse.
- La forma educada es una señal llamada SIGTERM, que significa 'por favor, termina lo que estés haciendo y sal', como describió la Sección 10.2. La plataforma espera un periodo de gracia antes de forzar el fin del proceso.
- Un worker desechable reacciona de inmediato. Deja de tomar mensajes nuevos, termina la confirmación que está enviando o devuelve ese mensaje a la cola para que otra copia lo recoja, y sale. No se pierde nada.
- Pero un proceso también puede morir sin aviso: su máquina falla, o se elimina por usar demasiada memoria. No hay ocasión de ordenar nada.
- Supón que el worker muere justo después de enviar una confirmación pero antes de avisar a la cola de que había terminado. El mensaje se ofrece de nuevo y otra copia lo trata otra vez. Por eso el tratamiento debe ser idempotente, como explicó el Capítulo 5: hacerlo dos veces tiene el mismo efecto que hacerlo una. Esto suele conseguirse dando a cada mensaje una clave única, que se transmite con la confirmación, para que una repetición se reconozca y se omita.
- **La lección:** trata la detención y la caída como sucesos cotidianos. Un proceso que arranca rápido, se detiene limpiamente cuando se le pide y no pierde nada cuando muere sin aviso puede sustituirse, moverse y escalarse libremente.

Paridad entre desarrollo y producción, logs y procesos de administración

El texto original identifica tres brechas entre desarrollo y producción: el tiempo (código escrito semanas antes de desplegarse), las personas (los desarrolladores lo escriben, los operadores lo despliegan) y las herramientas (SQLite en una computadora portátil, PostgreSQL en producción). La entrega continua cierra la primera, los equipos que operan lo que construyen cierran la segunda y los contenedores cierran buena parte de la tercera: un desarrollador puede ejecutar en local la misma versión de PostgreSQL, el mismo intermediario de mensajes y la misma caché, y herramientas como Testcontainers arrancan dependencias reales en las

pruebas de integración. La paridad perfecta no es ni asequible ni necesaria; lo que más importa es usar los mismos tipos y versiones de servicios de respaldo, porque ahí es donde viven las diferencias que sorprenden.

Los logs como flujos de eventos significan que la aplicación escribe cada evento en la salida estándar y deja el almacenamiento, la rotación y el envío a la plataforma. Eso sigue siendo correcto, con dos añadidos actuales: los logs deben ser estructurados (por ejemplo, JSON con nombres de campo coherentes y un identificador de solicitud o de traza), y los logs son una de las tres señales, junto con las métricas y las trazas, que hacen observable un sistema (observabilidad). El volumen de logs es también un factor de costo real; la Sección 10.8 vuelve sobre ello.

Los procesos de administración, como las migraciones de bases de datos, las reparaciones de datos y los informes puntuales, deben ejecutarse con la misma versión (código y configuración) que la aplicación, como procesos separados. En Kubernetes eso suele significar un Job que el pipeline ejecuta antes de desplegar la nueva versión. Las migraciones requieren un cuidado especial: deben ser compatibles tanto con la versión antigua como con la nueva del código, que se ejecutan en paralelo, que es la disciplina de expandir y contraer de la evolución de esquemas.

Más allá de los doce factores

Los doce factores nunca pretendieron ser completos, y autores posteriores los ampliaron. *Beyond the Twelve-Factor App* (O'Reilly, 2016), de Kevin Hoffman, revisó la lista hasta quince, añadiendo *API primero* (diseñar el contrato antes que la implementación), *telemetría* (incorporar métricas, estado de salud y trazado) y *autenticación y autorización* (la seguridad como inquietud de diseño desde el principio). La práctica actual añade cuatro temas que toda aplicación nativa de la nube debería cumplir.

Desechabilidad, en serio. Diseña para que te eliminen en cualquier momento: operaciones idempotentes, timeouts de solicitud cortos, tiempo de arranque acotado, drenaje ordenado y ningún proceso cuya pérdida importe. **Inmutabilidad.** Nunca parchees un contenedor o un servidor en ejecución; compila una imagen nueva y sustitúyelo (Sección 10.4, sobre la infraestructura inmutable). Así cada instancia en ejecución puede rastrearse hasta un commit y desaparece la deriva. **Observabilidad.** Emite logs estructurados, métricas y trazas mediante un estándar como OpenTelemetry, expón endpoints de estado que reflejen el estado del propio proceso y propaga el contexto de traza entre llamadas, para que los operadores puedan responder preguntas que no habían previsto. **Seguridad.** Ejecuta como usuario sin privilegios de root con un sistema de archivos de solo lectura siempre que sea posible; obtén las

credenciales mediante la identidad de carga de trabajo en lugar de claves de larga duración; aplica el mínimo privilegio a cada servicio de respaldo; y compila a partir de imágenes mínimas, analizadas y firmadas (seguridad de la cadena de suministro).

EJEMPLO PRÁCTICO

Ejemplo práctico: auditoría del servicio de alta de restaurantes

El servicio de alta de restaurantes de la plataforma de reparto de comida es una aplicación web de cinco años que se ejecuta en dos máquinas virtuales configuradas a mano detrás de un balanceador de carga con sesiones persistentes. El equipo quiere trasladarlo a la plataforma de contenedores de la empresa. Una auditoría frente a los factores encuentra nueve problemas; a cada uno se le asigna un riesgo, una solución y una prioridad.

1. Contraseña de la base de datos de producción en un archivo de propiedades del repositorio (III)	Riesgo: cualquiera con acceso de lectura al código, incluidos antiguos contratistas, tiene la credencial. Solución: rotar la contraseña ya, trasladarla al gestor de secretos e inyectarla en el momento del despliegue. **Prioridad 1**.
2. Cartas y logotipos subidos que se escriben en el disco local de la VM (VI, IV)	Riesgo: los archivos desaparecen cuando una solicitud llega a la otra VM o cuando se sustituye una VM; bloquea cualquier paso a contenedores desechables. Solución: almacenamiento de objetos, con una copia puntual de los archivos existentes. **Prioridad 1**.
3. Sesiones en la memoria del proceso, balanceo de carga persistente (VI)	Riesgo: cada despliegue cierra la sesión del personal del restaurante a mitad de un formulario. Solución: tokens de sesión firmados o un almacén de sesiones compartido. **Prioridad 2**.
4. Despliegue por SSH, `git pull` y reinicio en cada VM (V, II)	Riesgo: las dos VM ejecutan código distinto tras un pull fallido; no hay rollback. Solución: CI compila una imagen por commit; versiones numeradas. **Prioridad 2**.
5. Migraciones que ambas instancias ejecutan automáticamente al arrancar (XII)	Riesgo: dos instancias compiten por la misma migración. Solución: un paso de migración separado en el pipeline, con cambios de expandir y contraer. **Prioridad 2** (Prioridad 1 si ya se ha producido una carrera).
6. Procesos eliminados con `kill -9` durante los despliegues (IX)	Riesgo: formularios de alta guardados a medias. Solución: tratar SIGTERM, drenar las solicitudes, añadir readiness y un retardo previo a la detención (pre-stop). **Prioridad 2**.
7. Logs escritos en archivos rotativos en cada VM (XI)	Riesgo: los logs se pierden con la VM; es difícil buscar entre instancias. Solución: JSON estructurado en la salida estándar, recogido de forma centralizada. **Prioridad 3**, barato de hacer junto con el punto 4.
8. Informe nocturno de pagos a restaurantes como entrada de cron solo en la VM 1 (XII, VIII)	Riesgo: el trabajo deja de ejecutarse sin avisar cuando se reconstruye la VM 1, o se ejecuta dos veces si se copia. Solución: un trabajo programado en la plataforma, hecho idempotente. **Prioridad 3**.

9. Los desarrolladores usan SQLite y producción usa PostgreSQL (X)	Riesgo: SQL que funciona en local falla en producción. Solución: PostgreSQL en un contenedor para desarrollo y pruebas. **Prioridad 3**.

Las prioridades siguen primero al riesgo (una credencial expuesta es un incidente en curso), después a lo que bloquea la plataforma de destino (el estado local), luego a la fiabilidad y por último a la comodidad.

El equipo no necesita arreglarlo todo antes de migrar: los puntos 1 y 2 son requisitos previos; los puntos 3 a 6 pueden hacerse durante el traslado; los puntos 7 a 9 pueden venir después.

Una auditoría como esta es también un buen material para un registro de decisiones de arquitectura: explica por qué se hace cada cambio, en qué orden y qué se ha aplazado deliberadamente.

ATENCIÓN

Interpretaciones erróneas habituales de los doce factores

- Sin estado no significa que el sistema no tenga estado; significa que los procesos no lo guardan. Trasladar el estado a una única caché sin réplicas solo traslada el problema.
- Los doce factores no son una definición de microservicios. Se aplican igual a un monolito bien construido, y un conjunto de servicios puede incumplirlos.
- Poner los secretos en variables de entorno cumple la letra del factor III, pero no la práctica de seguridad actual; es preferible un gestor de secretos y la identidad de carga de trabajo.
- La paridad entre desarrollo y producción no significa una copia completa de producción para cada desarrollador; significa los mismos tipos y versiones de dependencias, y brechas cortas en tiempo y personas.
- La asignación de puertos y los logs en la salida estándar son victorias fáciles; a veces los equipos hacen esto y declaran que cumplen mientras siguen guardando archivos en el disco local.

Preguntas frecuentes

¿Toda aplicación necesita cumplir los doce factores?
No. Son una lista de comprobación de propiedades que abaratan el despliegue, el escalado y la recuperación en una plataforma moderna. Trata cada carencia como un riesgo con un costo, y corrige primero las que bloquean tu plataforma o amenazan la seguridad, como en la auditoría anterior.

¿Se aplica la metodología a las funciones serverless?
En gran medida, y la plataforma impone la mayor parte: las funciones no tienen estado, se configuran desde fuera, son desechables y la plataforma registra sus logs. Los factores que siguen necesitando atención son la configuración y los secretos, los servicios de respaldo, la idempotencia y la paridad entre desarrollo y producción, ya que la emulación local de las fuentes de eventos de un proveedor es imperfecta.

¿Siguen siendo relevantes los doce factores ahora que existe Kubernetes?
Sí, porque Kubernetes los da por supuestos. Un pod que guarda sesiones en memoria, escribe archivos en local o tarda dos minutos en arrancar choca con las actualizaciones graduales, el autoescalado y la reprogramación de la plataforma. Los factores describen lo que una aplicación debe hacer para que la automatización de la plataforma ayude en lugar de perjudicar.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Qué prueba sencilla muestra si la configuración se ha filtrado al código?**

Pregúntate si el código base podría hacerse público hoy sin exponer ninguna credencial. Si no, la configuración (al menos los secretos) vive en el código.

▼ **¿Por qué build, release, run hace que el rollback sea rápido?**

Cada versión es una combinación inmutable de un artefacto compilado y una configuración, con su propia identidad, así que hacer rollback significa volver a ejecutar la versión anterior, sin recompilar ni editar nada.

▼ **Un servicio guarda los archivos subidos en el disco local de su servidor. ¿Qué factor incumple y qué se rompe?**

Procesos (VI) y, se puede argumentar, servicios de respaldo (IV): los archivos faltan cuando las solicitudes llegan a otra copia y se pierden cuando se sustituye la instancia, lo que bloquea el escalado horizontal y los contenedores desechables.

▼ **¿Por qué no deben ejecutarse las migraciones automáticamente en cada réplica al arrancar?**

Varias réplicas pueden ejecutar la misma migración a la vez y competir entre sí, y el arranque se vuelve lento y frágil. Las migraciones deben ejecutarse una sola vez, como un paso de administración separado en el pipeline, usando la misma versión.

▼ **Inténtalo: Al auditar un servicio, encuentras: una contraseña de producción en el repositorio; cartas subidas guardadas en el disco local; logs escritos en un archivo; scripts de administración puntuales ejecutados iniciando sesión en un servidor. Ordénalos de cara a un traslado a una plataforma de contenedores.**

Primero, la contraseña: está expuesta ahora, así que rótala y muévela a un gestor de secretos o a una identidad de carga de trabajo. Segundo, los archivos locales, porque bloquean la migración: los contenedores son desechables, así que las subidas van al almacenamiento de objetos como servicio de respaldo. Tercero, los scripts puntuales: ejecútalos como procesos de administración separados del mismo lanzamiento, para que sean repetibles y auditados. Cuarto, los logs: escríbelos en stdout como un flujo. Primero el riesgo, luego los bloqueos y después la comodidad.

FUENTES

- The Twelve-Factor App: https://12factor.net/
- The Twelve-Factor App: III. Config: https://12factor.net/config
- Twelve-Factor App methodology is now open source (2024): https://www.12factor.net/blog/open-source-announcement
- Kubernetes documentation: Configure liveness, readiness and startup probes: https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/

Infraestructura como código e infraestructura inmutable

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.7 Servidores y la nube
- 0.8 Cómo se hace el software
- 10.2 Contenedores y Kubernetes

Términos anteriores que utiliza: Estado deseado, Bucle de reconciliación, Pull request, Radio de impacto (blast radius)

LA IDEA EN BREVE

Imagina que tienes que amueblar un hotel con doscientas habitaciones idénticas. Podrías decirle a cada decorador qué hacer y esperar que lo recuerde, o podrías escribir una especificación precisa de una habitación, guardarla en una carpeta y exigir que cada cambio se apruebe por escrito. Con la especificación, cualquier habitación puede comprobarse frente a ella, una habitación dañada puede reconstruirse exactamente igual y una nueva ala puede amueblarse en días. La infraestructura como código es esa especificación para las computadoras, las redes y las bases de datos sobre las que se ejecuta un sistema: escrita, revisada, versionada y aplicada por máquinas, no de memoria.

LO QUE NECESITAS SABER

- La infraestructura como código define la infraestructura en archivos con control de versiones que aplican herramientas, lo que hace que los entornos sean reproducibles, revisables, auditables y rápidos de crear.
- Las herramientas declarativas describen el estado final deseado y calculan los pasos, lo que hace idempotentes las ejecuciones repetidas; los scripts imperativos enumeran pasos y son difíciles de volver a ejecutar con seguridad.
- HashiCorp pasó Terraform a la Business Source License en agosto de 2023; le siguió la bifurcación OpenTofu bajo la Linux Foundation (estable en enero de 2024 y proyecto sandbox de la CNCF desde 2025); CloudFormation, Bicep, Pulumi y Crossplane son las principales alternativas.
- El estado asocia el código con los recursos reales; debe ser remoto, estar bloqueado, versionado, con acceso controlado y dividido para limitar el radio de impacto (blast radius), y la deriva debe detectarse con planes programados y después codificarse o revertirse.

- Los módulos versionados con interfaces pequeñas, reutilizados en los distintos entornos con entradas diferentes, evitan que los entornos hechos a base de copiar y pegar diverjan.
- La infraestructura inmutable sustituye los servidores y los contenedores a partir de imágenes nuevas en lugar de modificarlos in situ; la gestión de la configuración sigue siendo útil para los parques de sistemas de larga duración y para construir imágenes.
- GitOps guarda el estado deseado en Git y deja que unos agentes dentro del clúster (Argo CD, Flux) lo obtengan y lo reconcilien continuamente; la política como código y las pruebas por capas hacen seguros los cambios.

Qué es la infraestructura como código

La infraestructura como código (IaC) consiste en definir las redes, las máquinas virtuales, los clústeres, las bases de datos, las colas, los registros DNS, los permisos y cualquier otra pieza de infraestructura en archivos de texto guardados en control de versiones, y dejar que las herramientas creen y modifiquen los recursos reales a partir de esos archivos. En lugar de que un ingeniero vaya haciendo clic en una consola de la nube, un cambio es un commit: revisado en un pull request, probado en un pipeline, aplicado automáticamente y registrado para siempre.

La idea es anterior a la nube pública. CFEngine (1993), de Mark Burgess, gestionaba máquinas Unix a partir de políticas declarativas; Puppet (2005), Chef (2009) y Ansible (2012) generalizaron la gestión de la configuración. La nube la hizo imprescindible, porque la propia infraestructura pasó a estar disponible mediante API: AWS CloudFormation (2011) y Terraform (2014), de HashiCorp, describen entornos completos, no solo el software de los servidores. El libro de Kief Morris *Infrastructure as Code*, ya en su tercera edición (2025), es la referencia estándar sobre la práctica y sus principios.

Los beneficios son los de la ingeniería de software aplicada a la infraestructura. Los entornos se vuelven reproducibles: staging puede construirse a partir de las mismas definiciones que producción, y una región puede reconstruirse tras un desastre a partir del código y no de la memoria. Los cambios se vuelven revisables y auditables, algo que valoran los reguladores y los equipos de seguridad. El conocimiento deja de vivir en la cabeza de un solo ingeniero. Y la velocidad aumenta, porque crear un entorno para un nuevo servicio de reparto de comida pasa a ser un pull request en lugar de una solicitud que espera dos semanas.

Declarativo frente a imperativo

Un enfoque *imperativo* enumera los pasos: crear una red, luego una subred, luego una base de datos y después abrir este puerto. Ejecutar el script dos veces intenta crearlo todo dos veces. Un enfoque *declarativo* describe el estado final, el estado deseado de la Sección 10.2: una red

con estas subredes, una base de datos de este tamaño, un firewall que permite este puerto. La herramienta compara la descripción con lo que existe y deduce los pasos, de modo que aplicarla dos veces no cambia nada la segunda vez. Esa propiedad, la *idempotencia*, es lo que hace que el código de infraestructura pueda ejecutarse de forma segura una y otra vez, y de forma automatizada.

La mayoría de las herramientas maduras son declarativas en su núcleo, pero difieren en el lenguaje con el que se escribe la declaración y en quién lleva la cuenta de lo que existe. La tabla resume las principales familias.

Enfoques del código de infraestructura

Enfoque	Lo que escribes	Ejemplos	Ventajas	Puntos débiles
Scripts imperativos	Secuencias de llamadas a la CLI o al SDK	Scripts de shell que llaman a las CLI de la nube, programas con el SDK	Conocidos, flexibles, rápidos para tareas puntuales	No son idempotentes sin muchísimo cuidado; no hay registro de lo que existe; difíciles de cambiar con seguridad
Declarativo, lenguaje específico de dominio	Los recursos y sus relaciones en un lenguaje de configuración	Terraform y OpenTofu (HCL), AWS CloudFormation (YAML o JSON), Azure Bicep, manifiestos de Kubernetes	Planes legibles y revisables; idempotente; grandes ecosistemas de proveedores	Los bucles, las condiciones y la abstracción pueden resultar incómodos; hay que aprender un lenguaje
Modelo declarativo, lenguaje de propósito general	Código en TypeScript, Python, Go, C# o Java que construye un modelo declarativo	Pulumi, AWS Cloud Development Kit (CDK)	Abstracciones reales, herramientas de prueba y editores; conocido por los desarrolladores	Es fácil escribir infraestructura ingeniosa y difícil de revisar; el plan se ve menos en el código
Reconciliación continua	Recursos al estilo de Kubernetes que describen la infraestructura en la nube	Crossplane, controladores de Kubernetes de los proveedores de nube	El mismo bucle de reconciliación y las mismas herramientas que las aplicaciones; la deriva se corrige continuamente	Necesita un plano de control de Kubernetes; la depuración abarca dos sistemas
Gestión de la configuración	La configuración deseada del software en servidores existentes	Ansible, Chef, Puppet	Gestiona máquinas de larga duración, dispositivos de red y parques de sistemas locales (on-premises)	Los servidores acumulan historia y deriva; más lento y menos predecible que sustituirlos

Las herramientas y la bifurcación OpenTofu

Terraform se convirtió en la herramienta multinube más usada porque un solo lenguaje y un solo flujo de trabajo cubren miles de proveedores: AWS, Azure, Google Cloud, Cloudflare, Kubernetes, GitHub, servicios de monitorización y muchos más. El 10 de agosto de 2023, HashiCorp pasó Terraform de la licencia de código abierto Mozilla Public License 2.0 a la Business Source License 1.1, que permite el uso interno y comercial pero prohíbe ofrecerlo en productos que compitan con los de HashiCorp. La versión 1.5.7 fue la última lanzada con la licencia anterior.

La respuesta de la comunidad fue una bifurcación. A los pocos días apareció un manifiesto que pedía volver al código abierto; en septiembre de 2023, la Linux Foundation aceptó la bifurcación con el nombre de OpenTofu; su primera versión estable, la 1.6.0, llegó en enero de 2024; y en 2025 fue aceptada en la CNCF como proyecto sandbox. IBM completó la adquisición de HashiCorp en febrero de 2025. Las dos herramientas siguen siendo en gran medida compatibles en lenguaje y proveedores, pero sus funcionalidades divergen: OpenTofu añadió, por ejemplo, el cifrado del estado en el lado del cliente, mientras que la plataforma comercial de Terraform añadió capacidades propias. Para un arquitecto, el episodio recuerda que la licencia y la gobernanza de una herramienta forman parte de su perfil de riesgo, y que los estándares abiertos y una alternativa creíble mantienen ese riesgo bajo control.

Entre las herramientas para una sola nube, AWS CloudFormation (2011) y Bicep de Microsoft (presentado en 2020, que se compila a plantillas de Azure Resource Manager) los ejecuta el propio proveedor, que lleva la cuenta de lo que existe, así que no hay ningún archivo de estado que gestionar, a cambio de quedar atado a una sola nube. Pulumi (2018) y AWS CDK permiten a los desarrolladores escribir la infraestructura en lenguajes de propósito general; Crossplane amplía Kubernetes para gestionar recursos en la nube mediante reconciliación. La elección correcta depende de qué nubes uses, de las habilidades de tu equipo, de la madurez de los proveedores que necesites, de la política de licencias y de lo que ya utilice el resto de la organización; la coherencia entre equipos suele valer más que los méritos marginales de cualquier herramienta concreta.

```
terraform {
  required_version = "≥ 1.6"
  backend "gcs" {                 # remote state; this backend locks natively
    bucket = "fd-infra-state"
    prefix = "orders/prod"
  }
}

variable "environment" { type = string }

module "orders_db" {
  source      = "git::https://git.example.com/modules/postgres.git?ref=v3.2.0"
  name        = "orders-${var.environment}"
  size        = var.environment == "prod" ? "large" : "small"
  high_avail  = var.environment == "prod"   # standby in a second zone
  backup_days = 14
}

output "orders_db_host" {
  value = module.orders_db.host
}
```

OpenTofu o Terraform (HCL): estado remoto y un módulo interno versionado que oculta los detalles del proveedor detrás de una interfaz pequeña.

