BE SOFTWARE ARCHITECTURE WISE · CAPÍTULO 4

Documentar y comunicar la arquitectura

Este capítulo muestra cómo describir y comunicar una arquitectura de software para que las personas adecuadas la entiendan y la descripción siga siendo fiel a la realidad. Trata de para quién es la documentación y cuánto escribir, las vistas y los puntos de vista de ISO/IEC/IEEE 42010, el 4+1 y Rozanski y Woods, el modelo C4, los diagramas UML que los arquitectos todavía usan, los registros de decisiones de arquitectura (ADR), arc42, los diagramas como código y la presentación de la arquitectura a distintos públicos. Es práctico de principio a fin, usa una plataforma de reparto de comida como ejemplo recurrente y está escrito para desarrolladores, arquitectos y líderes técnicos.

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

DE UN VISTAZO

Aprenderás a

- Explicar quién lee la documentación de arquitectura, qué necesita cada lector y cómo decidir qué merece la pena dejar por escrito.
- Distinguir las partes interesadas, las inquietudes, los puntos de vista, las vistas y los tipos de modelo tal como los define ISO/IEC/IEEE 42010:2022.
- Comparar el modelo de vistas 4+1, los puntos de vista y las perspectivas de Rozanski y Woods, y las tres categorías de vistas del SEI.
- Dibujar diagramas de contexto del sistema, de contenedores y de componentes que sigan las reglas de notación de C4.
- Leer y dibujar los diagramas UML de secuencia, de máquina de estados, de componentes, de despliegue, de actividades y de clases que usan los arquitectos.
- Escribir registros de decisiones de arquitectura y gestionar su ciclo de vida en una bitácora de decisiones.
- Aplicar arc42 y las prácticas de diagramas como código para que la documentación siga siendo útil y esté al día.
- Adaptar una explicación de la arquitectura a directivos, desarrolladores, operaciones, personal de seguridad y auditores.

Antes de empezar

Capítulos 1 a 3, especialmente las partes interesadas y las inquietudes (Sección 1.5) y las decisiones y los compromisos (Sección 1.6).

Tiempo

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

Términos clave

Descripción de arquitectura · Vista · Punto de vista · Inquietud · Modelo de vistas 4+1 · Modelo C4 · Contenedor (C4) · UML · Diagrama de secuencia · Diagrama de máquina de estados · Registro de decisiones de arquitectura · arc42 · Diagramas como código · Documentación como código (docs as code)

Por qué documentar la arquitectura forma parte del trabajo

Una arquitectura que solo existe en la cabeza de unas pocas personas es frágil. Nadie fuera de la sala puede revisarla, el próximo ingeniero que se incorpore no puede aprenderla, un auditor no puede comprobarla, y se va con las personas que la conocen. El Capítulo 1 separó una arquitectura (lo que el sistema es realmente) de una descripción de arquitectura (los documentos, diagramas y registros mediante los cuales las personas la comparten). Este capítulo trata de la segunda: cómo producir descripciones que la gente realmente lea, que sigan siendo fieles a medida que el sistema cambia y que ayuden a tomar decisiones.

Es más una habilidad práctica que teórica, y es fácil equivocarse en cualquiera de las dos direcciones. Los equipos que escriben demasiado poco pierden las razones de su estructura y repiten una y otra vez las mismas discusiones. Los equipos que escriben demasiado producen wikis en las que nadie confía. El principio rector del capítulo procede de *Documenting Software Architectures: Views and Beyond*, del SEI: escribe para el lector. Decide quién usará cada pieza de documentación y qué necesita de ella, escribe eso de forma breve y precisa, y no escribas nada más.

Un mismo ejemplo recorre todo el capítulo: una plataforma de reparto de comida con una app de clientes, una app de restaurantes y una app de repartidores, servicios de pedidos, de despacho de repartidores y de notificaciones, y pagos con tarjeta a través de un proveedor externo. La verás como diagrama de contexto, diagrama de contenedores, diagrama de componentes, diagramas UML de secuencia y de estados, registro de decisiones y resumen arc42 de una página, para que puedas comparar cada técnica sobre el mismo sistema.