Ejecutar `tofu plan` (o `terraform plan`) muestra qué cambiaría antes de que ocurra nada. El plan es el artefacto más importante del flujo de trabajo: los revisores lo leen en el pull request, junto al código, y es lo que se aplica.

```
  # module.orders_db.google_sql_database_instance.this will be updated in-place
  ~ resource "google_sql_database_instance" "this" {
      ~ settings {
          ~ tier = "db-custom-2-8192" → "db-custom-4-16384"
        }
    }

Plan: 0 to add, 1 to change, 0 to destroy.
```

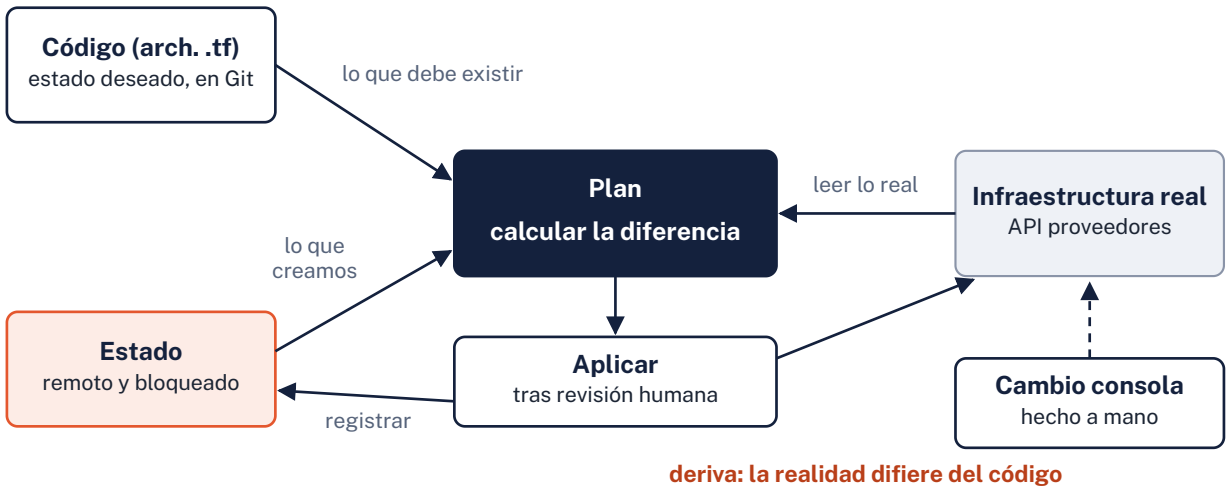
Un plan abreviado para cambiar el tamaño de la base de datos de pedidos de producción. La tilde indica una actualización in situ; un -/+ significaría destruir y volver a crear, lo que para una base de datos es una señal de alarma.

Estado, bloqueo y deriva

Las herramientas declarativas necesitan saber qué recursos reales corresponden a qué definiciones. Terraform y OpenTofu guardan esto en el estado: un archivo que asocia cada recurso del código con el identificador del objeto real, junto con sus últimos atributos

conocidos. Planificar compara tres cosas: el código (lo que debería existir), el estado (lo que creó la herramienta) y la infraestructura real, leída de la API del proveedor (lo que existe de verdad).

Plan, aplicación, estado y deriva



Un plan compara el código, el estado y la realidad; un cambio manual en la consola hace que la realidad se aparte de ambos, y el siguiente plan intentará deshacerlo.

El estado requiere cuidado porque es crítico y sensible. Debe guardarse en remoto (en almacenamiento de objetos o en un servicio gestionado), no en una computadora portátil, para que todo el equipo y el pipeline compartan una única versión. Debe estar *bloqueado* durante los cambios, para que dos aplicaciones no se intercalen y lo corrompan. Debe tener copias de seguridad y versiones, porque un archivo de estado perdido deja a la herramienta sin saber qué recursos creó. Y es un secreto: el estado contiene a menudo contraseñas de bases de datos y claves en texto plano, así que el acceso a él debe restringirse y debe cifrarse en reposo (OpenTofu también puede cifrarlo en el lado del cliente).

La deriva es cualquier diferencia entre el código y la realidad, normalmente por cambios manuales hechos con prisas, por otra herramienta que modifica el mismo recurso o por un fallo parcial durante la aplicación. La deriva es peligrosa porque la siguiente aplicación puede revertir en silencio un cambio del que alguien dependía, o fallar en producción por una diferencia cuya existencia nadie conocía. Los equipos la detectan ejecutando un plan de forma programada (por ejemplo, cada noche) y alertando ante cualquier cambio propuesto, y la previenen restringiendo el acceso de escritura a la consola a un procedimiento de emergencia (break-glass) registrado.

El estado también define el *radio de impacto*. Un único estado gigante para toda una empresa significa que cada plan es lento, cada aplicación lo arriesga todo y un solo bloqueo frena a todos los equipos. Dividir el estado por componente y entorno (la red, el clúster compartido, los recursos de cada servicio; producción separada de staging) mantiene los planes rápidos y los fallos contenidos, a cambio de tener que pasar salidas, como los identificadores de red, entre estados. La refactorización está soportada: los bloques `moved` renombran recursos sin destruirlos, e `import` pone bajo gestión los recursos existentes creados a mano.

EJEMPLO PRÁCTICO

Ejemplo práctico: una regla de firewall con deriva

Durante un incidente un viernes por la noche, un ingeniero abre a través de la consola de la nube el firewall de la base de datos de pedidos de producción a la dirección IP de un contratista, para que este ayude a diagnosticar una consulta lenta. Nadie actualiza el código.

Detección	El trabajo nocturno de detección de deriva ejecuta `tofu plan` para cada estado de producción. El sábado informa de *1 to change* para el firewall de la base de datos de pedidos y publica el plan en el canal del equipo de plataforma.
Diagnóstico	El plan muestra una dirección de origen permitida de más. El registro de auditoría de la nube identifica quién la añadió, cuándo y en qué incidente.
Decisión	El acceso era temporal y el incidente está cerrado, así que el código tiene razón y la realidad está equivocada. Aplicar el plan elimina la regla. Si el cambio hubiera sido necesario de forma permanente, la solución habría sido añadirlo al código mediante un pull request revisado.
Prevención	El acceso de escritura a la consola en producción se sustituye por un rol de emergencia (break-glass) que exige un motivo y genera una alerta al usarse. Una regla de política (ver más abajo) rechaza cualquier regla de firewall de base de datos abierta a todo internet.
Lo que muestra	La deriva no es solo un problema de orden: un agujero no documentado en un firewall es una exposición de seguridad y, sin detección, la siguiente aplicación no relacionada lo habría eliminado sin que nadie supiera por qué el contratista perdió el acceso.

La regla práctica: la realidad puede diferir del código durante minutos en una emergencia, pero la diferencia debe codificarse o revertirse antes del siguiente día laborable.

SI ESTO ES NUEVO PARA TI

Paso a paso: por qué la herramienta lleva un registro

El punto en el que más gente tropieza aquí es por qué herramientas como OpenTofu y Terraform guardan un estado. El código dice qué debería existir, pero no a qué recurso real de la cuenta en la nube se refiere. Sigue a la base de datos de pedidos de producción a lo largo de su vida.

- El equipo describe orders-prod en código, usando el módulo interno de base de datos. Todavía no existe nada, así que el estado está vacío y el primer plan propone añadirla.
- Al aplicar, la herramienta pide al proveedor que cree la base de datos. El proveedor le da un identificador, y la herramienta registra en el estado que este recurso del código es la base de datos real con ese identificador, junto con su configuración.
- Meses después, el equipo cambia el tamaño en el código. El plan lee el código, encuentra en el estado de qué base de datos real se trata, pide al proveedor su configuración actual y propone un único cambio in situ.
- Sin el estado, la herramienta no tendría constancia de que esta base de datos pertenece a este código, y propondría crearla de nuevo.
- El estado también revela lo que el código ya no menciona. Si el equipo elimina el módulo del código, el plan propone destruir la base de datos que el estado todavía registra.
- Por eso el estado se comparte, se bloquea, tiene copias de seguridad y se mantiene privado: dos aplicaciones a la vez podrían corromperlo, perderlo dejaría a la herramienta sin saber qué creó y a menudo contiene contraseñas.
- **La lección:** el código dice lo que debería existir, la infraestructura real es lo que existe y el estado vincula ambos. Lee cada plan teniendo en cuenta los tres, y trata un plan que destruye una base de datos como una señal de alarma.

Módulos y entornos

Un módulo es una unidad reutilizable de código de infraestructura con entradas y salidas: una base de datos PostgreSQL con copias de seguridad y monitorización, la red y el balanceador de carga de un servicio, un namespace estándar de Kubernetes con sus cuotas y políticas. Un buen módulo aplica la ocultación de información del Capítulo 3: una interfaz pequeña (nombre, tamaño, alta disponibilidad o no) delante de los muchos detalles específicos del proveedor que un equipo de servicio no debería tener que conocer. Los módulos se versionan y se lanzan como bibliotecas, y los entornos hacen referencia a una versión concreta, como hace el `ref=v3.2.0` del ejemplo, para que un cambio en un módulo llegue a producción solo cuando alguien lo actualice deliberadamente.

Los entornos deben construirse a partir de los mismos módulos con entradas diferentes, no a partir de código copiado. Una organización habitual es un directorio (o repositorio) por entorno y componente, cada uno con su propio estado, que llama a módulos compartidos y versionados; un cambio se promueve actualizando la versión o las entradas del módulo en staging y después en producción. Los entornos hechos a base de copiar y pegar divergen en cuestión de meses, hasta que staging deja de predecir lo que pasará en producción. Los workspaces, que dan a una configuración varios estados, encajan mejor con copias de corta duración, como los entornos de prueba por rama, que con producción y staging, que duran mucho.

Cuidado con el exceso de abstracción. Un módulo con cuarenta entradas booleanas que intenta servir a todos los equipos es tan difícil de cambiar como el código al que sustituyó. Es preferible tener unos pocos módulos con criterio propio que plasmen los estándares de la organización, que es exactamente lo que un equipo de plataforma ofrece como camino dorado en la Sección 10.7.

Infraestructura inmutable frente a gestión de la configuración

Las herramientas de gestión de la configuración mantienen servidores de larga duración en un estado descrito: Ansible se conecta por SSH y aplica los cambios sin agente; Chef y Puppet ejecutan agentes que obtienen su configuración y convergen hacia ella periódicamente. Funcionan, pero cada servidor acumula historia: paquetes instalados durante un incidente, archivos editados a mano, actualizaciones aplicadas a medias. Martin Fowler llamó al resultado *servidor copo de nieve* (snowflake server), único e imposible de reproducir, y propuso como remedio el *servidor fénix* (phoenix server), que se quema y se reconstruye desde cero con regularidad.

La infraestructura inmutable, un término popularizado por Chad Fowler en 2013, lleva el remedio hasta sus últimas consecuencias: no modificar nunca un servidor en ejecución. Cada cambio, ya sea un lanzamiento de la aplicación o un parche del sistema operativo, se hace construyendo una imagen nueva (una imagen de contenedor, o una imagen de máquina con una herramienta como Packer), probándola y sustituyendo las instancias antiguas, normalmente mediante un despliegue gradual o azul-verde. Los contenedores en Kubernetes son inmutables por defecto. Los beneficios son la ausencia de deriva, instancias idénticas, entornos trivialmente repetibles y el rollback mediante el redespliegue de la imagen anterior; los costos son un pipeline de imágenes, la necesidad de procesos desechables y sin estado (Sección 10.3) y una respuesta más lenta para un único cambio urgente.

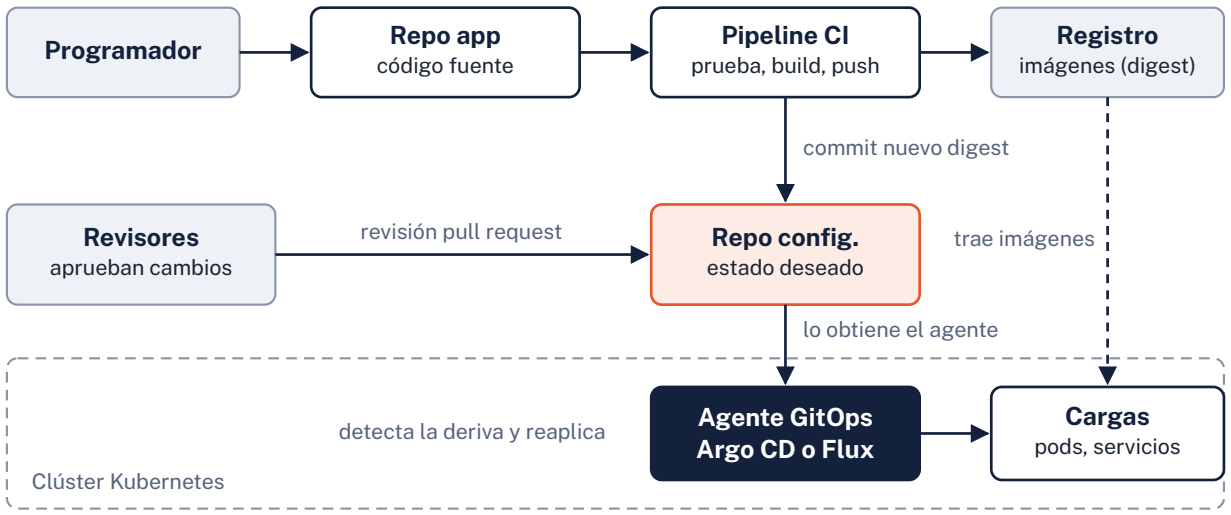
La gestión de la configuración no ha desaparecido. Sigue siendo la herramienta adecuada para los parques de sistemas que no pueden sustituirse en bloque: servidores locales (on-premises), equipos de red, bases de datos en hardware dedicado y las propias imágenes base, que a

menudo se construyen ejecutando Ansible dentro de la compilación de la imagen. La regla arquitectónica es llevar la mutabilidad lo más abajo y lo antes posible: configurar al construir las imágenes y después tratar lo que se ejecuta como de solo lectura.

GitOps

GitOps aplica el bucle de reconciliación de la Sección 10.2 a entornos completos. El término lo acuñó en 2017 Alexis Richardson, de Weaveworks, la empresa que creó la herramienta Flux. El proyecto OpenGitOps, un grupo de trabajo de la CNCF, define cuatro principios: el estado deseado es *declarativo*; está *versionado y es inmutable*, con un historial completo; lo *obtienen automáticamente* agentes de software; y se *reconcilia continuamente*, con agentes que observan el estado real y aplican el deseado. Los dos agentes principales, Argo CD y Flux, son proyectos graduados de la CNCF; Flux continuó como proyecto comunitario después de que Weaveworks cerrara en 2024.

Un flujo de entrega con GitOps



CI compila y publica las imágenes; Git guarda el estado deseado; un agente dentro del clúster lo obtiene, lo aplica y sigue corrigiendo la deriva.

El modelo pull tiene una ventaja de seguridad frente a un pipeline que empuja los cambios al clúster: el sistema de CI no necesita credenciales de producción, porque el agente dentro del clúster va a buscar lo que debe ejecutar. Git se convierte en el registro de auditoría y en la interfaz para los cambios, también para las operaciones: escalar un servicio o hacer rollback es un commit revisado o un revert. La deriva se corrige automáticamente, así que un `kubectl edit` manual dura solo hasta la siguiente sincronización.

GitOps trae sus propias preguntas de diseño. Los secretos no deben estar en Git en texto plano; los equipos los cifran (SOPS, Sealed Secrets) o guardan referencias que External Secrets Operator resuelve a partir de un gestor de secretos. La promoción entre entornos necesita una convención, como un directorio por entorno con pull requests que copian un nuevo digest de imagen de staging a producción. La corrección automática de la deriva debe pausarse deliberadamente durante algunas emergencias, y la estructura del repositorio debe reflejar la propiedad para que los equipos puedan cambiar sus propios servicios sin tocar los de otros. GitOps es lo más natural para Kubernetes, pero los mismos principios se aplican a la infraestructura en la nube mediante Crossplane o mediante pipelines que planifican y aplican el código de infraestructura a partir de los commits fusionados.

Política como código

La política como código expresa las reglas sobre la infraestructura como código comprobable por máquina en lugar de como una página de wiki: los buckets de almacenamiento no deben ser públicos, las bases de datos deben estar cifradas y tener copias de seguridad, todo recurso debe llevar una etiqueta de propietario y de centro de costos, solo pueden usarse regiones aprobadas, los contenedores no deben ejecutarse como root, las imágenes de producción deben venir del registro de la empresa. Las comprobaciones se ejecutan en varios puntos: sobre el plan en el pull request (Open Policy Agent con Conftest, Checkov, HashiCorp Sentinel en sus productos comerciales), en la admisión en un clúster de Kubernetes (OPA Gatekeeper, Kyverno) y de forma continua sobre los recursos en ejecución (los servicios de políticas de los proveedores de nube).

La política como código convierte la gobernanza, que deja de ser una reunión de revisión, en una salvaguarda rápida y coherente que se explica a sí misma en el pull request. Funciona mejor cuando las políticas son pocas, están claramente justificadas, versionadas y probadas como cualquier código, cuando empiezan en modo de advertencia antes de bloquear y cuando existe un proceso de excepciones registrado. El Capítulo 11 trata cómo encajan estas comprobaciones automatizadas en la gobernanza de la arquitectura (gobernanza en organizaciones ágiles).

Probar el código de infraestructura, de lo rápido a lo lento

1. **Comprobaciones estáticas en cada commit**
   El formateo, `validate`, el linting (por ejemplo, TFLint) y los escáneres de seguridad detectan en segundos errores de sintaxis, argumentos obsoletos y configuraciones erróneas evidentes.

2. **Comprobaciones de políticas sobre el plan**
   Genera el plan como JSON y evalúalo frente a las políticas de la organización; los revisores ven las infracciones en el pull request.

3. **Pruebas de módulos**
   `terraform test` y `tofu test` ejecutan aserciones sobre un módulo, con proveedores simulados para pruebas rápidas de tipo unitario o con proveedores reales para pruebas de integración.

4. **Pruebas de integración en entornos efímeros**
   Aplica el módulo o el entorno a una cuenta desechable, comprueba que funciona (una base de datos acepta conexiones, un servicio responde) y después destrúyelo. Herramientas como Terratest automatizan esto; es lento y cuesta dinero, así que resérvalo para los módulos compartidos.

5. **Revisión del plan y despliegue por etapas**
   Aplica primero en staging y después en producción, a partir de las mismas versiones; una persona lee cada plan de producción, sobre todo las destrucciones y las sustituciones.

6. **Detección de deriva y simulacros de restauración**
   Los planes programados detectan la deriva; reconstruir periódicamente un entorno a partir del código demuestra que la fuente de verdad es el código, no la memoria.

Lista de comprobación: código de infraestructura listo para producción

Úsala al revisar el repositorio de infraestructura de un equipo o un nuevo módulo de la plataforma.

- [ ] **Todo está en código**
  Ningún recurso de producción existe solo porque alguien hizo clic; los recursos existentes creados a mano se han importado.
- [ ] **Estado remoto, bloqueado y cifrado**
  Backend compartido con bloqueo, versionado y acceso restringido; estado dividido por componente y entorno para limitar el radio de impacto.
- [ ] **Planes revisados**
  Cada cambio es un pull request que muestra el plan; las destrucciones y sustituciones de recursos con estado requieren una aprobación explícita.
- [ ] **Módulos versionados**
  Los entornos fijan las versiones de los módulos; staging y producción usan los mismos módulos con entradas diferentes, no copias.
- [ ] **Políticas aplicadas automáticamente**
  Las reglas de cifrado, exposición de red, etiquetado y regiones se comprueban en cada plan y en la admisión al clúster.
- [ ] **Secretos fuera del código y de Git**
  Las credenciales vienen de un gestor de secretos o de la identidad de carga de trabajo; el acceso al estado se trata como acceso a secretos.
- [ ] **Pruebas en el pipeline**
  Comprobaciones estáticas y de políticas en cada cambio; pruebas de módulos para los módulos compartidos; pruebas de integración para los críticos.
- [ ] **Deriva detectada y gestionada**
  Los planes programados alertan de las diferencias; el acceso de escritura a la consola es un procedimiento de emergencia (break-glass) registrado.
- [ ] **Inmutable siempre que sea posible**
  Los servidores y los contenedores se sustituyen a partir de imágenes en lugar de parchearse in situ; la configuración se hace al construir la imagen.
- [ ] **Recuperación ensayada**
  Se ha reconstruido un entorno a partir del código dentro del tiempo de recuperación acordado, incluida la restauración de datos (ver la Sección 10.6).

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué es importante la idempotencia en el código de infraestructura?**

Porque la automatización vuelve a ejecutar el código a menudo. Una herramienta declarativa e idempotente no cambia nada cuando la realidad ya coincide con el código, así que volver a ejecutarla es seguro; un script no idempotente puede intentar crearlo todo otra vez.

▼ **Nombra las tres cosas que compara un plan y qué representa cada una.**

El código (lo que debería existir), el estado (lo que la herramienta creó y controla) y la infraestructura real leída de la API del proveedor (lo que existe de verdad).

▼ **¿Por qué no debería guardarse el estado como un único archivo para toda una organización?**

Un único estado significa planes lentos, un solo bloqueo que frena a todos los equipos y un radio de impacto que lo abarca todo; dividirlo por componente y entorno mantiene los cambios rápidos y los fallos contenidos.

▼ **¿Qué ventaja de seguridad tiene el modelo pull de GitOps frente a un pipeline que empuja los cambios al clúster?**

El sistema de CI no necesita credenciales de producción, porque un agente dentro del clúster obtiene el estado deseado de Git y lo aplica.

▼ **¿Cuándo sigue siendo una buena opción la gestión de la configuración?**

Para máquinas de larga duración que no pueden sustituirse sin más, como los servidores locales (on-premises) y los dispositivos de red, y para configurar las imágenes base durante su compilación.

▼ **Inténtalo: Un equipo guarda su archivo de estado de infraestructura en la computadora portátil de un ingeniero, con todos los entornos en un solo estado, y creó staging copiando el código de producción y editándolo. Enumera los riesgos y di cómo lo reestructurarías.**

El estado podría perderse, nadie más puede ejecutar cambios con seguridad y dos personas podrían colisionar. Un solo estado también significa que un error puede dañar todos los entornos. El código copiado derivará. Mueve el estado a un almacén remoto, con bloqueo, versionado y control de acceso, dividido por entorno y componente para limitar el radio de impacto. Construye módulos versionados con interfaces pequeñas y úsalos en cada entorno con entradas diferentes. Añade planes programados para detectar la deriva.

FUENTES

- OpenTofu: https://opentofu.org/
- Terraform documentation: State: https://developer.hashicorp.com/terraform/language/state
- Terraform documentation: Tests: https://developer.hashicorp.com/terraform/language/tests
- OpenGitOps: GitOps principles: https://opengitops.dev/
- Martin Fowler: PhoenixServer: https://martinfowler.com/bliki/PhoenixServer.html

SECCIÓN 10.5 DE 10.8 · INTERMEDIO · 25 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Pipelines de CI/CD y estrategias de despliegue

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.8 Cómo se hace el software
- 2.6 Modificabilidad y desplegabilidad
- 7.7 Evolución de esquemas

Términos anteriores que utiliza: Imagen de contenedor, Expandir y contraer (expand and contract), Objetivo de nivel de servicio (SLO), Monolito distribuido

LA IDEA EN BREVE

Piensa en una cadena de panaderías que lanza un pan nuevo. Hornea un lote de prueba, lo comprueba en su propio obrador y luego lo vende en una sola tienda durante una mañana antes de enviarlo a todas las tiendas. Si los clientes se quejan, solo se han visto afectados los clientes de una tienda y el pan de siempre sigue en los estantes. Un pipeline de software es la misma rutina, automatizada: cada cambio se compila una sola vez, se comprueba por etapas, se prueba con unos pocos usuarios y solo se extiende cuando las pruebas indican que es bueno. Un interruptor aparte decide cuándo ven realmente los clientes una nueva funcionalidad.

LO QUE NECESITAS SABER

- Un pipeline de despliegue (Humble y Farley, 2010) hace pasar cada commit por etapas automatizadas, las más rápidas primero, y un fallo en cualquier etapa detiene ese cambio y avisa a su autor.
- Build once, deploy many (compilar una vez, desplegar muchas): empaqueta cada cambio en un único artefacto inmutable, guárdalo en un repositorio de artefactos y promueve ese mismo artefacto por todos los entornos, cambiando solo la configuración.
- El desarrollo basado en trunk (trunk-based development) mantiene las ramas de corta duración y los cambios pequeños, lo que hace real la integración continua y facilita revisar, probar y deshacer cada cambio.
- Los despliegues por recreación (recreate), graduales (rolling), azul-verde (blue-green), canario (canary) y en sombra (shadow) cambian infraestructura y complejidad adicionales por una menor exposición de los usuarios y un rollback más rápido; todos salvo la recreación exigen que las versiones antigua y nueva sean compatibles.

- Los feature flags separan el despliegue del lanzamiento; las categorías de Hodgson son toggles de lanzamiento, de experimento, de operaciones y de permisos, y cada flag tiene un costo hasta que se elimina.
- Los cambios de base de datos deben ser compatibles hacia atrás (expandir y contraer) y ejecutarse como pasos separados del pipeline, porque los datos no se pueden revertir como el código.
- La entrega progresiva automatiza las decisiones sobre el canario comparándolo estadísticamente con una línea base recién creada; la respuesta por defecto ante un mal lanzamiento es revertir primero y diagnosticar después.

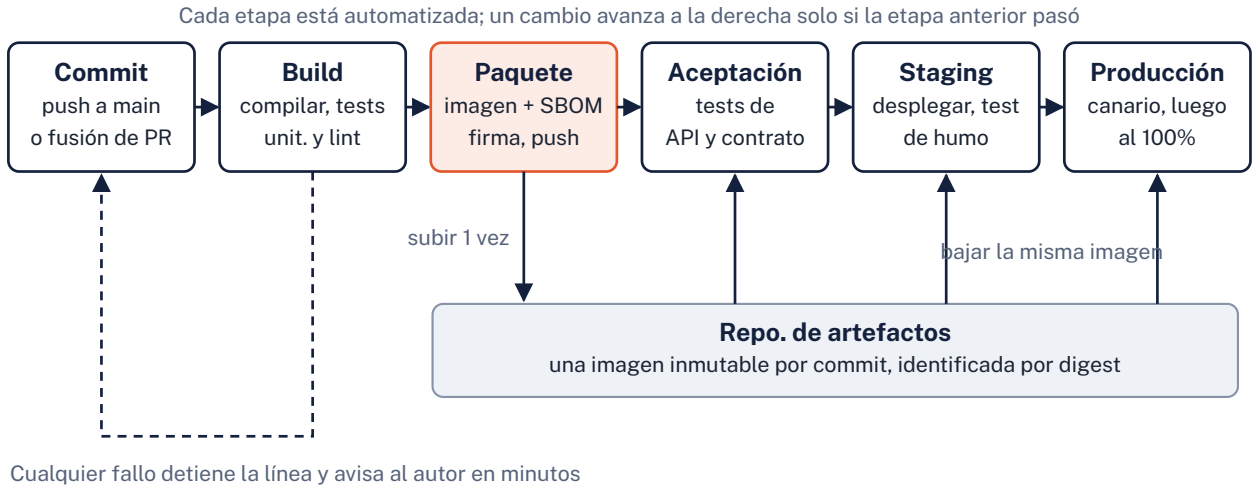
Del commit a producción: para qué sirve un pipeline

La Sección 0.8 (cómo se hace el software) presentó la integración continua (cada cambio se fusiona y se verifica automáticamente), la entrega continua (la rama principal está siempre lista para lanzarse) y el despliegue continuo (cada cambio que supera las comprobaciones pasa a producción sin una decisión humana). Esta sección los trata como arquitectura: el diseño del camino que sigue un cambio y las estrategias que lo llevan a producción sin perjudicar a los usuarios.

La idea central es el pipeline de despliegue, descrito por Jez Humble y David Farley en *Continuous Delivery* (2010). Cada commit desencadena una secuencia automatizada de etapas. Las primeras etapas son rápidas y baratas y detectan la mayoría de los problemas; las posteriores son más lentas, más realistas y más caras. Un cambio pasa a la siguiente etapa solo si la anterior se superó, de modo que cuanto más avanza una compilación, más confianza se ha ganado. Un fallo en cualquier punto detiene esa compilación y vuelve directamente a la persona que hizo el cambio, mientras aún lo tiene fresco en la mente.

El pipeline importa a los arquitectos por tres razones. Primero, es donde la desplegabilidad, un atributo de calidad de la sección 2.6, se logra o se pierde de verdad. Segundo, su velocidad limita la rapidez con la que la organización puede responder a un cambio del mercado o a una vulnerabilidad de seguridad: si una corrección de una línea tarda dos días en llegar a producción, la arquitectura tiene dos días de lentitud, se vea como se vea el código. Tercero, la forma del sistema condiciona la forma del pipeline: los servicios desplegables de forma independiente pueden tener pipelines independientes, pero un monolito distribuido obliga a lanzamientos coordinados por muy ingeniosas que sean las herramientas.

Las etapas de un pipeline típico



Un artefacto se compila y se sube una sola vez; cada etapa posterior despliega exactamente ese artefacto, y cualquier fallo detiene la línea.

Qué hace cada etapa y cuánto debería tardar

1. **Etapa de commit (objetivo: menos de 10 minutos)**
   Compila, ejecuta las pruebas unitarias, los linters, el análisis estático y los escaneos de vulnerabilidades de las dependencias. Es la etapa que esperan los desarrolladores, así que debe ser rápida; las pruebas lentas van después.

2. **Empaquetado**
   Construye el artefacto desplegable (normalmente una imagen de contenedor), genera una lista de materiales de software (SBOM), fírmalo y súbelo al repositorio de artefactos. Los controles de la cadena de suministro se tratan en la sección 8.6.

3. **Aceptación e integración**
   Despliega el artefacto en un entorno de pruebas temporal o compartido y ejecuta pruebas de API, pruebas de contrato dirigidas por el consumidor y unos pocos recorridos de extremo a extremo. Las pruebas de rendimiento y de seguridad pueden ejecutarse aquí en paralelo o de forma programada.

4. **Staging**
   Despliega en un entorno configurado como producción, aplica las migraciones de base de datos y ejecuta pruebas de humo. Su valor depende de lo mucho que se parezca a producción en configuración, forma de los datos y dependencias.

5 **Producción**
Despliega con una estrategia que limite la exposición (gradual, azul-verde o canario), verifica con métricas reales frente a los objetivos de nivel de servicio y luego completa o revierte automáticamente.

Build once, deploy many y el repositorio de artefactos

Una regla de *Continuous Delivery* que se sigue incumpliendo con sorprendente frecuencia es build once, deploy many (compilar una vez, desplegar muchas): compila y empaqueta cada cambio exactamente una vez y luego promueve ese mismo artefacto por todos los entornos. Si un equipo recompila para staging y otra vez para producción, lo que probó no es lo que entregó. Un parche distinto del compilador, una dependencia resuelta durante la noche a una versión más reciente o una imagen base cambiada pueden hacer que las dos compilaciones difieran, y las pruebas no demuestran nada sobre la segunda.

Lo que cambia entre entornos es la configuración, proporcionada en el momento del despliegue o del arranque (direcciones de bases de datos, credenciales, ajustes de los feature flags), nunca incorporada al artefacto. Es el factor de configuración de la aplicación de doce factores. Una imagen de contenedor etiquetada `orders:4f1c2ab` (el hash del commit) se despliega en pruebas, staging y producción; solo cambian sus variables de entorno y los secretos montados.

El repositorio de artefactos es el almacén entre la compilación y el despliegue: un registro de contenedores (Amazon ECR, Azure Container Registry, Google Artifact Registry, GitHub Container Registry, Harbor) o un repositorio universal como JFrog Artifactory o Sonatype Nexus para paquetes y binarios. Trátalo como infraestructura crítica. Despliega por digest inmutable (`sha256:...`) en lugar de por una etiqueta mutable como `latest` , que puede moverse para apuntar a otra imagen; conserva los artefactos que están o estuvieron en producción durante todo el tiempo que puedas necesitar volver a ellos o investigarlos; y registra la procedencia, es decir, qué commit, qué ejecución del pipeline y qué dependencias produjeron cada artefacto.

Desarrollo basado en trunk y lotes pequeños

La forma de usar las ramas decide con qué frecuencia se integra realmente el código. En el desarrollo basado en trunk (trunk-based development) todos integran cambios pequeños en una única rama principal (el trunk) al menos una vez al día, ya sea haciendo commit directamente o mediante ramas de corta duración y pull requests que viven horas, no semanas. El programa de investigación DORA ha asociado repetidamente las ramas de corta duración y

las fusiones frecuentes con un mayor rendimiento de entrega, y es una condición previa para la integración continua en sentido literal: el código que pasa tres semanas en una rama no se está integrando de forma continua.

Las ramas de funcionalidad de larga duración parecen más seguras porque el trabajo sin terminar se mantiene aparte, pero el riesgo solo se aplaza: cuanto más divergen dos ramas, mayor y más arriesgada es la fusión final, y más tiempo ve el pipeline un código que no es el que se va a entregar. GitFlow, un modelo de ramas con ramas develop y release de larga duración publicado por Vincent Driessen en 2010, se adoptó ampliamente; en 2020 Driessen añadió una nota según la cual los equipos que entregan de forma continua, como los de aplicaciones web, deberían considerar un flujo de trabajo más sencillo. Los equipos que sí necesitan trabajar en cambios grandes en el trunk usan dos técnicas: los feature flags, que ocultan el comportamiento sin terminar, y branch by abstraction, que introduce una interfaz delante de la implementación antigua, construye la nueva detrás de ella y hace el cambio cuando está lista.

Los profesionales discrepan de verdad en los extremos. Los proyectos de código abierto con muchos colaboradores no fiables dependen de pull requests más largos y de la revisión, y los equipos regulados a veces necesitan ramas de lanzamiento explícitas para la auditoría. El principio que sobrevive al desacuerdo es el tamaño del lote: cuanto más pequeño es cada cambio, más fácil es revisarlo, probarlo, desplegarlo, diagnosticarlo y deshacerlo.

```
name: delivery
on:
  push:
    branches: [main]
env:
  IMAGE: registry.example.com/food/orders
jobs:
  build:
    runs-on: ubuntu-latest
    outputs:
      digest: ${{ steps.push.outputs.digest }}
    steps:
      - uses: actions/checkout@v4
      - run: make lint test              # fast feedback first
      - id: push
        run: |
          docker build -t "$IMAGE:${{ github.sha }}" .
          docker push "$IMAGE:${{ github.sha }}"
          ref=$(docker inspect --format '{{index .RepoDigests 0}}' "$IMAGE:${{ github.sha
}}")
          echo "digest=${ref#*@}" >> "$GITHUB_OUTPUT"   # the image's sha256 digest
  staging:
    needs: build
    runs-on: ubuntu-latest
    environment: staging
    steps:
      - uses: actions/checkout@v4
      - run: ./deploy.sh staging "$IMAGE@${{ needs.build.outputs.digest }}" && ./smoke-
test.sh staging
  production:
    needs: [build, staging]
    runs-on: ubuntu-latest
    environment: production            # can require an approval
    steps:
      - uses: actions/checkout@v4
      - run: ./deploy.sh production "$IMAGE@${{ needs.build.outputs.digest }}" --canary 5
```

Un flujo de trabajo mínimo de entrega continua con la sintaxis de GitHub Actions: compilar y probar una vez, subir una imagen etiquetada con el commit y luego desplegar esa misma imagen por su digest en staging y después en producción como canario. GitLab CI, Azure Pipelines, Jenkins y otros expresan las mismas etapas. Se omiten el inicio de sesión en el registro y los scripts de despliegue.

Estrategias de despliegue

Una estrategia de despliegue decide cómo sustituye la nueva versión a la antigua en producción y, por tanto, a cuántos usuarios puede perjudicar una versión defectuosa antes de que se detenga. Las estrategias difieren en la infraestructura adicional que necesitan, en la

rapidez del rollback y en si se ejecutan dos versiones a la vez. Siempre que se ejecutan dos versiones a la vez, deben ser compatibles con los datos y los mensajes de la otra, y ese es el precio de todas las estrategias salvo la recreación.

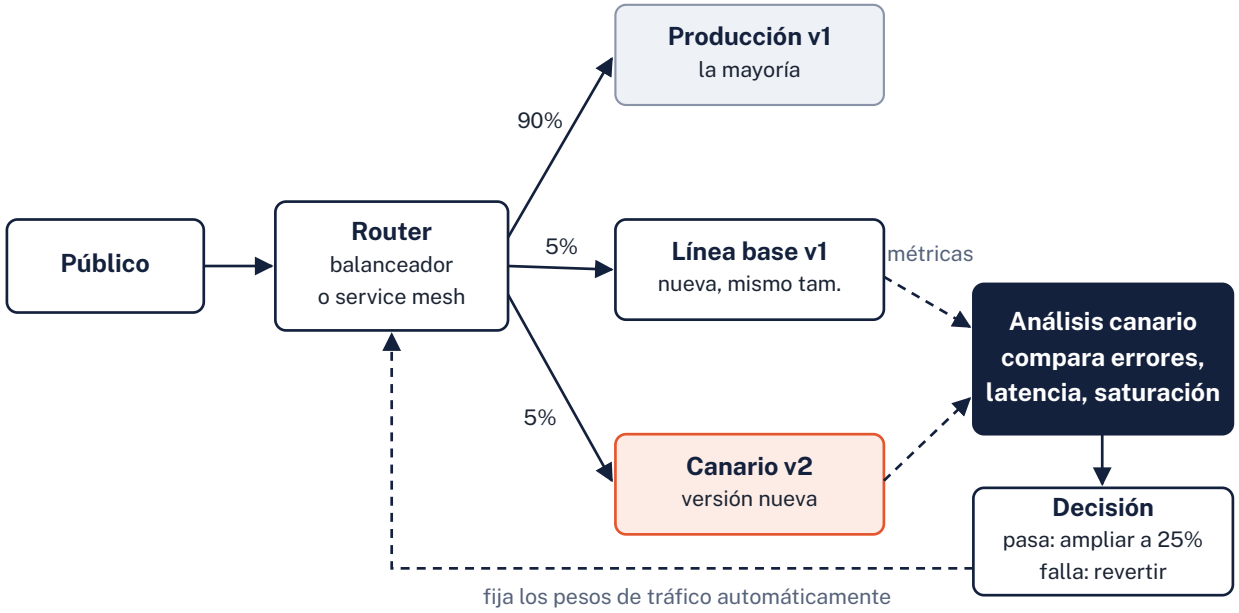
Comparación de las estrategias de despliegue

Estrategia	Cómo funciona	Riesgo para los usuarios	Rollback	Costo y complejidad adicionales
Recreación (recreate)	Detener todas las instancias antiguas y luego arrancar las nuevas.	Tiempo de inactividad durante el cambio; todos los usuarios se encuentran a la vez con una versión defectuosa.	Volver a desplegar la versión antigua, con más tiempo de inactividad.	Ninguno. Aceptable para herramientas internas, trabajos por lotes y ventanas de mantenimiento.
Gradual (rolling)	Sustituir las instancias unas pocas a la vez (la opción por defecto de Kubernetes para los Deployments, con maxSurge y maxUnavailable ambos en un 25% por defecto).	La exposición crece a medida que avanza el despliegue; las comprobaciones de estado detectan las caídas, pero no los errores sutiles.	Se revierte desplegando de nuevo gradualmente la versión antigua, lo que tarda tanto como el despliegue.	Bajo. Las versiones antigua y nueva se ejecutan en paralelo, así que deben ser compatibles.
Azul-verde (blue-green)	Ejecutar un entorno nuevo completo (verde) junto al que está en producción (azul), probarlo y luego cambiar todo el tráfico en el router.	Todos los usuarios pasan a la vez, pero el entorno nuevo se verificó antes del cambio.	Rápido: volver a cambiar el router mientras el azul sigue en marcha.	Hasta el doble de capacidad durante el cambio; las bases de datos compartidas deben funcionar con ambas versiones.
Canario (canary)	Enviar una pequeña parte del tráfico (por ejemplo, un 1-5%) a la nueva versión, comparar sus métricas y luego ampliar por pasos.	Solo la parte de usuarios del canario se encuentra con una versión defectuosa, y solo hasta que el análisis la detiene.	Rápido: poner a cero el peso del canario.	Necesita enrutamiento ponderado, buenas métricas y suficiente tráfico para que la comparación tenga sentido.

Estrategia	Cómo funciona	Riesgo para los usuarios	Rollback	Costo y complejidad adicionales
En sombra (dark launch)	Copiar solicitudes reales a la nueva versión; sus respuestas se comparan o se descartan, nunca se devuelven.	Ninguno directamente, siempre que los efectos secundarios (pagos, correos electrónicos, escrituras) se supriman o se simulen.	No hay nada que revertir; los usuarios nunca dependieron de ella.	El doble de carga en la ruta en sombra y un manejo cuidadoso de los efectos secundarios; ideal para rutas de lectura y reescrituras.

Martin Fowler describió el despliegue azul-verde en 2010, atribuyéndolo a Dan North y Jez Humble; el lanzamiento canario, llamado así por los canarios que llevaban los mineros para detectar gas, lo documentó Danilo Sato en el mismo sitio en 2014. En Kubernetes, las actualizaciones graduales vienen integradas; los lanzamientos azul-verde y canario suelen venir de un controlador de entrega progresiva (Argo Rollouts, Flagger) o de un service mesh y del enrutamiento ponderado de la Gateway API. En las plataformas gestionadas, las funciones equivalentes se llaman división del tráfico (traffic splitting) o ranuras de despliegue (deployment slots).

Lanzamiento canario con análisis automatizado



Comparar el canario con una línea base nueva de la versión antigua, en lugar de con la flota que lleva tiempo en marcha, elimina las diferencias causadas solo por el tiempo de actividad y las cachés ya calientes.

SI ESTO ES NUEVO PARA TI

Paso a paso: una versión defectuosa, cuatro formas de desplegarla

La idea más difícil de esta sección es que la estrategia de despliegue decide a cuántos clientes llega una versión defectuosa antes de detenerla y con qué rapidez se puede deshacer. Sigue una versión defectuosa del servicio de pedidos a través de cuatro estrategias.

- La nueva versión tiene un bug sutil: no se cae, pero cobra una tarifa de envío incorrecta en algunos pedidos. Las comprobaciones de estado pasan, porque el proceso en sí está sano.
- Recreación: todas las copias antiguas se detienen y luego arrancan las nuevas. Tras una breve interrupción, todos los clientes se encuentran a la vez con la versión defectuosa. Deshacerlo significa volver a desplegar la versión antigua, con otra interrupción.
- Gradual: las copias se sustituyen unas pocas a la vez. Como nada se cae, las comprobaciones de estado dejan que el despliegue continúe hasta que todas las copias son defectuosas. Revertir tarda tanto como desplegar.
- Azul-verde: la nueva versión se ejecuta como un segundo entorno completo y se prueba antes de pasarle todo el tráfico. Si las pruebas no detectan el bug, todos los clientes pasan a la vez, pero deshacerlo es rápido: se vuelve a cambiar el router al entorno antiguo, que sigue en marcha.
- Canario: el 5% del tráfico va a la nueva versión, y sus métricas se comparan con una copia nueva de la versión antigua. Si la comparación incluye una señal de negocio, como las tarifas cobradas, puede detectar la diferencia, y la parte del canario se pone a cero. Solo los clientes de ese 5% se encontraron con el bug, y solo hasta que el análisis lo detuvo.
- **La lección:** elige una estrategia según a cuántos clientes puedes permitirte exponer y con qué rapidez debes deshacer un error, y asegúrate de que tus comprobaciones pueden ver los fallos que temes, no solo las caídas.

Separar el despliegue del lanzamiento con feature flags

El **despliegue** instala una nueva versión del software en un entorno; el **lanzamiento** pone un comportamiento a disposición de los usuarios. Tratarlos como un solo acontecimiento convierte cada despliegue en un acontecimiento de negocio, con todo el nerviosismo que eso conlleva. Los feature flags (feature toggles) los separan: el código nuevo se despliega desactivado, y el comportamiento se lanza más tarde, de forma gradual y reversible, cambiando la

configuración. El artículo de Pete Hodgson *Feature Toggles (aka Feature Flags)* en martinfowler.com (2017) distingue cuatro categorías según cuánto tiempo viven y con qué dinamismo deciden.

Las cuatro categorías de feature toggle de Hodgson

Categoría	Propósito	Vida típica	Grado de dinamismo
Toggles de lanzamiento	Entregar en el trunk funcionalidades incompletas o no lanzadas, desactivadas.	De días a unas pocas semanas; se eliminan tras el lanzamiento.	Mayormente estáticos en cada versión.
Toggles de experimento	Pruebas A/B o multivariantes que dirigen a cada usuario a una cohorte.	El tiempo suficiente para obtener resultados estadísticamente sólidos; luego se eliminan.	Por solicitud, según la cohorte del usuario.
Toggles de operaciones	Controles operativos, como desactivar un costoso panel de recomendaciones cuando hay mucha carga (interruptores de emergencia o kill switches).	Mayormente cortos; unos pocos interruptores de emergencia se mantienen a largo plazo.	Se cambian en tiempo de ejecución, a menudo durante un incidente.
Toggles de permisos	Funcionalidades para usuarios concretos: niveles premium, probadores beta, personal interno.	Pueden vivir años.	Por solicitud, según el usuario.

Los flags tienen un costo de mantenimiento. Cada flag duplica los caminos por el código que protege, añade combinaciones de prueba y deja código muerto una vez que su decisión está tomada. Hodgson recomienda tratar los flags como un inventario: añadir una tarea de eliminación cuando se crea un toggle de lanzamiento, dar a los flags fechas de caducidad, añadir pruebas que fallen cuando un flag ha pasado su fecha y limitar cuántos pueden existir a la vez. La pérdida de Knight Capital en 2012, descrita en la sección 1.3, tuvo que ver con un flag reutilizado que reactivó código antiguo en un servidor que se había saltado un despliegue manual: un recordatorio de no reutilizar nunca los nombres de los flags y de borrar el código muerto que hay detrás de los flags ya decididos. A escala, los flags los sirve un servicio dedicado (opciones de código abierto como Unleash y Flagsmith, comerciales como LaunchDarkly, o la API neutral respecto al proveedor OpenFeature) para que un cambio se propague en segundos y quede auditado.

Cambios de base de datos en el pipeline

El código se puede revertir en segundos; los datos, no. Un despliegue que renombra una columna y luego se revierte deja al código antiguo frente a un esquema que no entiende. La solución, tratada en profundidad en evolución de esquemas, es hacer que cada cambio de esquema sea compatible hacia atrás con la versión del código que se está ejecutando, usando expandir y contraer: añade primero la nueva estructura, despliega código que escriba en ambas y lea de cualquiera, rellena los datos (backfill), cambia los lectores y elimina la estructura antigua solo en un lanzamiento posterior, cuando ya nada la use.

En el pipeline, esto significa que las migraciones son artefactos versionados que se ejecutan como un paso propio, antes de desplegar el código que las necesita, y que se prueban con una copia de los datos del tamaño de producción, para que un bloqueo en una tabla grande se descubra en staging y no a la hora de comer. Los pasos destructivos (eliminar columnas o tablas) se retrasan y se ejecutan por separado, porque son el único tipo de cambio que un rollback no puede deshacer. Una comprobación útil del pipeline hace fallar la compilación cuando una migración no es aditiva, salvo que esté marcada explícitamente como paso de contracción.