Describir y dibujar: secciones 4.1 a 4.4

Las cuatro primeras secciones tratan de *qué* describir y *cómo dibujarlo*. La Sección 4.1 se pregunta para quién es la documentación, qué dice y qué no dice el Manifiesto Ágil sobre ella, cómo decidir qué merece ponerse por escrito y qué antipatrones evitar. La respuesta corta es escribir a mano lo que el código no puede decirte (el contexto, los objetivos de calidad, las decisiones y sus razones) y generar lo que el código ya expresa.

La Sección 4.2 presenta la idea central de que un solo diagrama nunca basta. Una arquitectura se describe mediante varias vistas, cada una de las cuales responde a las inquietudes de determinadas partes interesadas y se construye según un punto de vista. La sección trata del modelo conceptual de ISO/IEC/IEEE 42010:2022, el modelo de vistas 4+1 de Kruchten, los puntos de vista y las perspectivas de Rozanski y Woods, y las categorías de módulos, de componentes y conectores y de asignación del SEI, y muestra cómo elegir el conjunto mínimo de vistas que necesita un sistema.

La Sección 4.3 convierte esa teoría en el enfoque práctico más utilizado, el modelo C4 de Simon Brown: cuatro niveles de zoom (contexto del sistema, contenedor, componente, código), unos pocos diagramas complementarios y un breve conjunto de reglas de notación que hacen que los diagramas se expliquen solos. La Sección 4.4 trata de las partes de UML que los arquitectos todavía usan, sobre todo los diagramas de secuencia, de máquina de estados, de componentes, de despliegue, de actividades y los diagramas de clases pequeños, y de cuándo la precisión de UML compensa el esfuerzo.

Decisiones, plantillas, herramientas y personas: secciones 4.5 a 4.8

Los diagramas muestran la estructura; rara vez muestran el *porqué*. La Sección 4.5 trata de los registros de decisiones de arquitectura: el formato de Michael Nygard de 2011, alternativas como MADR y las declaraciones Y, el ciclo de vida de un registro desde propuesto hasta sustituido, y un registro práctico completo para la plataforma de reparto. La Sección 4.6 muestra cómo una plantilla como arc42 da a toda la descripción una organización estándar, para que quienes escriben sepan qué cubrir y quienes leen sepan dónde buscar, y cómo usarla de forma ligera en un sistema pequeño.

La Sección 4.7 aborda el problema que derrota a la mayor parte de la documentación: la obsolescencia. Trata de escribir diagramas como texto con PlantUML, Mermaid, Structurizr DSL y D2, la diferencia entre las herramientas de dibujo y las de modelado, la publicación de la documentación desde el pipeline de compilación y los hábitos de propiedad y revisión que mantienen la documentación fiel a la realidad. La Sección 4.8 amplía el foco de los documentos a las personas: adaptar el mensaje a directivos, desarrolladores, personal de operaciones, equipos de seguridad y auditores, explicar los compromisos, seguir una solicitud a través de un sistema, los documentos de diseño y los procesos de RFC, presentar en una revisión y construir consenso cuando las personas discrepan.

La idea que debes llevarte

Todas las técnicas de este capítulo sirven a un único objetivo: un entendimiento compartido y preciso del sistema entre las personas que lo construyen, lo operan, lo pagan y dependen de él. Un diagrama, un registro de decisiones o una plantilla solo merecen producirse si un lector real los necesita, y solo mientras alguien los mantenga fieles a la realidad. Una documentación pequeña, actual y con propietario, que vive junto al código, es mejor que un documento exhaustivo en el que nadie confía. Si recuerdas una sola pregunta de este capítulo, que sea esta: *¿quién va a leer esto y qué hará con ello?*

SECCIÓN 4.1 DE 4.8 · INTERMEDIO · 28 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Por qué y cuánto documentar

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.1 Definir la arquitectura
- 1.5 Partes interesadas e inquietudes

Términos anteriores que utiliza: Descripción de arquitectura, Parte interesada, Control de versiones, Pull request

LA IDEA EN BREVE

Piensa en la documentación de una casa. Un comprador quiere un plano de la planta, un electricista quiere el esquema de la instalación eléctrica, una aseguradora quiere el informe de tasación, y quien la reforme dentro de veinte años quiere saber por qué nunca se movió aquella pared. Nadie quiere una lista de todos los ladrillos, y un plano que muestra una pared que ya se ha derribado es peor que no tener ningún plano. La documentación de arquitectura de software es igual: un pequeño conjunto de documentos precisos, cada uno escrito para alguien que realmente lo necesita, que explican lo que no se puede ver solo con mirar el código.

LO QUE NECESITAS SABER

- Una descripción de arquitectura es el conjunto de documentos, diagramas y registros mediante los cuales las personas comparten una arquitectura; sirve para la formación, la comunicación entre las partes interesadas y el análisis.
- Los distintos lectores (recién llegados, desarrolladores, operaciones, seguridad, auditores, futuros responsables del mantenimiento y responsables de las decisiones) traen preguntas diferentes, por lo que la documentación debe organizarse en torno a los lectores y sus tareas.
- El Manifiesto Ágil valora el software que funciona por encima de la documentación exhaustiva, pero dice explícitamente que la documentación sigue teniendo valor; rechaza la documentación como fin en sí misma, no la documentación.
- Escribe a mano lo que no puede recuperarse a partir del código (el contexto, los objetivos de calidad, las decisiones y sus razones, los principios); genera lo que el código ya expresa y cambia a menudo.

- Las siete reglas de Views and Beyond: escribe para el lector, evita la repetición innecesaria, evita la ambigüedad, usa una organización estándar, registra la justificación, mantén la documentación al día pero no demasiado al día, y revísala para comprobar que es adecuada para su propósito.
- La documentación como código y la documentación viva mantienen la documentación bajo control de versiones, revisada junto con los cambios de código y derivada del sistema siempre que sea posible, lo que constituye la defensa más fiable contra la obsolescencia.
- Los antipatrones habituales son la wiki obsoleta, el documento big-bang que nadie lee, los diagramas sin leyenda, los niveles de abstracción mezclados y las descripciones que omiten el porqué.

Por qué hay que poner una arquitectura por escrito

Todo sistema tiene una arquitectura, la describa alguien o no (consulta la Sección 1.1). La descripción de arquitectura es otra cosa: los documentos, diagramas, modelos y registros de decisiones mediante los cuales las personas comparten un entendimiento de esa arquitectura. En un equipo de tres personas que trabajan juntas y escribieron cada línea, gran parte de la descripción puede vivir en su cabeza. En cuanto el equipo crece, alguien se va, se incorpora un segundo equipo, llega un auditor o el sistema sobrevive a sus autores, el conocimiento que solo existe en la cabeza de las personas se convierte en un riesgo. La definición de Ralph Johnson de la arquitectura como el entendimiento compartido de los desarrolladores expertos encierra una advertencia: un entendimiento que no está por escrito solo puede compartirse hablando, y hablar no escala.

Documenting Software Architectures: Views and Beyond (Paul Clements y sus colegas del Software Engineering Institute, 2.ª edición, 2010) señala tres usos de la documentación de arquitectura, y siguen siendo la forma más clara de pensar por qué merece la pena el esfuerzo. **Forma**: presenta el sistema a los nuevos miembros del equipo, a los revisores externos y a los nuevos arquitectos. Es el principal vehículo de **comunicación entre las partes interesadas**: desarrolladores, operadores, personal de seguridad y directivos discuten sobre el sistema a través de ella, y las decisiones alcanzadas en una reunión llegan a quienes no estuvieron presentes. Y es la **base para el análisis y la construcción**: indica a quienes implementan lo que pueden y no pueden hacer, y es lo que examina un evaluador cuando se pregunta si el sistema cumplirá sus objetivos de calidad (los métodos de evaluación del Capítulo 11 necesitan una descripción de la que partir).