Entrega progresiva y análisis canario automatizado

La entrega progresiva, un término popularizado por James Governor, de la firma de análisis RedMonk, en 2018, combina canarios, feature flags y verificación automatizada: un cambio llega a públicos cada vez mayores y, en cada paso, es el sistema, no una persona que mira paneles, quien decide si continuar. La decisión usa el análisis canario automatizado: las métricas del canario (tasa de errores, percentiles de latencia, saturación, señales de negocio como los pedidos realizados) se comparan estadísticamente con una línea base. Netflix y Google publicaron como código abierto su motor de análisis, Kayenta, en 2018; Argo Rollouts y Flagger ofrecen pasos de análisis similares en Kubernetes.

Tres detalles deciden si se puede confiar en el análisis. Compara lo comparable: una línea base nueva de la versión antigua que recibe la misma parte del tráfico evita culpar a la nueva versión de diferencias causadas por cachés frías o reinicios recientes. Ten suficiente tráfico: con el 5% de un servicio que atiende diez solicitudes por minuto, una hora de canario ve treinta solicitudes, demasiado pocas para detectar una tasa de errores del 1%, así que los servicios con poco tráfico necesitan pasos más largos, carga sintética o una estrategia distinta. Elige métricas ligadas a los objetivos de nivel de servicio y a los resultados de negocio, porque una versión puede ser rápida y no tener errores mientras cobra en silencio una tarifa de envío incorrecta.

Rollback o roll forward

Cuando un lanzamiento sale mal, hay dos salidas. El rollback vuelve al artefacto anterior, lo que es rápido y no necesita código nuevo, siempre que la versión anterior siga funcionando con los datos actuales y con las versiones de los demás servicios. El roll forward (corrección hacia adelante) entrega una corrección a través del pipeline, lo que es más lento, pero a veces es la única opción, por ejemplo tras una migración de datos que no se puede revertir o un cambio del que ya dependen otros servicios.

Un buen criterio por defecto es “revertir primero, diagnosticar después”: restablece el servicio con la acción segura más rápida y luego investiga sin presión. Corregir hacia adelante bajo presión tiende a saltarse las comprobaciones del pipeline, y una corrección precipitada de un incidente en producción es una fuente clásica del siguiente incidente. El rollback solo es una opción real si se diseña desde el principio: conservar los artefactos anteriores, hacer que los cambios de esquema sean compatibles en ambas direcciones durante al menos una versión, que los mensajes los puedan leer consumidores antiguos y nuevos, y ejercitar con regularidad el propio camino de rollback. Un equipo que nunca ha hecho un rollback debería suponer que no puede hacerlo.

EJEMPLO PRÁCTICO

Ejemplo práctico: lanzar una nueva regla de tarifa de envío

El servicio de pedidos de la plataforma de reparto de comida va a cambiar la forma de calcular las tarifas de envío, usando la distancia y la demanda en lugar de una tarifa fija. El equipo usa desarrollo basado en trunk, un despliegue canario y un flag de lanzamiento.

Cambio	Nuevo calculador de tarifas detrás de un flag de lanzamiento `dynamic-delivery-fee`, desactivado por defecto; una migración solo de expansión añade una columna que admite nulos, `fee_rule_version`.
Del commit al artefacto	Se fusiona en main a las 10:02. Lint, pruebas unitarias y escaneos tardan 6 minutos; la imagen `orders:4f1c2ab`, con su SBOM y su firma, se sube a las 10:09.
Migración	La migración aditiva se ejecuta a las 10:15 como paso propio del pipeline. La versión en ejecución ignora la nueva columna, así que nada se rompe.
Staging	La misma imagen se despliega a las 10:16; las pruebas de contrato con los servicios de pagos y de restaurantes y las pruebas de humo pasan a las 10:31.
Canario	A las 10:35, el 5% del tráfico de producción va a la nueva versión, todavía con el flag desactivado. El análisis durante 20 minutos compara la latencia p99, la tasa de 5xx y el éxito de los pedidos con una línea base nueva: pasa.
Despliegue progresivo	Los pesos pasan al 25%, al 50% y al 100% en pasos de 15 minutos; termina a las 11:40. El despliegue está completo y, sin embargo, ningún cliente ha visto la nueva tarifa.
Lanzamiento	Al día siguiente, el responsable de producto activa el flag para el 10% de los clientes de una ciudad y vigila la conversión de pedidos y las quejas; luego lo amplía a todas las ciudades a lo largo de una semana.
Limpieza	El flag y el código de la tarifa fija se eliminan en el sprint siguiente; una migración de contracción posterior hace obligatoria la columna.

El riesgo del despliegue (¿funciona el código?) y el riesgo del lanzamiento (¿les gustan a los clientes las nuevas tarifas?) se gestionaron por separado, cada uno con su propio radio de impacto reducido.

En cada paso hasta la migración de contracción, la recuperación era un cambio de configuración o de peso en el router, no una corrección de emergencia. Del commit al despliegue completo en producción pasaron alrededor de 1 hora y 40 minutos.

Medir el pipeline

Las métricas DORA de entrega de software presentadas en la sección 2.6 (tiempo de entrega de los cambios, frecuencia de despliegue, tasa de fallos de los cambios, tiempo de recuperación de un despliegue fallido y tasa de retrabajo de despliegues) son en gran medida propiedades del pipeline y de la arquitectura en conjunto. Los lotes pequeños y el desarrollo basado en trunk acortan el tiempo de entrega y aumentan la frecuencia; las pruebas automatizadas y el análisis canario reducen la tasa de fallos de los cambios; el rollback rápido y los flags acortan el tiempo de recuperación. Mídelas por servicio, observa las tendencias en lugar de valores aislados y nunca las conviertas en objetivos individuales, lo que solo invita a manipularlas.

ATENCIÓN

Errores comunes en los pipelines

- Recompilar el artefacto para cada entorno, de modo que producción ejecuta algo que nunca se probó.
- Una etapa de commit que tarda 40 minutos, de modo que los desarrolladores agrupan cambios para no esperar, lo que anula su propósito.
- Pruebas inestables (flaky) que se repiten hasta que salen en verde, lo que enseña a todos a ignorar las compilaciones en rojo.
- Aprobaciones manuales en cada etapa que añaden días de espera sin aportar información.
- Migraciones incluidas en el arranque de la aplicación, de modo que todas las instancias compiten por aplicarlas y un rollback se encuentra con un esquema incompatible.
- Feature flags que nunca se eliminan y dejan cientos de combinaciones sin probar.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué es un problema recompilar el artefacto por separado para producción aunque el código fuente sea idéntico?

Las dependencias, las imágenes base o las herramientas de compilación pueden haber cambiado entre compilaciones, así que el artefacto de producción no es el que se probó; compila una vez y promueve el mismo artefacto.

▼ ¿Cuál es la diferencia entre desplegar una funcionalidad y lanzarla?

El despliegue instala el código en un entorno; el lanzamiento pone el comportamiento a disposición de los usuarios. Los feature flags permiten desplegar el código desactivado y lanzarlo más tarde, de forma gradual y reversible.

▼ ¿Por qué un canario en un servicio con poco tráfico da a menudo resultados poco fiables?

Una pequeña parte de poco tráfico produce demasiado pocas solicitudes para detectar diferencias en la tasa de errores o en la latencia con confianza estadística, así que hay que alargar los pasos, usar tráfico sintético o elegir otra estrategia.

▼ ¿Cuándo puede ser corregir hacia adelante la única opción?

Cuando el lanzamiento incluyó un cambio irreversible, como una migración destructiva, o cuando otros servicios ya dependen del nuevo comportamiento, de modo que la versión anterior ya no puede funcionar correctamente.

▼ Inténtalo: Un equipo recompila la imagen para cada entorno, despliega deteniendo todo y reiniciando a la hora del almuerzo, y la nueva versión elimina una columna de la base de datos en el mismo lanzamiento. Encuentra tres problemas y propón un enfoque mejor.

Recompilar significa que en producción se ejecuta algo que nunca se ha probado, así que compila una vez y promueve un único artefacto inmutable. Recreate a la hora del almuerzo causa tiempo de inactividad en el pico, así que usa un despliegue gradual o canary, con comparación automática frente a una línea base. Eliminar la columna en el lanzamiento hace imposible el rollback, y las versiones antigua y nueva no pueden funcionar juntas, así que usa expandir y contraer, eliminándola en un lanzamiento posterior. Añade un feature flag para separar el despliegue del lanzamiento.

FUENTES

- Pete Hodgson: Feature Toggles (aka Feature Flags), martinfowler.com: https://martinfowler.com/articles/feature-toggles.html
- Martin Fowler: BlueGreenDeployment: https://martinfowler.com/bliki/BlueGreenDeployment.html
- Danilo Sato: CanaryRelease: https://martinfowler.com/bliki/CanaryRelease.html
- Trunk Based Development: https://trunkbaseddevelopment.com/
- DORA: software delivery performance metrics: https://dora.dev/guides/dora-metrics-four-keys/

SECCIÓN 10.6 DE 10.8 · AVANZADO · 24 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Multirregión, recuperación ante desastres y edge computing

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.5 Disponibilidad y fiabilidad
- 6.6 CAP y consistencia
- 7.4 Replicación y particionamiento

Términos anteriores que utiliza: Failover (conmutación por error), Retardo de replicación (replication lag), PACELC, Radio de impacto (blast radius)

LA IDEA EN BREVE

Una cadena de tiendas que guarda todas sus existencias en un solo almacén tiene un problema si ese almacén se inunda. Podría guardar solo un inventario y una llave de repuesto de un edificio vacío en otro lugar (barato, pero lento para recuperarse), mantener un segundo almacén pequeño abastecido y con personal (más caro, pero rápido) o tener dos almacenes completos funcionando todo el tiempo (lo más caro, casi sin interrupción). También puede poner pequeños quioscos cerca de los clientes con los artículos más populares. El diseño multirregión y en el borde toma las mismas decisiones para el software: cuánto pagar por adelantado frente a cuánto tiempo pueden esperar los clientes.

LO QUE NECESITAS SABER

- Opera en varias regiones solo por latencia, recuperación ante desastres o residencia de datos; varias zonas de disponibilidad en una región son la base normal y resultan mucho más baratas.
- El RTO (cuánto tiempo puedes estar sin servicio) y el RPO (cuántos datos puedes perder) los fija el negocio y deciden la estrategia de recuperación ante desastres y su costo.
- Las estrategias de AWS Well-Architected, copia de seguridad y restauración (backup and restore), pilot light, warm standby y activo-activo multisitio (multi-site active-active), compran un RTO y un RPO progresivamente menores a cambio de un costo continuo y una complejidad mayores.
- Los datos entre regiones son la parte difícil: la replicación síncrona añade largos viajes de ida y vuelta a las escrituras, el retardo de la replicación asíncrona se convierte en el RPO efectivo, y la replicación copia los borrados y la corrupción, así que las copias de seguridad siguen siendo esenciales.

- El failover lo dirigen el DNS o balanceadores de carga globales con anycast; los diseños con estabilidad estática mantienen capacidad aprovisionada de antemano y evitan depender del plano de control de la región caída.
- La arquitectura basada en células y el shuffle sharding limitan el radio de impacto de los malos despliegues, las sobrecargas y las entradas envenenadas que también sufriría una segunda región.
- Las CDN y las funciones en el borde, como Cloudflare Workers, que se ejecuta en isolates de V8, sirven lecturas y lógica ligera cerca de los usuarios, mientras que los sistemas de registro suelen quedarse en una región.

Por qué operar en más de una región

La Sección 0.7 (servidores y la nube) presentó las regiones (zonas geográficas separadas) y las zonas de disponibilidad (grupos independientes de centros de datos dentro de una región), y la sección 10.1 mostró cómo las organizan los proveedores. Operar en varias zonas de disponibilidad de una región es la base normal de cualquier sistema en producción: sobrevive a la pérdida de un centro de datos con poco costo adicional, porque las zonas están lo bastante cerca para la replicación síncrona. Operar en varias *regiones* es un paso mucho mayor, y solo hay tres buenas razones para darlo.

Latencia. La luz en la fibra óptica viaja a unos 200 000 km por segundo, así que un viaje de ida y vuelta entre Londres y Sídney (unos 17 000 km en línea recta) no puede tardar menos de unos 170 ms, y las rutas reales tardan bastante más. Una solicitud que necesita varios viajes de ida y vuelta a una región lejana se percibe lenta, por rápidos que sean los servidores. Servir a los usuarios desde una región cercana, o desde el borde, es la única solución.

Recuperación ante desastres. Las regiones enteras sí fallan, normalmente por una dependencia compartida más que por un desastre físico. El 19 y 20 de octubre de 2025, una condición de carrera en la automatización que gestiona los registros DNS de Amazon DynamoDB dejó su endpoint de US-EAST-1 con un registro DNS vacío; los errores de DynamoDB duraron unas tres horas, pero los fallos derivados al lanzar nuevas máquinas virtuales y en las comprobaciones de estado de los balanceadores de carga se prolongaron hasta la tarde siguiente. Todos los grandes proveedores han tenido incidentes regionales comparables. La recuperación ante desastres se prepara para ellos.

Residencia de datos. Los requisitos de residencia de datos, procedentes de la ley, de los reguladores o de los contratos con clientes, pueden exigir que ciertos datos se almacenen o se traten en jurisdicciones concretas. El propio RGPD (GDPR) restringe las transferencias de datos personales fuera del Espacio Económico Europeo en lugar de exigir el almacenamiento local,

pero las normas sectoriales, las leyes nacionales y los contratos empresariales a menudo van más allá (consulta privacidad desde el diseño). La residencia obliga a menudo a diseños multirregión en los que los datos deliberadamente *no* se replican en todas partes.

La multirregión es cara: infraestructura duplicada, cargos por transferencia de datos entre regiones, una consistencia de los datos más difícil y unas operaciones y pruebas más complejas. Muchos equipos que creen necesitarla necesitan en realidad un buen diseño multizona, copias de seguridad probadas y una CDN.

Objetivos de recuperación y las cuatro estrategias

La Sección 2.5 (disponibilidad y fiabilidad) definió los dos números que guían todo diseño de recuperación ante desastres. El objetivo de tiempo de recuperación (RTO) es cuánto tiempo puede estar el servicio no disponible; el objetivo de punto de recuperación (RPO) es cuántos datos recientes se pueden perder, expresado como tiempo. Ambos los fija el negocio, idealmente a partir del costo de una hora de interrupción y de un pedido perdido, no de las preferencias de los ingenieros. Los valores más bajos cuestan más, y el costo se dispara cerca de cero.

El whitepaper de AWS Well-Architected *Disaster Recovery of Workloads on AWS: Recovery in the Cloud* describe cuatro estrategias, en orden creciente de costo y complejidad. Los nombres se usan ampliamente más allá de AWS y se aplican igual a Azure, a Google Cloud o a un par de centros de datos privados.

Estrategias de recuperación ante desastres



Cada paso a la derecha mantiene en marcha una parte mayor del sistema en la segunda región, y compra un RTO y un RPO menores a cambio de un costo continuo mayor.

Comparación de las estrategias de recuperación ante desastres

Estrategia	Qué se ejecuta en la segunda región	RTO típico	RPO típico	Costo relativo y riesgo principal
Copia de seguridad y restauración	Nada más que copias de seguridad (instantáneas de la base de datos, almacenamiento de objetos) y el código de infraestructura para reconstruir.	Horas, según el volumen de datos y la automatización.	El intervalo entre copias (minutos con copias continuas del registro, horas con instantáneas diarias).	El más bajo. Las restauraciones son lentas y a menudo no se han probado; sin infraestructura como código, el RTO es ficción.
Pilot light (piloto encendido)	Bases de datos replicadas e infraestructura básica; servidores de aplicaciones definidos, pero no en ejecución.	Decenas de minutos: arrancar y escalar el nivel de aplicación y luego redirigir el tráfico.	De segundos a minutos (retardo de replicación).	Bajo. Escalar durante una crisis regional compite con todos los demás por la capacidad.
Warm standby (espera semiactiva)	Una copia reducida, pero totalmente funcional, de todo el sistema.	Minutos: redirigir el tráfico y luego escalar.	De segundos a minutos.	Moderado. La reserva se puede probar con tráfico real, pero se desvía si no se despliega con cada lanzamiento.
Activo-activo multisitio	Una copia completa que sirve tráfico real en cada región.	Casi cero: el tráfico se redistribuye, sin necesidad de failover.	Casi cero para los datos replicados; la corrupción sigue necesitando copias de seguridad.	El más alto. Requiere un diseño de datos para escrituras concurrentes en varias regiones.

El propio whitepaper da una única cifra aproximada que abarca el RTO y el RPO de cada estrategia; la tabla los separa porque, con replicación continua, el RPO de pilot light y de warm standby lo determina el retardo de replicación y no cuánta parte de la aplicación está en marcha. Hay dos advertencias que se aplican a todas las columnas. La replicación no es una copia de seguridad: un borrado accidental, una mala migración o un ransomware se replican fielmente en segundos, así que hacen falta copias de seguridad de un momento dado, idealmente inmutables y en una cuenta separada, incluso en un diseño activo-activo. Y una

estrategia que no se ha ensayado es una esperanza, no un plan. Los equipos que hacen un ejercicio de failover regional al menos unas cuantas veces al año descubren permisos que faltan, endpoints escritos en el código, certificados caducados y cuotas demasiado pequeñas en la región de recuperación, mientras todavía es barato encontrarlos.

Replicar datos entre regiones

Los servidores de aplicaciones sin estado son fáciles de ejecutar en dos lugares; los datos son la parte difícil, y los compromisos son los de la consistencia, CAP y PACELC a distancias continentales. La replicación síncrona entre regiones significa que cada escritura espera al menos un viaje de ida y vuelta largo, quizá de 70 a 150 ms o más, lo que es aceptable para unas pocas escrituras críticas y ruinoso para las que son muy frecuentes. La replicación asíncrona mantiene rápidas las escrituras, pero hace que la réplica vaya con retraso, y todo lo que no haya llegado cuando falle la región primaria se pierde: el retardo de replicación *es* el RPO efectivo.

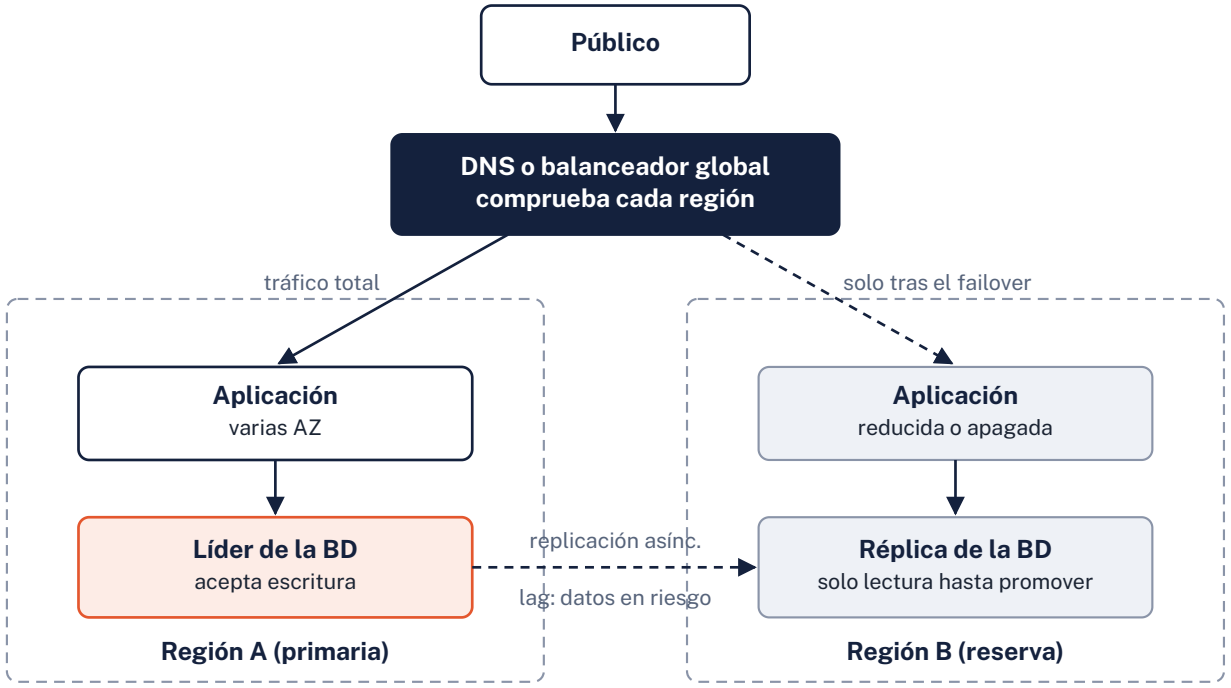
Hay cuatro diseños de datos habituales. **Región primaria única**: todas las escrituras van a una región y las réplicas de otros lugares sirven lecturas y recuperación; es sencillo, pero los usuarios lejanos pagan la latencia de escritura y el failover implica promover una réplica (con los riesgos de split brain descritos en replicación y particionamiento). **Región de origen por entidad**: cada cliente, restaurante o tenant tiene una región de origen donde se escriben sus datos, lo que mantiene locales las escrituras y facilita la residencia, a costa del enrutamiento y de algunas operaciones ocasionales entre regiones. **Múltiples escritores con resolución de conflictos**: todas las regiones aceptan escrituras y los conflictos se fusionan, por ejemplo con last-writer-wins (la opción por defecto en las tablas globales de DynamoDB, que desde 2025 también ofrecen consistencia fuerte multirregión) o con tipos de datos replicados libres de conflictos; solo es seguro para datos en los que fusionar tiene sentido, como un carrito, y es peligroso para el dinero o el inventario. **Bases de datos globalmente consistentes** como Google Spanner o CockroachDB pagan la latencia del consenso en las escrituras para ofrecer consistencia fuerte entre regiones; Azure Cosmos DB permite a cada aplicación elegir entre cinco niveles de consistencia, de fuerte a eventual.

La mayoría de los sistemas reales combinan estos diseños según el conjunto de datos, tal como recomienda la sección 6.6: los pagos y el estado de los pedidos en una región primaria única o de origen, las cartas y los datos de los restaurantes replicados en todas partes para las lecturas, y la analítica enviada de forma asíncrona.

Failover, DNS y estabilidad estática

El tráfico se dirige entre regiones mediante el DNS o un balanceador de carga global. Con el failover (conmutación por error) basado en DNS, el servicio DNS comprueba el estado de cada región y devuelve la dirección de una que esté sana; su debilidad es la caché, porque los resolvedores y los clientes pueden conservar respuestas antiguas durante más tiempo que el TTL del registro. Los balanceadores de carga globales (por ejemplo, AWS Global Accelerator, Azure Front Door, el balanceador de carga externo global de Google Cloud o Cloudflare) usan direcciones anycast anunciadas desde muchas ubicaciones, de modo que los clientes mantienen una sola dirección y el proveedor desplaza el tráfico en segundos.

Failover entre una región primaria y una de reserva



Lo que no se haya replicado cuando falle la región A se pierde, así que el retardo de replicación fija el punto de recuperación real.

Decidir *cuándo* hacer el failover es más difícil que el mecanismo. El failover automático reacciona más rápido, pero puede dispararse por un fallo de la monitorización o por un breve corte de red, y hacer failover de una base de datos corre el riesgo de que dos primarios acepten escrituras. Por eso muchas organizaciones automatizan la detección y los pasos del failover, pero mantienen una decisión humana para iniciar un failover regional, con un runbook y un límite de tiempo. Volver a la región original tras el incidente (failback), incluida la resincronización de los datos escritos en la región de recuperación, también necesita un plan.