De ello se derivan dos consecuencias. Primero, la documentación es un medio, no un fin: un documento que no sirve a ninguno de estos usos es un desperdicio, por bien formateado que esté. Segundo, es el lector quien decide si ha funcionado. Una descripción que su autor

considera completa pero que un recién llegado no puede usar ha fracasado en la formación, y un diagrama que el equipo de operaciones no puede relacionar con los servidores reales ha fracasado en el análisis.

Quién lee la documentación de arquitectura y qué necesita

Lector	Preguntas que trae	Lo que mejor le sirve
Recién llegados (nuevos desarrolladores, contratistas, un nuevo arquitecto)	¿Qué hace este sistema, para quién y por dónde empiezo? ¿Cuáles son las partes principales y las palabras que usa la gente?	Una visión general de una página, un diagrama de contexto del sistema, un diagrama de contenedores, un glosario y un breve orden de lectura
Desarrolladores actuales	¿Dónde encaja esta nueva funcionalidad? ¿De qué puedo depender? ¿Por qué está construido así y puedo cambiarlo?	Estructura de módulos o componentes, convenciones, conceptos transversales (gestión de errores, seguridad, logging) y registros de decisiones
Personal de operaciones y SRE	¿Dónde se ejecuta, de qué depende, cómo falla y cómo nos recuperamos?	Vista de despliegue por entorno, dependencias externas, runbooks, monitorización y objetivos de recuperación
Personal de seguridad y privacidad	¿Dónde están los límites de confianza? ¿Por dónde fluyen y dónde se almacenan los datos personales? ¿Quién puede llamar a qué?	Vistas de contexto y de flujo de datos, concepto de identidad y acceso, lista de servicios de terceros
Auditores y reguladores	¿Se puede demostrar que los controles existen, que las decisiones se tomaron de forma deliberada y que los cambios son trazables?	Registros de decisiones con fechas y responsables, trazabilidad desde los requisitos hasta el diseño y las pruebas, historial de versiones
Futuros responsables del mantenimiento	¿Por qué demonios se hizo esto? ¿Qué restricciones siguen vigentes? ¿Qué se puede eliminar sin riesgo?	Ante todo, la justificación: las decisiones con su contexto, las opciones rechazadas y los supuestos asumidos
Responsables de las decisiones (patrocinadores, responsables de producto y de ingeniería)	¿Cuáles son los riesgos, los costos y las opciones? ¿Qué tengo que decidir y para cuándo?	Un resumen de una página, los objetivos de calidad, los principales riesgos y la deuda técnica, las decisiones pendientes de un responsable

El software que funciona por encima de la documentación exhaustiva

El Manifiesto Ágil (2001) enumera cuatro pares de valores, uno de los cuales es “Software que funciona por encima de la documentación exhaustiva”. Se cita, a menudo, como permiso para no documentar nada. Leído completo, el manifiesto dice algo más acotado. Sus autores afirman que, aunque valoran los elementos de la derecha, valoran más los de la izquierda. El valor es una prioridad para cuando ambos compiten, no un rechazo de la documentación.

Lo que el valor rechaza es el proyecto dirigido por la documentación de la década de 1990: cientos de páginas de especificaciones aprobadas antes de escribir el código, tratadas como un avance en sí mismas y abandonadas en cuanto el código se aparta de ellas. La palabra que importa es *exhaustiva*: una documentación que intenta describirlo todo, como fin en sí misma, y que se valora más que el sistema en funcionamiento. Nada en el manifiesto se opone a un diagrama de contexto, a un breve registro de por qué el equipo eligió la mensajería asíncrona o a un runbook que permite al ingeniero de guardia restablecer el servicio a las tres de la madrugada.