El incidente de octubre de 2025 ilustra un principio que la AWS Builders' Library llama estabilidad estática: un sistema debe seguir funcionando durante un fallo sin tener que hacer cambios en ese momento. Los sistemas que ya tenían suficiente capacidad en marcha superaron el problema; los que necesitaban lanzar nuevas instancias, o cuya recuperación dependía de acciones del plano de control en la región afectada, no pudieron recuperarse hasta que lo hizo la región. Para la recuperación ante desastres, esto aboga por la capacidad aprovisionada de antemano (warm standby en lugar de pilot light, para las rutas críticas), por un failover que dependa solo del plano de datos de la región sana, y por un DNS, una identidad y unos secretos que no vivan únicamente en la región de la que se huye.

SI ESTO ES NUEVO PARA TI

Paso a paso: recuperarse sin pedir nada

La idea que requiere más atención aquí es la estabilidad estática: un plan de recuperación solo es tan bueno como lo que necesita en el momento del fallo, y el plan más seguro no necesita nada nuevo. Compara dos formas de preparar una segunda región para el núcleo de pedidos de la plataforma de reparto de comida.

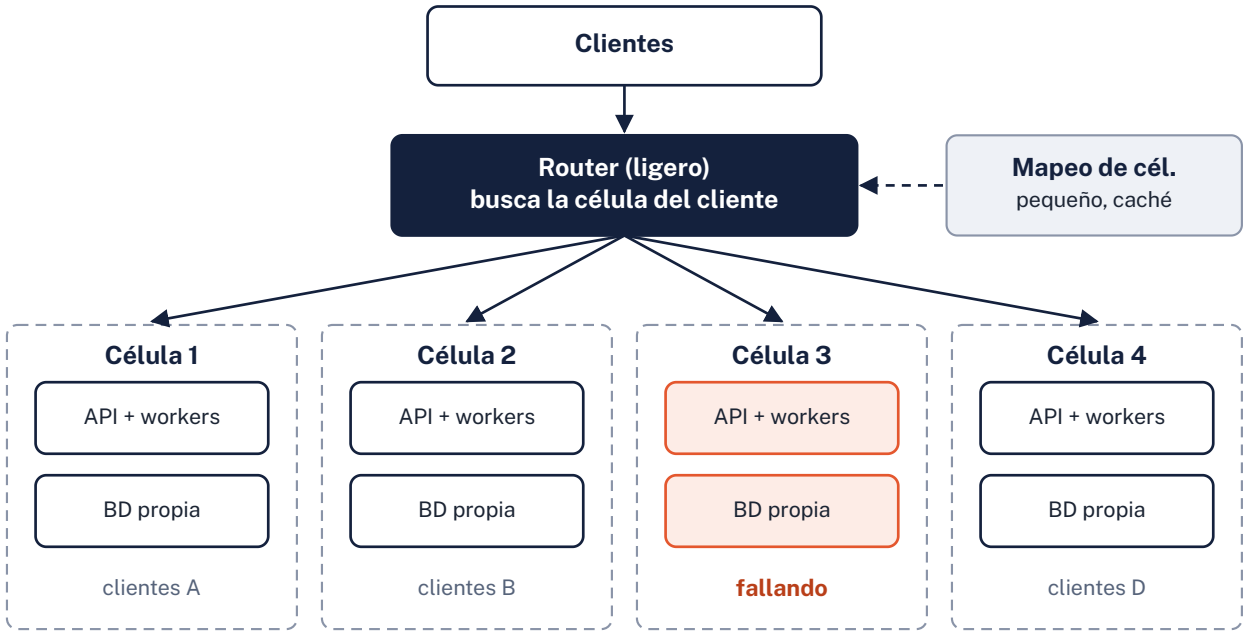
- Con un pilot light, la segunda región guarda una copia de los datos actualizada continuamente y la infraestructura básica, pero los servidores de aplicaciones están apagados. Con un warm standby, ya se ejecuta allí una pequeña copia en funcionamiento del núcleo de pedidos, que recibe un poco de tráfico real.
- A la hora de la cena falla la región primaria. Con pilot light, recuperarse significa hacer cambios en ese momento: arrancar los servidores de aplicaciones, escalarlos y luego redirigir el tráfico. Cada arranque es una solicitud al plano de control del proveedor, la maquinaria que crea y modifica los recursos.
- Es el peor momento para pedir. Otros clientes quieren capacidad al mismo tiempo, y cualquier paso que dependa de algo guardado solo en la región caída, como el DNS, la identidad o los secretos, debe esperar a que esa región se recupere.
- Con warm standby, la copia ya está en marcha y probada con tráfico real. El tráfico se redirige a ella de inmediato; sirve con capacidad reducida mientras escala. Si el escalado es lento, los clientes reciben un servicio reducido en lugar de ninguno.
- La estabilidad estática cuesta dinero: una copia que funciona todo el tiempo cuesta más cada mes que los servidores apagados de pilot light, como muestra el ejemplo práctico más adelante en esta sección.
- **La lección:** para las rutas críticas, prefiere una recuperación que dependa solo de lo que ya está en marcha en la región sana. Cualquier cosa que haya que crear o modificar durante el desastre puede no estar disponible cuando se necesite.

Células: limitar el radio de impacto

Las regiones protegen frente a desastres regionales; sin embargo, la mayoría de las interrupciones las causan los propios cambios del sistema, las sobrecargas y las entradas envenenadas, que una segunda región con el mismo código y la misma configuración también sufrirá. La arquitectura basada en células limita el radio de impacto (blast radius) de esos fallos ejecutando muchas copias completas e independientes del sistema, llamadas células, cada una de las cuales atiende a un subconjunto fijo de clientes. Un router ligero asigna cada solicitud a su célula, normalmente por el identificador del cliente o del tenant. Las células no comparten nada en tiempo de ejecución: cada una tiene sus propias instancias de aplicación, colas y bases de datos.

Si una célula sufre un mal despliegue, un cliente con demasiada carga o una corrupción de datos, solo se ven afectados los clientes de esa célula: con 20 células, alrededor del 5% de ellos. Los despliegues avanzan célula a célula en oleadas, así que un mal lanzamiento se detecta en la primera célula. El diseño se describe en el whitepaper de AWS Well-Architected *Reducing the Scope of Impact with Cell-Based Architecture* (2023) y lo usan a gran escala los propios servicios de AWS y empresas como Slack. Una técnica relacionada, el shuffle sharding, da a cada cliente una combinación aleatoria de unos pocos workers entre muchos (con 8 workers y 2 por cliente hay 28 combinaciones), de modo que un cliente cuyas solicitudes hacen caer a los workers solo tumba su propia combinación, y la mayoría de los demás clientes comparten como mucho un worker con él.

Arquitectura basada en células



Un fallo, mal despliegue o cliente ruidoso en la célula 3 afecta a un cuarto de los clientes, no a todos

El router debe ser el componente más sencillo y fiable del sistema, porque es lo único que comparten todas las células.

Las células tienen un costo: más instancias y bases de datos que operar, operaciones entre células (informes sobre todos los clientes, mover un cliente de una célula a otra) que necesitan una maquinaria aparte, y un router y un servicio de mapeo que deben ser extremadamente sencillos y fiables. Encajan en grandes sistemas multitenant con claves de partición naturales; para una aplicación interna de un solo tenant son un exceso de ingeniería.

El borde: CDN y funciones en el borde

Entre los usuarios y las regiones está el borde (edge): los cientos de ubicaciones que operan las redes de distribución de contenidos como Akamai, Cloudflare, Fastly, Amazon CloudFront y Azure Front Door. Una CDN almacena en caché el contenido estático y cacheable (imágenes, scripts, páginas de cartas) cerca de los usuarios, siguiendo las reglas de caché de HTTP descritas en caché, y absorbe los picos de tráfico y muchos ataques de denegación de servicio antes de que lleguen al origen.

Las funciones en el borde (edge functions) van más allá y ejecutan código en esas ubicaciones: Cloudflare Workers, Fastly Compute, CloudFront Functions y Lambda@Edge, Akamai EdgeWorkers, y las funciones en el borde de plataformas como Vercel y Netlify. Cloudflare Workers ejecuta el código en isolates ligeros de V8, los entornos aislados que usan los motores de JavaScript para las pestañas del navegador, en lugar de en un contenedor por función, así

que el arranque es muy rápido y muchos tenants comparten un proceso de forma segura. Este mismo curso se ejecuta en Cloudflare Workers. Buenos usos son el enrutamiento de solicitudes y la asignación de pruebas A/B, las comprobaciones de autenticación y la validación de tokens, las reglas de cabeceras y redirecciones, el redimensionado de imágenes, la personalización sencilla y los proxies de API que mantienen las claves secretas fuera del navegador.

Los límites son arquitectónicos. Las funciones en el borde tienen límites estrictos de tiempo de CPU y de memoria; el almacenamiento en el borde suele tener consistencia eventual (Workers KV de Cloudflare puede tardar un tiempo en propagar una escritura a todo el mundo) u ofrece consistencia fuerte solo para un único objeto coordinador, como hacen los Durable Objects; y el sistema de registro normalmente se queda en una región. Por eso un patrón habitual es *leer en el borde, escribir en el origen*: servir las cartas, las páginas de restaurantes y las comprobaciones de sesión desde el borde, y enviar los pedidos y los pagos a la región de origen.

GUÍA DE DECISIÓN

Guía de decisión: ¿cuánta multirregión necesitas?

Responde para cada recorrido de usuario importante, no para todo el sistema a la vez.

Q1 **¿Puede el negocio tolerar que este recorrido no esté disponible durante varias horas mientras se reconstruye una región a partir de las copias de seguridad?**

- **Sí** Usa varias zonas de disponibilidad en una región más copias de seguridad y restauración entre regiones, probadas y automatizadas. Revísalo si el negocio cambia.
- **No** Continúa: hace falta una reserva en otra región.

Q2 **¿Es aceptable un tiempo de recuperación de decenas de minutos, y puedes contar con obtener capacidad durante un incidente regional?**

- **Sí** Pilot light: replica los datos de forma continua, ten lista la infraestructura como código y automatiza el escalado y el cambio de tráfico.
- **No** Continúa: la capacidad ya debe estar en marcha.

Q3 **¿Es aceptable un tiempo de recuperación de unos pocos minutos, con capacidad reducida al principio?**

- **Sí** Warm standby: ejecuta una pequeña copia en funcionamiento, despliégala con cada lanzamiento y envíale un poco de tráfico real para demostrar que funciona.
- **No** Activo-activo, con un diseño de datos para escrituras multirregión (regiones de origen o una base de datos globalmente consistente); presupuesta la ingeniería, no solo la infraestructura.

Q4 **¿Necesitan los usuarios lejanos escrituras rápidas, o deben algunos datos quedarse en su jurisdicción?**

- **Sí** Particiona por geografía: asigna a los clientes una región de origen y conserva allí sus datos, en lugar de replicarlo todo en todas partes.
- **No** Sirve las lecturas de los usuarios lejanos mediante una CDN y funciones en el borde, y mantén las escrituras en una región primaria.

EJEMPLO PRÁCTICO

Ejemplo práctico: elegir una estrategia de DR para la plataforma de reparto de comida

La plataforma opera en una región con tres zonas de disponibilidad. El núcleo de pedidos (pedidos, pagos, despacho) cuesta unos 60 000 al mes en infraestructura. Gestiona unos 40 000 pedidos al día, con picos de 6000 por hora a la hora de la cena, y unos 5 de ingresos para la plataforma por pedido. La dirección pide un plan de recuperación para una interrupción regional. Todas las cifras son ilustrativas y están en libras.

Costo de una interrupción	Los ingresos son de unos 8000 por hora de media a lo largo del día y llegan a 30 000 en el pico. La pérdida de confianza de clientes, restaurantes y repartidores duplica aproximadamente el costo real. Los contratos con grandes cadenas de restaurantes prometen que la recepción de pedidos se restablece en menos de una hora.
Requisito	RTO de 1 hora o menos y RPO de menos de 1 minuto para el núcleo de pedidos; RTO de 24 horas para la analítica, las recomendaciones y el marketing.
Copia de seguridad y restauración	Unos 2000 adicionales al mes. Reconstrucción medida: de 4 a 6 horas (domina la restauración de una base de datos de 2 TB). No cumple el RTO de 1 hora.
Pilot light (piloto encendido)	Unos 8000 adicionales al mes (réplica continua de la base de datos, red, imágenes listas). Failover ensayado: de 35 a 50 minutos, sobre todo para escalar el nivel de aplicación. Cumple el RTO solo si se puede obtener capacidad durante un evento regional.
Warm standby (espera semiactiva)	Unos 18 000 adicionales al mes (réplica más la aplicación al 20% de escala). Failover ensayado: unos 10 minutos, y luego escalado mientras sirve. Cumple el RTO con margen.
Activo-activo	Unos 45 000 adicionales al mes, más un estimado de seis meses-ingeniero para rediseñar los datos de pedidos y pagos para escrituras multirregión. RTO casi cero, muy por encima del requisito.
Decisión	Warm standby para el núcleo de pedidos; copia de seguridad y restauración para la analítica, las recomendaciones y el marketing; cartas y páginas de restaurantes servidas desde la CDN para que se pueda seguir navegando durante el failover. Costo adicional total de unos 20 000 al mes, alrededor de un tercio de la infraestructura del núcleo.
Condiciones	La reserva la despliega el mismo pipeline con cada lanzamiento, recibe el 2% del tráfico real para demostrar que funciona, y cada trimestre se ensaya un failover completo. La decisión se registra en un ADR.

La pérdida esperada por una interrupción regional poco frecuente no justificaría por sí sola el warm standby; lo justifican el RTO contractual y el daño reputacional de una interrupción larga a la hora de la cena. La DR se compra como un seguro, y el negocio fija la prima que está dispuesto a pagar.

Clasificar por recorrido redujo mucho el costo: solo la parte del sistema que debe recuperarse rápido paga por una recuperación rápida.

ATENCIÓN

Trampas de la multirregión

- Una región de recuperación que nunca ha recibido tráfico: la deriva de configuración, los secretos que faltan y las cuotas pequeñas solo salen a la luz durante el desastre.
- Herramientas de failover, DNS, identidad o pipeline de despliegue alojados solo en la región que ha fallado.
- Suponer que la replicación protege frente a los borrados o la corrupción; los copia.
- Llamadas síncronas entre regiones en la ruta crítica del usuario, que añaden largos viajes de ida y vuelta a cada solicitud.
- Tratar el activo-activo como un ajuste de infraestructura en lugar de como un problema de diseño de datos.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué un diseño activo-activo sigue necesitando copias de seguridad de un momento dado?

La replicación copia los borrados, las malas migraciones y la corrupción a todas las regiones en segundos; solo las copias de seguridad permiten restaurar los datos tal como estaban antes del error.

▼ En un diseño con replicación asíncrona a una región de reserva, ¿qué determina cuántos datos se pierden en un fallo regional?

El retardo de replicación en el momento del fallo: las escrituras que aún no habían llegado a la réplica se pierden, así que el retardo es el RPO efectivo.

▼ **¿Frente a qué fallos protege una arquitectura basada en células que no cubre una segunda región con el mismo código?**

Los fallos causados por el propio sistema, como un mal despliegue, una entrada envenenada o un cliente sobrecargado, que afectarían a todas las regiones, pero solo a una célula.

▼ **¿Por qué pilot light es más arriesgado que warm standby durante un gran incidente regional?**

Pilot light tiene que lanzar y escalar el nivel de aplicación en el momento del fallo, cuando la capacidad y las operaciones del plano de control pueden escasear; warm standby ya tiene capacidad en funcionamiento.

▼ **Inténtalo: Ilustrativo: copia de seguridad y restauración supone una caída regional de 8 horas con £8000 perdidas por hora. Un warm standby (espera semiactiva) cuesta £6000 al mes y la reduce a 20 minutos. Se espera una caída regional cada cinco años. Compara los costos anuales y da tu consejo.**

Copia de seguridad: 8 x £8000 = £64 000 por interrupción, o £12 800 al año. Standby: £72 000 al año, más unos £533 de pérdida esperada. Solo por ingresos no se justifica. Comprueba el RTO y el RPO con el negocio: los contratos, la seguridad (safety) o la pérdida de clientes podrían cambiar la respuesta, y un pilot light más barato podría encajar. Mantén copias de seguridad probadas elijas lo que elijas, porque la replicación copia los borrados y la corrupción.

FUENTES

- AWS Whitepaper: Disaster Recovery of Workloads on AWS, 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
- AWS Whitepaper: Reducing the Scope of Impact with Cell-Based Architecture: https://docs.aws.amazon.com/wellarchitected/latest/reducing-scope-of-impact-with-cell-based-architecture/reducing-scope-of-impact-with-cell-based-architecture.html
- AWS: Summary of the Amazon DynamoDB service disruption in the Northern Virginia (US-EAST-1) Region: https://aws.amazon.com/message/101925/
- Amazon Builders' Library: Workload isolation using shuffle-sharding: https://aws.amazon.com/builders-library/workload-isolation-using-shuffle-sharding/
- Cloudflare Workers documentation: How Workers works: https://developers.cloudflare.com/workers/reference/how-workers-works/

SECCIÓN 10.7 DE 10.8 · INTERMEDIO · 24 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Ingeniería de plataformas y plataformas internas para desarrolladores

ANTES DE LEER ESTO

Esta sección se basa en:

- 3.7 Ley de Conway
- 10.4 Infraestructura como código
- 10.5 CI/CD y despliegue

Términos anteriores que utiliza: Kubernetes, Infraestructura como código (IaC), Política como código, DevOps, Observabilidad

LA IDEA EN BREVE

Imagina una gran oficina en la que cada departamento tiene que comprar sus propias impresoras, cablear su propia red y contratar a su propio personal de limpieza. El trabajo se hace, pero todos pierden tiempo en tareas rutinarias y algunos departamentos las hacen mal. Un buen equipo de servicios generales ofrece, en cambio, servicios listos para usar: reservar una sala de reuniones en línea, pedir una computadora portátil de una lista breve, solicitar un escritorio con unos pocos clics. Nadie está obligado a usarlos, pero son la forma más fácil. La ingeniería de plataformas es ese equipo de servicios generales para los equipos de software: hace que las tareas comunes y necesarias de construir y operar software sean rápidas, seguras y de autoservicio.

LO QUE NECESITAS SABER

- La ingeniería de plataformas construye una plataforma interna para desarrolladores: herramientas, servicios, conocimiento y soporte de autoservicio que permiten a los equipos de producto crear y operar software sin tickets, reduciendo el esfuerzo duplicado y la calidad desigual.
- Un camino dorado (golden path o paved road) es una forma opinada, respaldada, de extremo a extremo y opcional de realizar una tarea habitual, como crear un servicio nuevo y desplegarlo en producción.
- Backstage, creado en Spotify y publicado como código abierto en 2020, es un proyecto de la CNCF que ofrece un catálogo de software, plantillas, TechDocs y plugins; un portal es la puerta de entrada, no la plataforma en sí.

- La plataforma es un producto con clientes internos: necesita un responsable de producto, investigación de usuarios, documentación, sus propios SLO y una adopción que se gane en lugar de imponerse (el modo X como servicio y la plataforma viable más delgada de Team Topologies).
- El autoservicio se vuelve seguro gracias a los guardrails, comprobaciones automáticas de políticas con mensajes claros y una vía para las excepciones, en lugar de compuertas de aprobación humana.
- El éxito se mide por los resultados de los demás equipos: adopción, tiempo hasta el primer despliegue, métricas DORA y experiencia de los desarrolladores, usando varias dimensiones de SPACE en lugar de un único número.
- Los fallos típicos son construir una plataforma que nadie quiere, cambiar el nombre a un equipo de operaciones que trabaja con tickets, imponer la adopción y crear abstracciones que ocultan demasiado para poder depurar.

El problema que resuelven las plataformas

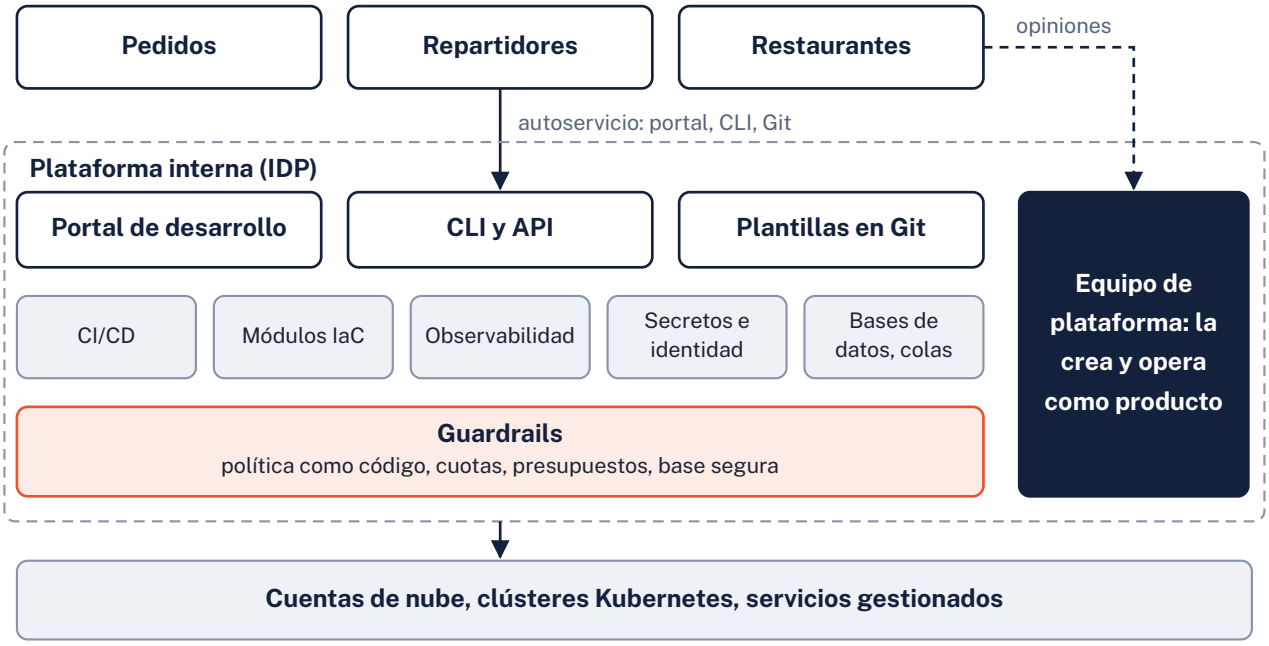
En 2006, el director de tecnología de Amazon, Werner Vogels, resumió el modelo operativo de su empresa como “tú lo construyes, tú lo operas” (you build it, you run it): el equipo que escribe un servicio también lo despliega, lo monitoriza y lo corrige. La idea pasó a ser central en DevOps y en los microservicios, y funciona: los equipos que operan su propio software sienten las consecuencias de sus decisiones de diseño y las corrigen. Pero el conjunto de herramientas nativas de la nube descrito en este capítulo es amplio. Un equipo de producto que hoy deba construir, operar y proteger un servicio puede necesitar entender imágenes de contenedor, objetos de Kubernetes, infraestructura como código, pipelines, observabilidad, gestión de secretos, políticas de red, análisis de la cadena de suministro y controles de costos, antes de escribir una sola línea de lógica de negocio.

Cuando cada equipo resuelve todo esto por su cuenta, ocurren tres cosas. El esfuerzo se duplica, porque diez equipos escriben diez pipelines ligeramente distintos. La calidad varía, porque algunos equipos cifran sus bases de datos y configuran copias de seguridad y otros lo olvidan. Y el tiempo que los equipos de producto dedican a la infraestructura es tiempo que no dedican al producto. La ingeniería de plataformas es la respuesta: un equipo dedicado construye una plataforma interna para desarrolladores (IDP) que empaqueta estas capacidades para que los equipos de producto puedan usarlas mediante autoservicio.

El artículo de Evan Bottcher de 2018 en martinfowler.com ofrece una descripción muy citada: una plataforma digital es una base de API, herramientas, servicios, conocimiento y soporte de autoservicio, organizada como un producto interno atractivo. Cada parte de esa descripción importa. *Autoservicio* significa que no hay tickets para las necesidades rutinarias. *Conocimiento y soporte* significa documentación y personas, no solo software. *Atractivo* significa que los

equipos eligen usarla porque es mejor que hacer el trabajo ellos mismos. *Producto interno* significa que se gestiona como un producto, con clientes, y no como un proyecto o un centro de costos.

Qué contiene una plataforma interna para desarrolladores



Los equipos de producto consumen la plataforma mediante interfaces de autoservicio; el equipo de plataforma la opera como un producto y aprende de sus opiniones.

Una IDP no es un único producto. Suele ser una integración de herramientas existentes (un sistema de CI, un controlador de GitOps, los servicios gestionados de un proveedor de nube, una pila de observabilidad, un almacén de secretos) detrás de interfaces coherentes y valores predeterminados razonables. El trabajo distintivo del equipo de plataforma es elegirlas, integrarlas, documentarlas y darles soporte, y diseñar la experiencia de usarlas, no escribir un nuevo orquestador.

Caminos dorados y caminos pavimentados

Lo más valioso que puede ofrecer una plataforma joven es un camino dorado (golden path): una forma opinada, respaldada y documentada de realizar de extremo a extremo una tarea habitual, como crear un servicio nuevo y llevarlo a producción. Spotify popularizó el término (su blog de ingeniería describió sus caminos dorados en 2020); Netflix habla del *camino pavimentado* (paved road). La metáfora es precisa. Un camino pavimentado es la ruta más fácil, y se mantiene, pero aún puedes salirte de él y circular campo a través si tienes una buena razón; simplemente no cuentas con el soporte del camino.

Un buen camino dorado tiene cinco propiedades. Es **de extremo a extremo**: lleva a un equipo desde un repositorio vacío hasta un servicio monitorizado en producción, no solo a través de un paso. Es **opinado**: toma las decisiones comunes (versiones del lenguaje, formato de logging, estrategia de despliegue, valores predeterminados de la base de datos) para que los equipos no tengan que tomarlas. Es **opcional**: los equipos con necesidades poco habituales pueden abandonarlo, lo que obliga al equipo de plataforma a ser honesto sobre si el camino es realmente mejor. Está **respaldado**: cuando falla, el equipo de plataforma lo arregla. Y está **en evolución**: los servicios creados a partir de la plantilla del año pasado reciben actualizaciones, en lugar de quedarse congelados en la versión con la que empezaron.

Qué suele proporcionar un camino dorado para un servicio nuevo

Elemento	Qué recibe el equipo de producto	Qué sustituye
Plantilla de servicio	Un repositorio con un esqueleto funcional, pruebas, Dockerfile, linting y un README, generado en minutos.	Copiar un servicio antiguo y borrar lo que no encaja.
Pipeline	Un pipeline de CI/CD listo, con análisis, firma y despliegue progresivo, como en la sección 10.5.	Que cada equipo escriba y mantenga su propio pipeline.
Configuración de ejecución	Ajustes de Kubernetes o de la plataforma gestionada con solicitudes de recursos, sondas y autoescalado razonables.	Prueba y error en producción.
Módulos de infraestructura	Módulos de infraestructura como código revisados para bases de datos, colas y buckets, con cifrado, copias de seguridad y etiquetas incorporados.	Tickets a un equipo de operaciones, o módulos escritos a mano de calidad variable.
Observabilidad	Logging estructurado, métricas, trazas y un panel inicial con alertas sobre los objetivos de nivel de servicio.	Monitorización añadida después del primer incidente.
Entrada en el catálogo y documentación	El servicio registrado con su responsable, sus dependencias y su documentación.	Hojas de cálculo y conocimiento informal sobre quién es responsable de qué.

Backstage y los portales para desarrolladores

Un portal para desarrolladores es la puerta de entrada de muchas plataformas. El más conocido es Backstage, creado dentro de Spotify, publicado como código abierto en marzo de 2020, aceptado por la Cloud Native Computing Foundation (CNCF) como proyecto sandbox en septiembre de 2020 y promovido a incubación en marzo de 2022. Sus funciones principales son un **catálogo de software**, en el que cada servicio, sitio web, biblioteca, API y recurso se

registra con su responsable y sus relaciones; las **plantillas de software**, que generan proyectos nuevos a partir de plantillas de caminos dorados; **TechDocs**, que publica la documentación escrita en Markdown junto al código; y un amplio ecosistema de plugins que muestran pipelines, el estado de Kubernetes, los costos y los incidentes junto a cada entrada del catálogo.

Cada componente se describe a sí mismo en un pequeño archivo guardado en su propio repositorio, de modo que la información sobre su responsable vive con el código y se revisa como código:

```
apiVersion: backstage.io/v1alpha1
kind: Component
metadata:
  name: restaurant-ratings
  description: Stores and serves customer ratings of restaurants
  tags: [python, http]
spec:
  type: service
  lifecycle: production
  owner: team-restaurants
  system: restaurant-discovery
  providesApis: [ratings-api]
  dependsOn: [resource:ratings-db]
```

Una entrada del catálogo de Backstage (catalog-info.yaml) para un servicio de reparto de comida: su responsable, su ciclo de vida, el sistema al que pertenece, la API que proporciona y un recurso del que depende.

Dos advertencias. Backstage es un framework que una organización ensambla y opera, y que normalmente requiere ingenieros que conozcan TypeScript y React; además, la calidad de sus plugins es variable. Existen servicios de Backstage gestionados y portales comerciales para los equipos que no quieren operarlo. Y, sobre todo, **un portal no es una plataforma**. Un portal precioso delante de procesos manuales basados en tickets solo muestra la cola de forma más atractiva. Empieza por las capacidades y el camino dorado; añade un portal cuando haya detrás lo suficiente como para que merezca la pena encontrarlo.

La plataforma como producto

La idea más importante de la ingeniería de plataformas es que la plataforma es un producto y los desarrolladores que la usan son sus clientes. Eso implica un gerente de producto o un líder de plataforma que sea dueño de la hoja de ruta; investigación de usuarios (entrevistas, observar a un equipo mientras crea un servicio, encuestas) antes de construir; documentación e incorporación tratadas como funcionalidades; objetivos de nivel de servicio publicados para la

propia plataforma; políticas de versionado y de retirada para que los equipos puedan planificar; y, sobre todo, una adopción que se gane en lugar de imponerse. Si los equipos evitan la plataforma, eso es una señal del mercado, no insubordinación.

Team Topologies (Matthew Skelton y Manuel Pais, 2019), tratado en la sección 11.5, aporta el modelo organizativo. Un equipo de plataforma existe para reducir la carga de los equipos alineados al flujo, los que están alineados con un flujo de cambio del negocio, como los pedidos o el despacho de repartidores, e interactúa con ellos principalmente en modo *X como servicio* (X-as-a-Service): una oferta clara y de autoservicio que requiere poca colaboración. El modo de colaboración se usa de forma deliberada y breve, por ejemplo al diseñar una capacidad nueva con uno o dos equipos pioneros. Skelton y Pais también acuñaron la plataforma viable más delgada (thinnest viable platform): la plataforma más pequeña que ayuda de verdad, que para una organización pequeña puede ser una página wiki bien escrita con los servicios aprobados y una plantilla de pipeline compartida.

Esto también significa que la plataforma debe hacer que la ley de Conway juegue a su favor: las interfaces de la plataforma se convierten en los límites entre el equipo de plataforma y cada equipo de producto, así que deben ser estables, documentadas y estrechas, como cualquier buena abstracción.

Autoservicio con guardrails

El autoservicio elimina la cola; los guardrails (salvaguardas) lo mantienen seguro. La diferencia con los controles tradicionales está en dónde se produce la comprobación. Una **compuerta** (gate) es una persona que debe aprobar una solicitud, como un comité asesor de cambios o un administrador de bases de datos que revisa un ticket. Un **guardrail** es un límite automático que deja pasar todo lo que cumple la política y bloquea o señala lo que no, en el momento de la solicitud. Los guardrails suelen escribirse como política como código (presentada en la sección 10.4) y se aplican en los pipelines, en los controladores de admisión y en las políticas de organización de la nube.

Capacidades de autoservicio y sus guardrails

Capacidad	Interfaz de autoservicio	Ejemplos de guardrails
Servicio nuevo	Plantilla en el portal o la CLI	Debe tener un responsable en el catálogo; debe usar la imagen base aprobada.
Base de datos	Módulo de infraestructura o formulario del portal	Cifrado en reposo activado; copias de seguridad automáticas activadas; sin acceso desde la red pública; tamaño limitado por la cuota del equipo.
Despliegue	Hacer merge a main activa el pipeline	Solo imágenes firmadas del registro interno; solicitudes de recursos y sondas obligatorias; análisis canario obligatorio para los servicios de nivel 1.
Gasto en la nube	Cuentas o namespaces de equipo creados a petición	Etiquetas de costo obligatorias; alertas de presupuesto al 80% y al 100%; los tipos de instancia caros requieren una justificación.
Acceso	Solicitudes de roles a través del portal	El acceso a producción tiene límite de tiempo y queda registrado; nada de claves personales de larga duración.

Los buenos guardrails comparten tres propiedades: dan un mensaje claro que explica qué se ha bloqueado y cómo corregirlo; se versionan y se prueban como código, con un periodo de avisos antes de que una regla nueva empiece a bloquear; y vienen con una vía documentada para las excepciones justificadas, porque un guardrail sin vía de excepción vuelve a convertirse en una compuerta la primera vez que se equivoca.

SI ESTO ES NUEVO PARA TI

Paso a paso: ¿una compuerta o un guardrail?

El punto en el que más gente tropieza aquí es la diferencia entre una compuerta (gate) y un guardrail. Ambos pueden hacer cumplir las mismas reglas, pero una compuerta pone a una persona delante de cada solicitud, mientras que un guardrail comprueba automáticamente en el momento de la solicitud. Sigue una solicitud de cuenta de las dos maneras.

- El equipo de restaurantes necesita una cuenta en la nube para su nuevo servicio de valoraciones de restaurantes. Las reglas dicen que las alertas de presupuesto deben estar activadas, que cada recurso debe llevar etiquetas de costo y que los tipos de máquina caros requieren una justificación por escrito.
- Con una compuerta, el equipo abre un ticket. Cuando llega al principio de la cola, quizá días después, un ingeniero de operaciones de nube lo comprueba según las reglas y crea la cuenta. Las reglas se cumplen, pero a costa de esperar.
- Con un guardrail, el equipo solicita la cuenta a través del portal de la plataforma. Unas comprobaciones automáticas, escritas como política como código, la comparan con las mismas reglas. La solicitud las cumple y la cuenta se crea sin ticket ni aprobación humana.
- Más tarde, el equipo intenta arrancar un servidor sin etiquetas de costo. La comprobación lo bloquea al instante, con un mensaje que dice qué se ha bloqueado, por qué y cómo corregirlo. El equipo añade las etiquetas y lo vuelve a intentar.
- Si el equipo tuviera un caso que las reglas no previeron, usaría la vía de excepción documentada. Sin ella, la primera regla errónea deja al equipo bloqueado, y el guardrail vuelve a convertirse en una compuerta.
- **La lección:** un guardrail no debilita las reglas; traslada la comprobación de la cola de una persona al momento de la solicitud. Funciona cuando se explica a sí mismo, se prueba como código y deja una vía para las excepciones justificadas.

Reducir la carga cognitiva, no ocultarlo todo

Team Topologies toma la idea de carga cognitiva de la investigación educativa de John Sweller y distingue tres tipos. La carga intrínseca es inherente a la tarea, como entender cómo funcionan las tarifas de reparto. La carga extrínseca procede del entorno, como recordar los doce pasos para solicitar una base de datos. La carga pertinente (germane) es el esfuerzo de aprendizaje que facilita el trabajo futuro, como entender el dominio con más profundidad. Una plataforma debe eliminar la carga extrínseca para que los equipos puedan dedicar su atención limitada a la carga intrínseca y la pertinente de su propio dominio.

Eso no significa ocultarlo todo. Un equipo que opera su servicio sigue necesitando entender cómo se comporta en producción: su uso de recursos, sus modos de fallo, su latencia y sus costos. Las abstracciones que ocultan demasiado tienen fugas en el peor momento, durante un incidente, cuando el equipo descubre que no puede ver los logs o no sabe en qué nivel de base de datos está. Las buenas plataformas son transparentes: proporcionan valores predeterminados razonables e interfaces sencillas, y a la vez permiten a los equipos mirar por debajo y sustituir esos valores cuando lo necesitan.

Medir si la plataforma funciona

El valor de una plataforma se refleja en los resultados de otros equipos, así que sus medidas deben mirar ahí. Las medidas útiles se agrupan en cuatro grupos, y un conjunto equilibrado toma al menos una de cada uno.

Medidas del éxito de la plataforma

Grupo	Ejemplos de medidas	Cuidado con
Adopción	Proporción de servicios en el camino dorado; adopción voluntaria de las capacidades nuevas; equipos que abandonan el camino y por qué.	Una adopción impuesta no dice nada sobre el valor.
Entrega	Métricas DORA de los equipos dentro y fuera de la plataforma; tiempo desde la creación de un servicio hasta su primer despliegue en producción; tiempo de incorporación de un ingeniero nuevo.	Comparaciones entre equipos que hacen trabajos muy distintos.
Experiencia de los desarrolladores	Encuestas periódicas de satisfacción, fricción percibida y tiempo perdido por culpa de las herramientas; entrevistas.	Fatiga de encuestas; trata los resultados como señales, no como puntuaciones.
Salud de la plataforma	Los SLO de la propia plataforma (disponibilidad del pipeline, tiempo de aprovisionamiento), la carga de soporte, el costo por equipo.	Una plataforma fiable pero que nadie usa sigue siendo un fracaso.

El marco SPACE, publicado en ACM Queue en 2021 por Nicole Forsgren, Margaret-Anne Storey y colegas de GitHub, Microsoft Research y la Universidad de Victoria, advierte que la productividad de los desarrolladores no puede capturarse con un único número. Nombra cinco dimensiones: satisfacción y bienestar, desempeño, actividad, comunicación y colaboración, y eficiencia y flujo. Recomienda medir varias dimensiones a la vez, incluida al menos una medida de percepción, como una encuesta, y no juzgar nunca a las personas por recuentos de actividad como los commits o las líneas de código.

Las pruebas sobre las plataformas son alentadoras, pero no incondicionales. El informe *Accelerate State of DevOps* de DORA de 2024 concluyó que los usuarios de plataformas internas para desarrolladores declaraban una productividad individual alrededor de un 8% mayor y un desempeño del equipo un 10% mayor, pero el rendimiento (throughput) de entrega de software y la estabilidad de los cambios eran un 8% y un 14% menores. El informe ofrece posibles explicaciones, como los traspasos adicionales y los pasos que añade la propia capa de plataforma, y la lección para los arquitectos es medir los resultados de entrega además de la satisfacción, y esperar un bajón mientras la plataforma madura.

EJEMPLO PRÁCTICO

Ejemplo práctico: el primer camino dorado para un servicio nuevo de reparto de comida

La empresa de reparto de comida tiene doce equipos de producto. Una encuesta y una semana de observación muestran que crear un servicio nuevo lleva unas tres semanas y catorce tickets a cuatro equipos distintos. Se forma un equipo de plataforma de cuatro personas. Los primeros equipos de producto que necesitan un servicio nuevo son el equipo de restaurantes, que quiere un servicio de valoraciones de restaurantes, y el equipo de repartidores.

Elegir el camino	Inventario de los servicios existentes: el 70% son servicios HTTP sin estado en Python o Go con una base de datos PostgreSQL y, a menudo, una suscripción a una cola. Esa forma se convierte en el camino dorado v1; todo lo demás queda, por ahora, fuera del camino.
Diseñar con los usuarios	Dos ingenieros de los equipos de restaurantes y de repartidores se unen al equipo de plataforma durante tres semanas (modo de colaboración) para construir el camino con ellos, no para ellos.
Contenido	Repositorio de plantilla; el pipeline estándar con despliegue canario; ajustes de Kubernetes con solicitudes de recursos, sondas y autoescalado; un módulo de base de datos con cifrado, copias de seguridad y etiquetas de costo; logging, trazado y un panel de SLO; una entrada en el catálogo; una guía de primeros pasos.
Guardrails	Etiquetas de responsable y de costo obligatorias; solo imágenes firmadas; base de datos sin acceso público. Todo se comprueba automáticamente; avisos durante un mes antes de bloquear.
Primer resultado	El equipo de restaurantes crea restaurant-ratings desde el portal y despliega una versión hello-world en producción en 5 horas, frente a las tres semanas de antes. Tickets necesarios: ninguno.
Medir	Tiempo hasta el primer despliegue en producción, encuesta a los dos equipos piloto y métricas DORA de los servicios del camino, revisados cada mes. Otros tres equipos adoptan el camino voluntariamente en un trimestre.
Próximos pasos	Los equipos piden soporte para consumidores de eventos y entornos de vista previa por cada pull request; la hoja de ruta se ordena según la demanda. Una solicitud de soporte para un lenguaje minoritario se rechaza por ahora y se documenta como fuera del camino.

La plataforma empezó con un único camino para el caso más común, construido con sus primeros usuarios, y se juzgó por los resultados de estos y no por el número de funcionalidades que entregó.

La mayor parte del trabajo fue integración, valores predeterminados y documentación de herramientas existentes, no software nuevo.

Riesgos y modos de fallo

Las iniciativas de plataforma fracasan de maneras reconocibles. La más común es **construir una plataforma que nadie quiere**: un equipo pasa un año construyendo una ambiciosa plataforma de uso general, la lanza y descubre que los equipos de producto ya tienen soluciones que funcionan y no tienen tiempo para migrar. Empezar con algo delgado, con usuarios reales y un único camino dorado, es el antídoto. La segunda es **el equipo de operaciones con otro nombre**: a un equipo de infraestructura se le cambia la etiqueta a equipo de plataforma, pero sigue trabajando con una cola de tickets, de modo que no cambia nada salvo los cargos. Si las solicitudes rutinarias siguen necesitando un ticket y una persona, todavía no es una plataforma.

Otros riesgos son imponer la adopción, lo que elimina la retroalimentación que mantiene útil una plataforma; el exceso de abstracción, en el que una capa de desarrollo propio oculta tan completamente las herramientas subyacentes que los equipos no pueden depurarla ni ampliarla y el equipo de plataforma se convierte en un cuello de botella para cada cambio; descuidar la larga cola de servicios creados a partir de plantillas antiguas; y que la plataforma se convierta en un punto único de fallo cuya caída detiene todos los despliegues. Por el contrario, una organización pequeña puede no necesitar ningún equipo de plataforma: unas pocas plantillas compartidas mantenidas por voluntarios pueden ser la plataforma viable más delgada durante años.

Lista de comprobación: poner en marcha una plataforma

- [ ] **Pruebas del problema**
  Mide cuánto tardan hoy las tareas habituales (servicio nuevo, base de datos nueva, primer despliegue) y cuántos tickets y equipos implican.
- [ ] **Un responsable de producto**
  Alguien es dueño de la hoja de ruta, habla con los usuarios con regularidad y puede decir que no.
- [ ] **Primero, un único camino dorado**
  Elige la forma de servicio más común y hazla excelente de extremo a extremo antes de ampliar el alcance.
- [ ] **Autoservicio, no tickets**
  Toda solicitud rutinaria dentro del camino se completa sin aprobación humana; las excepciones tienen una vía documentada.
- [ ] **Guardrails como código**
  Reglas de seguridad, costo y fiabilidad aplicadas automáticamente, con mensajes claros y un periodo de avisos.
- [ ] **Abstracciones transparentes**
  Los equipos pueden ver los logs, las métricas, los costos y los recursos subyacentes, y sustituir los valores predeterminados cuando esté justificado.
- [ ] **Medidas de resultados**
  Adopción, tiempo hasta el primer despliegue, métricas DORA y encuestas a los desarrolladores, revisadas con regularidad y publicadas.
- [ ] **Vía de actualización**
  Los servicios creados a partir de plantillas antiguas reciben las mejoras, y las retiradas se anuncian con antelación y con ayuda para migrar.

COMPRUEBA LO QUE HAS ENTENDIDO

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

Una compuerta necesita que una persona apruebe cada solicitud; un guardrail es una comprobación automática que deja pasar de inmediato las solicitudes conformes y bloquea o señala el resto.

▼ **¿Por qué debe ser opcional un camino dorado?**

Para que los equipos con necesidades poco habituales no queden bloqueados, y para que el equipo de plataforma aprenda, a partir de quién abandona el camino, si este es realmente mejor que las alternativas.

▼ **Una empresa instala Backstage, pero crear una base de datos sigue requiriendo un ticket y una semana de espera. ¿Qué ha construido?**

Un portal delante de un proceso manual, no una plataforma; lo que crea valor son las capacidades de autoservicio que hay detrás del portal.

▼ **¿Por qué el marco SPACE desaconseja medir la productividad de los desarrolladores solo por los commits o las líneas de código?**

Los recuentos de actividad capturan una sola dimensión, son fáciles de manipular y no dicen nada sobre la satisfacción, la colaboración, el flujo ni el valor del resultado.

▼ **Inténtalo: La dirección renombra al equipo de operaciones como "Plataforma", obliga a todos los equipos a usarlo y mide el éxito por el número de tickets cerrados. Solicitar un servicio nuevo sigue requiriendo catorce tickets. Critica esto y di qué cambiarías.**

Este es un equipo guiado por tickets con otro nombre, no una plataforma. Trátalo como un producto: dale un responsable de producto, investiga lo que necesitan los equipos de producto y construye un camino dorado fino para la tarea más común, un nuevo servicio sin estado, como autoservicio con guardrails automáticos. Hazlo opcional y gánate la adopción. Mide los resultados de los demás, como el tiempo hasta el primer despliegue, la adopción, las métricas DORA y la experiencia de los desarrolladores. El objetivo es tener menos tickets; contar los cierres premia lo equivocado.

FUENTES

- Evan Bottcher: What I Talk About When I Talk About Platforms, martinfowler.com: https://martinfowler.com/articles/talk-about-platforms.html
- CNCF TAG App Delivery: Platforms White Paper: https://tag-app-delivery.cncf.io/whitepapers/platforms/
- Backstage documentation: What is Backstage?: https://backstage.io/docs/overview/what-is-backstage
- Forsgren et al.: The SPACE of Developer Productivity, ACM Queue (2021): https://queue.acm.org/detail.cfm?id=3454124

SECCIÓN 10.8 DE 10.8 · INTERMEDIO · 23 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

FinOps, arquitectura de costos y dependencia del proveedor (lock-in)

ANTES DE LEER ESTO

Esta sección se basa en:

- 2.7 Seguridad, usabilidad y más
- 5.6 Hexagonal y limpia
- 10.1 Modelos de nube

Términos anteriores que utiliza: Servicio gestionado, Multinube (multi-cloud), Solicitudes y límites de recursos (requests y limits), Puertos y adaptadores (arquitectura hexagonal), Serverless

LA IDEA EN BREVE

Alquilar un vehículo por un día es barato; alquilar uno todos los días durante tres años es mucho más caro que comprarlo. Dejar el motor en marcha en el estacionamiento gasta combustible para nada. Y si contratas en leasing un vehículo con un conector de carga especial, cambiar de marca más adelante también implica cargadores nuevos. La computación en la nube plantea las tres preguntas a la vez. FinOps es el hábito del presupuesto doméstico aplicado a la nube: ver cuánto cuesta cada parte, recortar el despilfarro, comprar con criterio. El lock-in es la pregunta del conector especial: cierta dependencia merece la pena, siempre que la hayas elegido a propósito y sepas cómo saldrías.