Los profesionales sí discrepan sobre hasta dónde llegar. Algunos coaches ágiles sostienen que la conversación y las pruebas casi siempre bastan, y que la documentación escrita de arquitectura se queda obsoleta antes de amortizarse. Otros, sobre todo quienes mantienen sistemas de larga vida o trabajan en sectores regulados, señalan que las personas que más necesitan la documentación (los sucesores, los auditores, el equipo de operaciones) son precisamente las que están ausentes de las conversaciones. Ambas posturas coinciden en la economía subyacente: la documentación tiene un costo de escritura, un costo de mantenimiento y un costo de lectura, y solo merece producirse donde su valor para los lectores supera esos costos.

Documentación justa y necesaria

La expresión de Scott Ambler "just barely good enough" (apenas lo bastante buena) describe el objetivo: un modelo o un documento es suficiente cuando cumple su propósito para su público, y el esfuerzo más allá de ese punto cuesta más de lo que aporta. Su idea de que es el público, y no el autor, quien decide si un artefacto es suficiente es el mismo principio que escribir para el lector. George Fairbanks hace el argumento equivalente para el trabajo de arquitectura en su conjunto en *Just Enough Software Architecture* (2010): haz tanto como justifiquen los riesgos (consulta la Sección 1.8).

Por tanto, la documentación justa y necesaria no es una cantidad fija. Crece con el tamaño y la vida útil del sistema, el número de equipos, la rotación de personas, el costo de un fallo y las exigencias de los reguladores. Un prototipo de dos personas necesita un README y quizá un diagrama. Una plataforma de pagos gestionada por ocho equipos bajo regulación financiera

necesita vistas de contexto, de contenedores y de despliegue, una bitácora de decisiones que se remonte años atrás, conceptos de seguridad documentados y evidencias de que los documentos se revisan.

Una señal práctica es la pregunta repetida. Si los recién llegados preguntan lo mismo en su primera semana, o la misma discusión de diseño vuelve cada pocos meses, la respuesta debe ponerse por escrito. Otra es el error costoso: si un equipo incumplió una regla no escrita ("no llamar nunca al proveedor de pagos desde la app de restaurantes") porque nadie se la dijo, esa regla era arquitectura y debería haberse puesto por escrito.