LO QUE NECESITAS SABER

- En la nube cada decisión arquitectónica es también una decisión de compra, por lo que el costo es un atributo de calidad que cambia con cada despliegue y se expresa mejor por unidad de valor de negocio, como el costo por pedido.
- FinOps, definido por la FinOps Foundation (un proyecto de la Linux Foundation), reparte la responsabilidad financiera entre ingeniería, finanzas y negocio mediante tres fases iterativas: informar, optimizar y operar.
- Los factores de costo habituales son la salida de datos (egress) y la transferencia entre zonas, los recursos ociosos y sobredimensionados, el volumen de logging, los sobreprecios de los servicios gestionados y el crecimiento descontrolado del almacenamiento; la mayoría son consecuencia de la arquitectura.

- Los descuentos por compromiso de uso solo convienen para la base que se necesitará con certeza, la capacidad spot conviene para el trabajo interrumpible y reintentable, y el autoescalado bajo demanda cubre los picos; ajusta el tamaño antes de comprometerte.
- El etiquetado permite la asignación de costos; el showback informa de los costos a los equipos, y el chargeback los carga a sus presupuestos y necesita una asignación precisa y reglas acordadas para los costos compartidos.
- El lock-in es un compromiso: Hohpe (2019) distingue muchos tipos y compara el costo de cambio multiplicado por su probabilidad con el costo de evitarlo; acéptalo cuando aporte un valor único.
- La portabilidad tiene costos reales; mantén la lógica de negocio detrás de puertos en los puntos de unión probables, conserva los datos exportables y mantén una estrategia de salida probada, algo que la regulación financiera de la UE ya exige para los servicios críticos.

El costo es una cualidad arquitectónica

La sección 2.7 (seguridad, usabilidad y más) incluyó el costo entre los atributos de calidad y recomendó expresarlo por unidad de trabajo. En un centro de datos, el costo era mayormente fijo: el hardware se compraba cada pocos años y las decisiones de un arquitecto apenas movían la factura mensual. En la nube, cada decisión arquitectónica es también una decisión de compra, que los ingenieros toman de forma continua en lugar de que la tome compras una vez al año. Un límite de servicio con demasiadas llamadas, un nivel de log de depuración, un entorno de pruebas olvidado o una llamada síncrona entre regiones aparecen en la factura del mes siguiente. El costo se ha convertido en un atributo de calidad que cambia con cada despliegue.

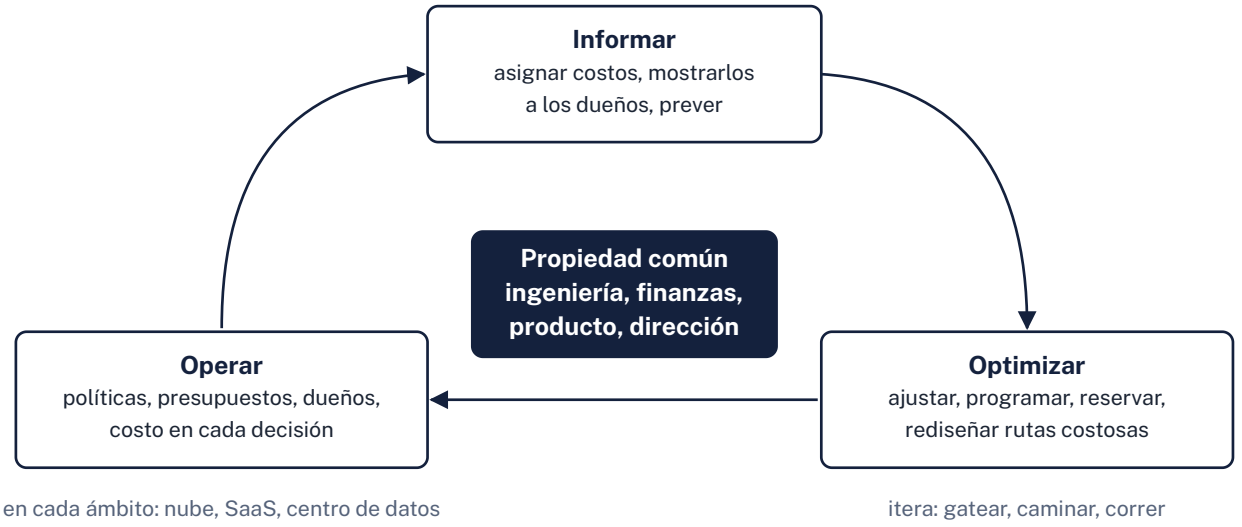
Dos disciplinas ayudan. FinOps es la práctica operativa que hace que el gasto en la nube sea visible, tenga dueño y se gestione. La arquitectura consciente del costo es la práctica de diseño que hace que los sistemas sean baratos de operar por unidad de valor desde el principio. Esta sección trata ambas y luego pasa a su incómodo vecino: el lock-in (la dependencia del proveedor), el costo de marcharse.

FinOps: fases, principios y quién hace qué

La FinOps Foundation, un proyecto de la Linux Foundation, define FinOps como un marco operativo y una práctica cultural que maximiza el valor de negocio de la tecnología mediante decisiones oportunas y basadas en datos, y una responsabilidad financiera compartida entre ingeniería, finanzas y negocio. El nombre recuerda deliberadamente a DevOps: así como DevOps derribó el muro entre construir y operar software, FinOps derriba el muro entre gastar dinero y decidir cómo se construye el software.

El FinOps Framework organiza el trabajo en tres fases iterativas. **Informar**: hacer visibles los costos, asignarlos a los equipos y productos que los generan, hacer previsiones y compararse con referencias. **Optimizar**: reducir el despilfarro y mejorar las tarifas, mediante el ajuste de tamaño, la programación de horarios, los compromisos de uso y los cambios de diseño. **Operar**: crear los hábitos que mantienen los costos bajo control, como los presupuestos, las políticas, la asignación de responsables y el costo considerado en cada decisión de diseño. Las organizaciones recorren el ciclo una y otra vez, madurando en cada capacidad de gatear a caminar y a correr. Entre los principios del marco están que los equipos colaboran, que el valor de negocio guía las decisiones tecnológicas, que todos se responsabilizan de su uso de la tecnología, que los datos de costos son accesibles, oportunos y precisos, que FinOps se habilita de forma centralizada y que el modelo de costo variable de la nube debe aprovecharse como ventaja. Las versiones recientes amplían el marco más allá de la nube pública a otros gastos tecnológicos, como las licencias SaaS, los centros de datos y los servicios de IA.

El ciclo de FinOps



FinOps es un ciclo que se repite, no un proyecto puntual de recorte de costos, y se basa en la propiedad compartida y no en un equipo central que vigila el gasto.

Economía unitaria: el costo por pedido

Una factura de la nube que sube de 80 000 a 100 000 al mes puede ser un desastre o un triunfo, según si el negocio creció un 5% o un 50%. La economía unitaria expresa el costo por unidad de valor de negocio: el costo por pedido para la plataforma de reparto de comida, por cliente activo para un servicio de streaming, por cada mil llamadas a la API para una plataforma para desarrolladores. Separa la eficiencia del crecimiento, conecta las decisiones de ingeniería con los resultados de negocio y da a los equipos un objetivo sobre el que pueden influir.

Calcularla bien requiere tres cosas. Primero, una **unidad que importe al negocio**, elegida con finanzas y producto, no inventada por los ingenieros. Segundo, la **asignación de costos**: los costos directos (el cómputo y la base de datos del servicio de pedidos) se asignan mediante etiquetas o namespaces, y los costos compartidos (el plano de control de Kubernetes, la pila de observabilidad, las herramientas del equipo de plataforma) se reparten según una regla acordada, como el uso proporcional. Tercero, distinguir entre **costo medio y costo marginal**: el costo medio por pedido incluye costos fijos que no crecen con los pedidos, así que el costo de los próximos mil pedidos suele ser menor de lo que sugiere la media, lo que importa al poner precio a una promoción o al prever el crecimiento.

SI ESTO ES NUEVO PARA TI

Paso a paso: cuánto cuestan los próximos mil pedidos

La idea más difícil de esta sección es la diferencia entre el costo medio y el costo marginal. La media reparte toda la factura entre todos los pedidos; el costo marginal es lo que añade un pedido más. Como parte de la factura es fija, ambos pueden diferir mucho. Sigue una promoción a través de los dos, con cifras ilustrativas.

- La factura mensual de la nube de la plataforma de reparto de comida es de £96 000 para 1.2 millones de pedidos, así que su costo por pedido, la medida de economía unitaria que recomienda esta sección, es de £0.080. Esa es la media.
- Una parte de la factura no se mueve con los pedidos: la base de datos de pedidos, dimensionada de antemano, los servidores siempre encendidos para las horas tranquilas, el plano de control de Kubernetes y las herramientas del equipo de plataforma. Supongamos que £36 000 son fijos de esta manera y £60 000 crecen con los pedidos: sobre todo cómputo autoescalado, transferencia de datos y logs.
- Así que cada pedido adicional añade unos £60 000 divididos entre 1.2 millones, es decir, £0.050: el costo marginal, muy por debajo de la media.
- El negocio planea una promoción que se espera que aporte 200 000 pedidos adicionales al mes. Con el precio medio, atenderlos parece costar 200 000 x £0.080 = £16 000. Con el costo marginal, son unos £10 000, siempre que los pedidos adicionales quepan en la capacidad fija ya pagada.
- El costo marginal no siempre es menor. Si la promoción llevara la base de datos de pedidos más allá de su tamaño actual, pasar a un tamaño mayor añadiría un escalón fijo, y los primeros pedidos por encima de ese umbral podrían costar mucho más que la media.
- **La lección:** sigue la eficiencia con el costo medio por pedido, pero pon precio al volumen adicional, como una promoción o una previsión de crecimiento, con el costo marginal. Pregunta primero qué costos se mueven con los pedidos y cuáles no.

Adónde va el dinero: los factores de costo habituales

La mayoría de las facturas de la nube que sorprenden proceden de unos pocos factores, y casi todos dependen de la arquitectura más que de negociar mejores precios.

Factores de costo habituales y respuestas arquitectónicas

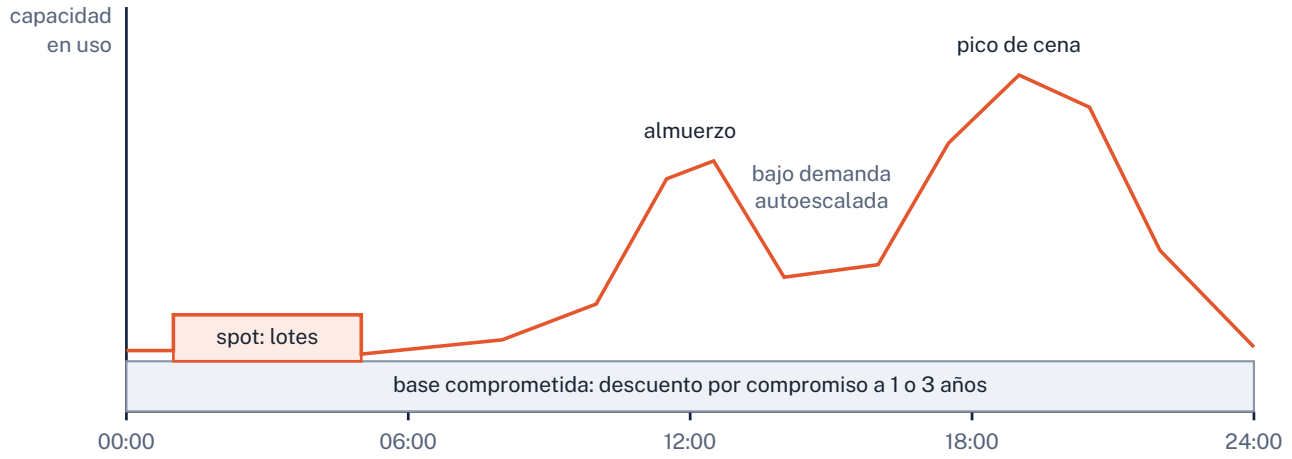
Factor	Por qué crece	Respuestas arquitectónicas
Salida de datos (egress) y transferencia	Los proveedores cobran por gigabyte los datos que salen a internet y, a menudo, el tráfico entre zonas y regiones y a través de las pasarelas NAT gestionadas. Los servicios con demasiadas llamadas repartidos entre zonas pagan en cada llamada.	Servir el contenido público a través de una CDN; mantener en la misma zona los servicios con muchas llamadas entre sí (enrutamiento consciente de la topología); usar endpoints privados hacia los servicios del proveedor; comprimir; evitar el tráfico síncrono entre regiones.
Recursos ociosos y sobredimensionados	Entornos de prueba encendidos 168 horas a la semana para 50 horas de uso; solicitudes de recursos de los contenedores fijadas muy por encima del uso real; volúmenes, snapshots y balanceadores de carga huérfanos.	Ajuste de tamaño (rightsizing) a partir del uso medido; horarios y autoescalado a cero para los entornos que no son de producción; entornos de vista previa con límite de tiempo; limpieza automática de los recursos no asociados.
Volumen de logging y de observabilidad	La ingesta y la retención se cobran por gigabyte; los logs de depuración en producción, el registro detallado de solicitudes y las etiquetas de métricas de alta cardinalidad crecen con el tráfico.	Niveles de log y muestreo de las solicitudes exitosas; retención por niveles (en caliente durante días, en archivo durante meses); métricas en lugar de logs para los recuentos; muestreo de trazas.
Sobreprecios de los servicios gestionados	Un servicio gestionado cuesta más por unidad de capacidad que operar el software uno mismo; el precio por solicitud crece linealmente con el volumen.	Comparar el costo total de propiedad, incluidas las personas y el riesgo; revisar los servicios con precio por solicitud cuando el volumen es alto y estable (consulta serverless); usarlos donde el sobreprecio compre fiabilidad o velocidad reales.
Crecimiento del almacenamiento	Los datos que nunca se borran, las versiones de objetos, las copias duplicadas y las copias de seguridad antiguas se acumulan sin hacer ruido.	Políticas de ciclo de vida hacia niveles más baratos y borrado; reglas de conservación acordadas con el negocio y conformes a la ley; copias de seguridad incrementales y deduplicadas.

Pagar menos por la misma capacidad

Una vez eliminado el despilfarro, se puede reducir la tarifa que se paga por la capacidad restante. La capacidad bajo demanda es la más flexible y la más cara. Un descuento por compromiso de uso cambia flexibilidad por precio: las Reserved Instances y los Savings Plans de AWS, las reservas y los planes de ahorro de Azure, y los committed use discounts de Google Cloud ofrecen reducciones sustanciales (AWS cita hasta un 72% para algunos compromisos) a cambio de comprometerse con un nivel de uso o de gasto durante uno o tres años. Los compromisos solo deben cubrir la base que con certeza se necesitará durante el plazo; comprometerse con la capacidad de pico convierte un descuento en despilfarro.

La capacidad spot (AWS Spot Instances, Azure Spot Virtual Machines, Google Cloud Spot VMs) vende la capacidad sobrante con grandes descuentos, a menudo del 60% al 90%, pero el proveedor puede recuperarla con poca antelación: dos minutos en AWS y unos 30 segundos en Azure y Google Cloud. Conviene para el trabajo sin estado, interrumpible y reintentable, como el procesamiento por lotes, los runners de CI, el renderizado y algunas capas web sin estado repartidas entre muchos tipos de instancia. No conviene para las bases de datos ni para nada que no pueda detenerse a mitad de una tarea. La arquitectura decide qué parte de una carga de trabajo puede usar spot: los trabajos idempotentes y con puntos de control pueden; los largos y con estado, no.

Adaptar los modelos de precios a la forma de la demanda



Comprométete con el mínimo, autoescala bajo demanda para los picos y lleva el trabajo interrumpible a la capacidad spot en las horas tranquilas.

Etiquetado, showback y chargeback

Nadie optimiza un costo que no puede ver. La asignación empieza con una política de etiquetado: cada recurso lleva un pequeño conjunto de etiquetas obligatorias, como el equipo responsable, el servicio, el entorno y el centro de costos, que se hacen cumplir con guardrails

en el código de infraestructura y en las políticas de organización de la nube, porque las etiquetas añadidas a mano pronto faltan. En Kubernetes, los costos se asignan por namespace y por etiqueta con herramientas como el proyecto de código abierto OpenCost o las funciones de asignación de costos de los propios proveedores. La especificación FOCUS de la FinOps Foundation define un formato común para los datos de facturación, de modo que los costos de varios proveedores puedan combinarse.

Con los costos asignados, una organización elige cómo usarlos. El showback informa a cada equipo de sus costos sin tocar su presupuesto; genera conciencia y es el punto de partida habitual. El chargeback (imputación de costos) carga los costos al presupuesto de cada equipo, lo que crea incentivos más fuertes, pero requiere una asignación precisa, reglas acordadas para los costos compartidos y una forma de resolver las disputas. En cualquier caso, publica la parte del gasto que no se puede asignar: una parte sin etiquetar de más de un pequeño porcentaje significa que las cifras todavía no son lo bastante fiables como para actuar en consecuencia.

Decisiones arquitectónicas que determinan el costo

Los mayores ahorros provienen del diseño, no de los descuentos, y son más baratos de obtener antes de construir un sistema. Cada una de las decisiones siguientes tiene un perfil de costo predecible. **Granularidad de los servicios**: cada salto de red cuesta cómputo, serialización y posiblemente transferencia de datos, así que los servicios de grano fino repartidos en muchas zonas cuestan más por solicitud que un monolito modular. **Fan-out síncrono**: una solicitud que llama a diez servicios mantiene ocupados once conjuntos de recursos mientras dura. **Número de entornos**: las copias completas de producción para cada equipo y cada rama multiplican la factura; los entornos compartidos con aislamiento o las vistas previas de corta duración cuestan mucho menos. **Retención y replicación de datos**: cada copia adicional y cada región adicional multiplican el almacenamiento y la transferencia. **Caché y CDN**: un acierto de caché suele ser mucho más barato que volver a calcular, y un acierto de CDN evita la salida de datos desde el origen. **Modelo de cómputo**: el precio serverless por solicitud es más barato que la capacidad siempre encendida por debajo de un volumen de cruce y más caro por encima de él.

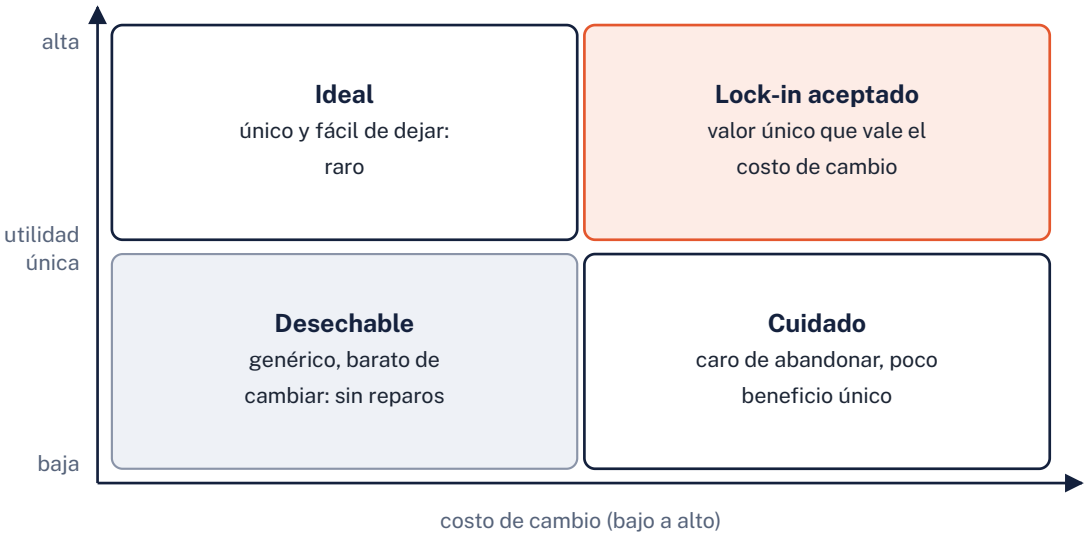
Registra las consecuencias de costo significativas en los registros de decisiones junto con los demás compromisos de calidad, y plantéate añadir el costo a las funciones de aptitud que protegen la arquitectura: por ejemplo, una comprobación mensual de que el costo por pedido no ha subido más de un 10% sin una explicación.

El lock-in es un compromiso, no un pecado

El lock-in es el costo y la dificultad de abandonar algo una vez que dependes de ello. Suele tratarse como algo que hay que evitar a toda costa; el artículo de Gregor Hohpe de 2019 en martinfowler.com, *Don't get locked up into avoiding lock-in*, sostiene que eso es en sí mismo una trampa. Distingue varios tipos de lock-in: de proveedor (contratos y condiciones comerciales), de producto (las API y funcionalidades de un producto concreto, aunque sea de código abierto), de versión, de arquitectura (un estilo como serverless), de plataforma (cuentas, identidad y modelos de seguridad), de habilidades (lo que sabe tu gente), legal (cumplimiento normativo y licencias) y mental (suposiciones que te impiden considerar alternativas). Usar software de código abierto reduce el lock-in de proveedor, pero deja la mayoría de los demás.

El paso clave es tratar el lock-in como un problema de valoración de opciones. El costo de cambio es lo que costaría trasladarse; multiplicado por la probabilidad de llegar a hacerlo, da el costo esperado de estar atado. Reducir el costo de cambio tiene su propio precio: abstracción adicional, diseños de mínimo común denominador y funcionalidades a las que se renuncia. Un cambio que costaría 50 000 y tiene un 5% de probabilidades de producirse tiene un costo esperado de 2500, así que gastar mucho más que eso para evitarlo es un mal negocio. La matriz de Hohpe de utilidad única frente a costo de cambio convierte esto en una clasificación rápida.

Utilidad única frente a costo de cambio



Según Gregor Hohpe (2019): el lock-in es aceptable cuando aporta un valor único; desconfía de las dependencias costosas que dan poco a cambio.

El costo de la portabilidad y la estrategia de salida

La portabilidad no es gratuita. Diseñar cada componente para que funcione en varias nubes significa usar solo las funcionalidades que todas comparten, construir y probar capas de abstracción que tienen fugas (consulta ocultación de información y abstracciones con fugas), mantener habilidades en varias plataformas y renunciar a servicios gestionados que habrían ahorrado esfuerzo. Muchas estrategias multinube (multi-cloud) acaban con los costos de todos sus proveedores y las ventajas de ninguno. La portabilidad es más barata donde ya existen estándares abiertos: las imágenes de contenedor OCI, los manifiestos de Kubernetes, las bases de datos compatibles con PostgreSQL, las API de almacenamiento de objetos compatibles con S3, los protocolos estándar como HTTP, AMQP y el protocolo de Kafka, y OpenTelemetry para la observabilidad. Incluso las herramientas que se usan para seguir siendo portables, como Terraform u OpenTofu, son dependencias por derecho propio.

Los profesionales no se ponen de acuerdo sobre hasta dónde llegar. Los defensores de la portabilidad señalan las subidas de precios, las caídas del servicio, las sanciones, la regulación y los casos en que salir de la nube resultó rentable (consulta la repatriación desde la nube en la sección 10.1). Los partidarios de comprometerse con un solo proveedor señalan la velocidad y la fiabilidad que se obtienen al aprovechar a fondo los servicios gestionados. Una posición intermedia sensata es aceptar el lock-in deliberadamente donde aporte un valor claro, mantener la lógica de negocio independiente de las API del proveedor detrás de puertos y adaptadores en los pocos puntos de unión con más probabilidades de cambiar, guardar los datos en formatos que se puedan exportar y dejar por escrito una estrategia de salida.

La regulación exige cada vez más ese último paso. El Reglamento de Resiliencia Operativa Digital de la UE (Digital Operational Resilience Act, Reglamento (UE) 2022/2554, aplicable desde el 17 de enero de 2025; no debe confundirse con las métricas de entrega DORA) exige a las entidades financieras disponer de estrategias de salida para los servicios TIC que respaldan funciones esenciales o importantes. Las normas sobre el cambio de proveedor de nube del Reglamento de Datos de la UE (Data Act) se aplican desde el 12 de septiembre de 2025: los clientes pueden cambiar de proveedor con un preaviso de dos meses como máximo, y los proveedores solo pueden cobrar sus costos directos por el cambio, incluida la salida de datos, hasta el 12 de enero de 2027, y nada a partir de entonces. En 2024 los mayores proveedores también anunciaron la transferencia de datos gratuita para los clientes que los abandonan, sujeta a condiciones. Una estrategia de salida debe nombrar el sustituto de cada servicio crítico, estimar el tiempo y el costo del traslado, demostrar que los datos pueden exportarse en un formato utilizable y ensayarse al menos en parte, por ejemplo restaurando una copia de seguridad en otro entorno.

EJEMPLO PRÁCTICO

Ejemplo práctico: el costo por pedido y tres cambios arquitectónicos

La plataforma de reparto de comida gestiona 1.2 millones de pedidos al mes. Su factura mensual de la nube es de 96 000: cómputo 42 000, bases de datos gestionadas 18 000, observabilidad 14 000, transferencia de datos 12 000, otros servicios gestionados 6000 y almacenamiento 4000. Todas las cifras son ilustrativas, en libras.

Punto de partida	96 000 / 1 200 000 pedidos = 0.080 por pedido. El showback revela que las solicitudes de recursos de cómputo duplican aproximadamente el uso medido, que la mitad del volumen de logs es salida de depuración y que los servicios de pedidos y de precios intercambian unas 40 llamadas por pedido entre zonas de disponibilidad.
Cambio 1: ajustar el tamaño y luego comprometerse	Las solicitudes fijadas a partir del uso p95 medido reducen el número de nodos en un 25%: el cómputo pasa de 42 000 a 31 500. El 60% estable de esa cifra (18 900) se cubre con un compromiso de un año con un descuento de alrededor del 30%, lo que ahorra 5670. El cómputo queda en 25 830, un ahorro de 16 170.
Cambio 2: registrar menos y durante menos tiempo	Logging de depuración desactivado en producción, solicitudes exitosas muestreadas al 10% y retención en caliente reducida de 90 a 14 días, con un nivel de archivo. La ingesta cae alrededor de un 60%: la observabilidad pasa de 14 000 a 6500, un ahorro de 7500.
Cambio 3: dejar de pagar por la distancia	La lógica de precios se traslada al proceso del servicio de pedidos para la ruta crítica (lo que elimina la mayoría de las 40 llamadas), con enrutamiento consciente de la zona para las llamadas restantes e imágenes de restaurantes servidas desde una CDN en lugar del origen. La transferencia de datos pasa de 12 000 a 6000; la CDN añade 1000. Ahorro neto: 5000.
Resultado	96 000 menos 28 670 = 67 330 al mes, o 0.056 por pedido: un 30% menos por pedido, sin cambios en las funcionalidades ni en los objetivos de fiabilidad.
Lo que no cambió	Los costos de bases de datos y de almacenamiento no se tocaron este trimestre porque los ahorros posibles eran pequeños en relación con el riesgo; siguen en el backlog.

El mayor cambio estructural fue arquitectónico (eliminar un límite con demasiadas llamadas e introducir una CDN), el cambio de logging fue una decisión de diseño sobre la observabilidad, y el descuento de tarifa se aplicó solo después del ajuste de tamaño, de modo que la empresa no se comprometió con el despilfarro.

Seguir el costo por pedido en lugar de la factura total muestra si el crecimiento futuro mantiene la mejora: si los pedidos se duplican y el costo por pedido se mantiene cerca de 0.056, la eficiencia se ha sostenido.

Preguntas frecuentes

¿Debe cobrarse a cada equipo por sus costos de nube?
Empieza con showback, para que los equipos vean sus costos y puedan actuar, y pasa a chargeback solo cuando la asignación sea precisa y los costos compartidos se repartan según reglas que todos acepten; cobrar cifras imprecisas genera discusiones, no ahorros.

¿Es la multinube la mejor protección contra el lock-in?
Rara vez por sí sola. Ejecutar el mismo sistema en varias nubes es caro y te limita a las funcionalidades comunes. Usar proveedores distintos para cargas de trabajo distintas e independientes, junto con una estrategia de salida probada para las críticas, suele dar la mayor parte de la protección por una fracción del costo.

¿Cuándo es prematura la optimización de costos?
Cuando el producto no ha encontrado su mercado y el tiempo de ingeniería es el recurso más escaso. Aun así, etiquetar desde el primer día y tener un costo aproximado por unidad es barato y facilita mucho la optimización posterior.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Por qué el costo por pedido es más útil que la factura mensual total?**

Separa la eficiencia del crecimiento: una factura creciente con un costo por pedido decreciente es sana, mientras que una factura estable con pedidos decrecientes no lo es.

▼ **¿Por qué el ajuste de tamaño debe ir antes de comprar descuentos por compromiso de uso?**

Un compromiso fija un nivel de gasto durante uno o tres años; comprometerse antes de eliminar el sobredimensionamiento convierte el despilfarro en una obligación a largo plazo.

▼ **¿Qué tipo de carga de trabajo conviene para la capacidad spot y por qué?**

El trabajo sin estado, interrumpible y reintentable, como los trabajos por lotes y los runners de CI, porque el proveedor puede recuperar la capacidad spot con segundos o minutos de antelación.

▼ **Según Hohpe, ¿cuándo es aceptable el lock-in?**

Cuando aporta una utilidad única que vale el costo de cambio, o cuando el cambio es tan improbable que el costo esperado es menor que el costo de evitar el lock-in.

▼ **Inténtalo: La factura es de £60 000 al mes para 1 500 000 pedidos: cómputo £30 000, logging £9000. El ajuste de tamaño (rightsizing) reduce el cómputo un 30%; eliminar los logs de depuración reduce el logging a la mitad. Halla el costo por pedido antes y después. ¿Qué debería ir antes que cualquier descuento por compromiso de uso, y por qué?**

Antes: £60 000 / 1 500 000 = 4.0 peniques. Los ahorros son £9000 y £4500, de modo que la factura baja a £46 500, o 3.1 peniques. Ajusta primero el tamaño, porque un compromiso sobre capacidad sobredimensionada fija el desperdicio durante todo su plazo. Después, comprométete solo con la base medida. Informa a los equipos del costo por pedido mediante showback, porque un total en aumento puede significar simplemente más negocio, mientras que un costo unitario en aumento indica un problema real.

FUENTES

- FinOps Foundation: FinOps Framework: https://www.finops.org/framework/
- Gregor Hohpe: Don't get locked up into avoiding lock-in, martinfowler.com: https://martinfowler.com/articles/oss-lockin.html
- AWS Well-Architected Framework: Cost Optimization Pillar: https://docs.aws.amazon.com/wellarchitected/latest/cost-optimization-pillar/welcome.html
- Regulation (EU) 2023/2854 (Data Act), EUR-Lex: https://eur-lex.europa.eu/eli/reg/2023/2854/oj

Lecturas recomendadas

- *The NIST Definition of Cloud Computing (SP 800-145)*, Peter Mell and Timothy Grance (2011), National Institute of Standards and Technology. La definición breve y muy citada de las características esenciales, los modelos de servicio y los modelos de despliegue.
  https://csrc.nist.gov/pubs/sp/800/145/final
- *The Twelve-Factor App*, Adam Wiggins (2011), 12factor.net (Heroku; community-maintained since 2024). Doce prácticas concisas para aplicaciones desplegables y escalables; sigue siendo el vocabulario común del diseño de aplicaciones nativas de la nube.
  https://12factor.net/
- *Kubernetes: Up and Running (3rd edition)*, Brendan Burns, Joe Beda, Kelsey Hightower and Lachlan Evenson (2022), O'Reilly Media. Una introducción clara a los objetos y las operaciones de Kubernetes, escrita por personas que ayudaron a crearlo.
- *Infrastructure as Code (3rd edition)*, Kief Morris (2025), O'Reilly Media. El libro de referencia sobre cómo diseñar, probar y entregar código de infraestructura, independientemente de cualquier herramienta concreta.
- *Continuous Delivery*, Jez Humble and David Farley (2010), Addison-Wesley. El texto fundacional sobre los pipelines de despliegue, el principio de compilar una vez y desplegar muchas, y cómo convertir los lanzamientos en rutina.
- *Accelerate: The Science of Lean Software and DevOps*, Nicole Forsgren, Jez Humble and Gene Kim (2018), IT Revolution. La investigación que hay detrás de las métricas DORA, que vincula las prácticas de entrega y una arquitectura poco acoplada con el rendimiento de la organización.
- *Site Reliability Engineering: How Google Runs Production Systems*, Betsy Beyer, Chris Jones, Jennifer Petoff and Niall Richard Murphy (eds) (2016), O'Reilly Media (free online). Cómo opera sus sistemas un gran operador con automatización, presupuestos de errores e ingeniería de lanzamientos.
  https://sre.google/books/
- *Don't get locked up into avoiding lock-in*, Gregor Hohpe (2019), martinfowler.com. Un análisis equilibrado de los tipos de dependencia del proveedor y del costo real de evitarlos.
  https://martinfowler.com/articles/oss-lockin.html

Glosario

Activo-activo (multisitio)
Una configuración en la que dos o más regiones atienden tráfico real al mismo tiempo, de modo que perder una región no requiere failover, solo redistribuir el tráfico.

Ajuste de tamaño (rightsizing)
Adecuar el tamaño de los recursos aprovisionados (tipos de máquina, solicitudes de los contenedores, niveles de base de datos) a lo que una carga de trabajo usa realmente.

Análisis canario automatizado
Comparación estadística de las métricas de un canario con las de una línea base que ejecuta la versión anterior, que el pipeline usa para promover o revertir sin juicio humano.

Aplicación de doce factores (twelve-factor app)
Metodología de doce prácticas para construir aplicaciones web desplegables y escalables, escrita por Adam Wiggins en Heroku en 2011 y publicada como código abierto en 2024.

Arquitectura basada en células
Ejecutar muchas copias completas e independientes (células) de un sistema, cada una al servicio de un subconjunto fijo de clientes, detrás de un enrutador ligero, de modo que un fallo afecte solo a una célula.

Bucle de reconciliación
El ciclo sin fin de un controlador que observa el estado real, lo compara con el estado deseado y actúa para eliminar la diferencia.

Build once, deploy many (compilar una vez, desplegar muchas)
La regla de que un cambio se compila en un único artefacto inmutable exactamente una vez, y ese mismo artefacto se promueve por todos los entornos, cambiando solo la configuración.

Build, release, run (construcción, lanzamiento, ejecución)
La separación estricta entre convertir el código en un artefacto (construcción), combinarlo con la configuración de un entorno (lanzamiento) y ejecutarlo (ejecución).

Camino dorado (golden path o paved road)
Una forma definida, con soporte y documentada de realizar una tarea común, como crear un nuevo servicio, que es la más fácil pero no la única permitida.

Canary release (lanzamiento canario)
Enviar primero una pequeña parte del tráfico real a una nueva versión, comparar su comportamiento con el de la versión anterior y ampliar esa parte solo si está sana.

Capacidad spot
Capacidad sobrante de la nube vendida con un gran descuento que el proveedor puede recuperar con poca antelación; solo es adecuada para trabajo que tolera interrupciones.

Carga cognitiva
La cantidad de esfuerzo mental que una persona o un equipo debe dedicar para entender un sistema y trabajar en él; Team Topologies la utiliza para dimensionar las responsabilidades de los equipos.

Chargeback (imputación de costos)
Cargar al presupuesto de cada equipo o producto los costos que se le atribuyen, para que el gasto aparezca en sus propias cuentas.

Contenedores como servicio (CaaS)
Un servicio en la nube que ejecuta imágenes de contenedor por ti, ya sea como Kubernetes gestionado o como una plataforma de contenedores serverless en la que nunca ves las máquinas.

Costo de cambio
El esfuerzo, el dinero y el riesgo totales necesarios para pasar de una tecnología o un proveedor a otro.

Dependencia del proveedor (lock-in)
El costo y la dificultad de dejar un proveedor, producto, plataforma o tecnología una vez que un sistema depende de él.

Deployment (Kubernetes)
Un objeto de Kubernetes que declara cuántos pods idénticos de una aplicación sin estado deben ejecutarse y despliega las nuevas versiones de forma gradual mediante ReplicaSets.

Deriva de configuración
Diferencias que se acumulan entre la configuración prevista en el código y la infraestructura real, normalmente por cambios manuales o actualizaciones parciales.

Desarrollo basado en trunk (trunk-based development)
Forma de trabajar en la que los desarrolladores integran cambios pequeños en una única rama principal compartida al menos una vez al día, en lugar de mantener ramas de funcionalidad de larga duración.

Descuento por compromiso de uso
Un precio más bajo a cambio de comprometerse con un nivel de uso o de gasto durante uno o tres años, como las instancias reservadas, los savings plans o los descuentos por compromiso de uso (committed use discounts).

Desechabilidad (disposability)
La propiedad de los procesos que arrancan rápido, se detienen de forma ordenada y pueden eliminarse en cualquier momento sin causar daño, de modo que pueden sustituirse y escalarse libremente.

Despliegue azul-verde (blue-green)
Ejecutar dos entornos de producción completos y cambiar todo el tráfico del antiguo (azul) al nuevo (verde) de una sola vez, manteniendo el antiguo listo para volver a él al instante.

Despliegue gradual (rolling deployment)
Sustituir las instancias de un servicio unas pocas cada vez, de modo que la versión antigua y la nueva se ejecuten lado a lado hasta que todas las instancias ejecuten la nueva.

Economía unitaria (unit economics)
Expresar el costo (y los ingresos) por unidad de valor de negocio, como el costo por pedido o por cliente activo, para poder juzgar la eficiencia con independencia del crecimiento.

Entrega progresiva
Lanzar un cambio para audiencias cada vez mayores con comprobaciones automáticas en cada paso, de modo que un cambio defectuoso llegue al menor número posible de usuarios antes de detenerlo.

Estabilidad estática
La propiedad de que un sistema siga funcionando durante un fallo sin tener que hacer cambios, como lanzar servidores nuevos o actualizar la configuración, en el momento del fallo.

Estado (infraestructura como código)
El registro que mantiene una herramienta de infraestructura de qué recursos reales corresponden a qué definiciones del código, y que se usa para calcular qué crear, cambiar o eliminar.

Estado deseado
Una descripción declarativa de cómo debería ser un sistema (por ejemplo, tres copias de la versión 2.4), que la automatización compara con el estado real y trabaja para alcanzar.

Estrategia de salida
Un plan documentado, e idealmente ensayado, para abandonar un proveedor o un producto, que abarca la exportación de datos, los servicios de sustitución, los contratos, el tiempo y el costo.

FinOps
Una práctica en la que los equipos de ingeniería, finanzas y negocio comparten la responsabilidad del gasto en la nube y en tecnología, usando datos de costos oportunos para tomar decisiones que maximicen el valor para el negocio.

Función en el borde (edge function)
Un pequeño fragmento de código que una CDN o un proveedor de borde (edge) ejecuta en ubicaciones cercanas a los usuarios, para gestionar o modificar las solicitudes antes de que lleguen al origen (o en lugar de ello).

Gateway API
La API de Kubernetes para enrutar el tráfico externo e interno hacia los servicios, disponible de forma general desde 2023 y sucesora recomendada de Ingress.

Gestión de la configuración
Herramientas como Ansible, Chef y Puppet que llevan servidores existentes y de larga duración a una configuración descrita, instalando paquetes y editando archivos en ellos.

GitOps
Operar sistemas guardando su estado deseado declarativo en Git y dejando que unos agentes lo obtengan y reconcilien continuamente el sistema en ejecución para que coincida con él.

Guardrail (salvaguarda)
Un límite o una comprobación automáticos que mantienen seguro el uso en autoservicio (por ejemplo, una política que bloquea los buckets de almacenamiento públicos) sin que una persona tenga que aprobar cada solicitud.

Horizontal Pod Autoscaler (HPA)
Un controlador de Kubernetes que cambia el número de réplicas de pods para mantener una métrica, como el uso medio de CPU en relación con lo solicitado (requests), cerca de un objetivo.

Infraestructura como código (IaC)
Definir servidores, redes, bases de datos y otra infraestructura en archivos bajo control de versiones que las herramientas aplican automáticamente, en lugar de configurarlos a mano.

Infraestructura inmutable
La práctica de no modificar nunca en su sitio un servidor o un contenedor en ejecución; cada cambio se hace construyendo una nueva imagen y sustituyendo las instancias antiguas.

Ingeniería de plataformas
La disciplina de construir y operar plataformas internas que permiten a los equipos de producto aprovisionar, desplegar y operar software mediante autoservicio.

Kubernetes
El orquestador de contenedores de código abierto más usado, creado en Google, anunciado en 2014 y publicado como versión 1.0 en 2015; mantiene el estado real de un clúster en coincidencia con un estado deseado declarado.

Marco SPACE
Marco de 2021 para la productividad de los desarrolladores que abarca la satisfacción y el bienestar, el desempeño, la actividad, la comunicación y la colaboración, y la eficiencia y el flujo.

Modelo de responsabilidad compartida
El reparto de las obligaciones de seguridad y operación entre un proveedor de nube, que protege la propia nube, y el cliente, que protege sus datos, identidades y configuración en la nube.

Multinube (multi-cloud)
Usar servicios de dos o más proveedores de nube pública, por servicios diferenciales, adquisiciones, poder de negociación o planes de salida exigidos por la regulación.

Nativo de la nube (cloud native)
Un enfoque para construir y operar software como sistemas débilmente acoplados, automatizados, observables y resilientes, que se despliegan de forma programática y repetible en nubes públicas, privadas o híbridas. Describe cómo se construye un sistema, no dónde se ejecuta.

Nube híbrida
Una combinación de nube privada y pública (o de sistemas locales) conectadas de modo que los datos y las aplicaciones puedan funcionar entre ellas.

Nube privada
Infraestructura de nube dedicada a una sola organización, que ofrece aprovisionamiento de autoservicio basado en API; puede estar en las propias instalaciones o alojada por un tercero.

Open Container Initiative (OCI)
Proyecto de la Linux Foundation, fundado en 2015, que mantiene estándares abiertos para formatos de imagen de contenedor, entornos de ejecución y registros, de modo que las herramientas de distintos proveedores interoperen.

OpenTofu
Bifurcación (fork) de código abierto de Terraform creada en 2023 después de que HashiCorp pasara Terraform a la Business Source Licence; alojada por la Linux Foundation y proyecto de la CNCF.

Orquestación de contenedores
Programar automáticamente los contenedores en las máquinas y mantenerlos en ejecución: reiniciar los que fallan, escalar, desplegar nuevas versiones y conectarlos a redes y almacenamiento.

Pilot light (piloto encendido)
Estrategia de recuperación ante desastres que mantiene los datos replicados y la infraestructura básica aprovisionada en una segunda región, pero que solo arranca y escala los servidores de aplicaciones durante un desastre.

Pipeline de despliegue
La secuencia automatizada de etapas (compilación, pruebas, empaquetado, despliegue, verificación) por la que pasa cada cambio en su camino desde un commit hasta producción.

Plano de control
Los componentes que toman decisiones para un clúster o un sistema (en Kubernetes: el servidor de API, etcd, el planificador y los controladores), a diferencia de los nodos que ejecutan las cargas de trabajo.

Plataforma interna para desarrolladores (IDP)
El conjunto de herramientas, servicios, plantillas y documentación de autoservicio que proporciona el equipo de plataforma de una organización para que los equipos de producto puedan construir y operar software sin tickets.

Plataforma viable más delgada (thinnest viable platform)
El conjunto más pequeño de servicios compartidos y documentación que realmente acelera a los equipos de producto; la advertencia de Team Topologies contra construir más plataforma de la necesaria.

Pod
La unidad desplegable más pequeña de Kubernetes: uno o más contenedores planificados juntos en un nodo, que comparten una dirección de red y volúmenes de almacenamiento.

Política como código
Escribir reglas sobre la infraestructura y los despliegues (ningún almacenamiento público, cifrado obligatorio, regiones aprobadas) como código que las herramientas comprueban automáticamente antes o durante los cambios.

Portal para desarrolladores
Una interfaz web, como Backstage, que reúne en un solo lugar un catálogo de software, plantillas, documentación y enlaces a los servicios de la plataforma.

Recuperación ante desastres (DR)
Los planes, las copias y los procedimientos que restablecen un servicio y sus datos tras un fallo grave, como la pérdida de todo un centro de datos o de una región.

Repatriación desde la nube
Sacar cargas de trabajo de una nube pública y devolverlas a hardware propio o en colocation, normalmente porque una carga estable hace que la propiedad resulte más barata.

Repositorio de artefactos
Un almacén de resultados compilados y versionados, como imágenes de contenedor y paquetes, desde el que todos los entornos despliegan el mismo archivo.

Roll forward (corrección hacia adelante)
Recuperarse de un lanzamiento defectuoso desplegando una nueva corrección en lugar de volver a la versión anterior.

Salida de datos (egress)
Los datos que salen de la red de un proveedor de nube, o que pasan entre sus zonas o regiones, lo que normalmente se cobra por gigabyte.

Service (Kubernetes)
Objeto de Kubernetes que da a un conjunto cambiante de pods un nombre estable y una dirección virtual, y reparte el tráfico entre ellos.

Servicio de respaldo (backing service)
Cualquier servicio que una aplicación usa a través de la red, como una base de datos, una cola, una caché o una API de correo electrónico, tratado como un recurso adjunto que la configuración puede intercambiar.

Showback
Informar a cada equipo o producto de los costos que se le atribuyen, sin cargarlos realmente a su presupuesto.

Shuffle sharding
Asignar a cada cliente una combinación pequeña y elegida al azar de recursos, de modo que dos clientes rara vez compartan todos los suyos y un cliente problemático afecte a pocos más.

Solicitudes y límites de recursos (requests y limits)
Ajustes de CPU y memoria por contenedor en Kubernetes: las solicitudes (requests) reservan capacidad y orientan la planificación; los límites (limits) acotan el uso, estrangulando la CPU y terminando los contenedores que superan la memoria.

Sonda (liveness, readiness, startup)
Comprobación periódica que Kubernetes hace sobre un contenedor: la de actividad (liveness) decide si reiniciarlo, la de preparación (readiness) si enviarle tráfico y la de arranque (startup) si ha terminado de arrancar.

Tráfico sombra (shadow traffic, dark launch)
Copiar solicitudes reales a una versión nueva cuyas respuestas se comparan o se descartan pero nunca se devuelven a los usuarios, para probarla con carga real sin afectar a los usuarios.

Warm standby (espera semiactiva)
Estrategia de recuperación ante desastres que mantiene en funcionamiento, en una segunda región, una copia reducida pero plenamente operativa del sistema, lista para recibir tráfico y escalar.