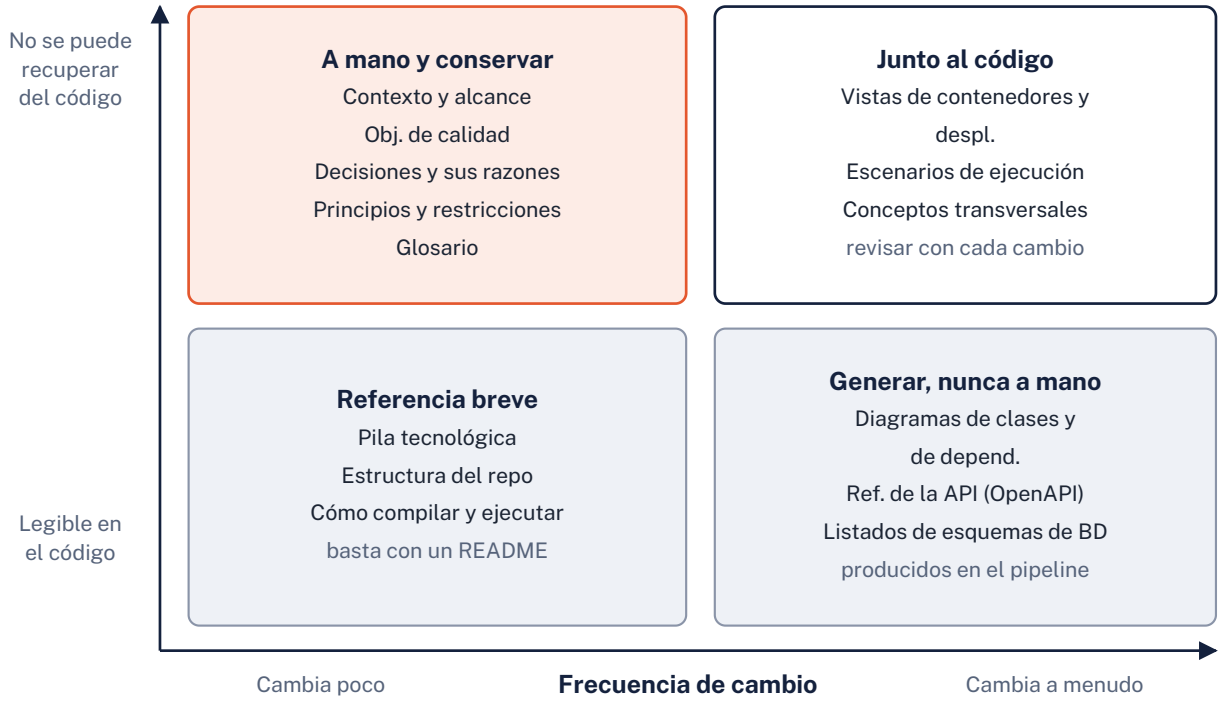
Qué merece documentarse y qué no

La información difiere en dos aspectos que deciden cómo debe documentarse. El primero es si puede **recuperarse a partir del código**. Los nombres de las clases, las firmas de los métodos y las tablas de la base de datos pueden leerse en el código fuente en cualquier momento; las razones por las que se tomó una decisión, los objetivos de calidad a los que sirve el sistema y las opciones que se rechazaron no, porque nunca estuvieron en el código. El segundo es **con qué frecuencia cambia**. El propósito del sistema y sus principales socios externos cambian poco; la lista de clases de un módulo cambia cada semana.

La documentación de arquitectura escrita a mano debe concentrarse en lo que es a la vez importante y no recuperable a partir del código: el **contexto y el alcance** del sistema (quién lo usa y de qué otros sistemas depende), sus **objetivos de calidad** y restricciones, las **decisiones clave y sus razones**, la **estructura** de grano grueso (las principales partes desplegables y sus responsabilidades), el **despliegue** sobre la infraestructura, los **escenarios de ejecución** importantes y los **conceptos transversales** que toda parte debe seguir, como la forma de gestionar los errores, de autenticar a los usuarios y de proteger los datos personales. Esta lista se parece mucho a la estructura de la plantilla arc42, que se trata en la Sección 4.6.

Lo que no merece escribirse a mano es todo lo que el código o las herramientas ya expresan con precisión y que cambia a menudo: los diagramas de clases de todo un código base, las listas de endpoints que ya contiene un archivo OpenAPI o las descripciones de cada función. Cuando esas vistas sean útiles, genéralas a partir del código fuente en el pipeline de compilación para que no puedan desviarse. Omite también el contenido aspiracional (diagramas de cómo debería ser el sistema, presentados como si ya lo fuera), las actas de reuniones cuyo resultado ya está recogido en otro lugar y el texto duplicado, que acabará contradiciendo a su copia.

Cómo decidir la forma de documentar una información



Escribe a mano lo que el código no puede decirte; genera lo que el código ya dice y cambia a menudo.

SI ESTO ES NUEVO PARA TI

Paso a paso: escribirlo, generarlo o hacer referencia a ello

La idea que requiere más atención aquí es decidir, para cada pieza de conocimiento sobre un sistema, si una persona debe ponerla por escrito, si una herramienta debe generarla o si basta con una breve referencia. Dos preguntas lo resuelven: ¿puede leerse en el código?, y ¿con qué frecuencia cambia? Prueba con cinco hechos sobre la plataforma de reparto de comida.

- Por qué los eventos de pedido se envían de forma asíncrona en lugar de mediante llamadas directas. El código no dice por qué ni qué se rechazó, y las razones cambian poco. Una persona las pone por escrito una vez, como registro de decisiones de arquitectura.
- La regla 'no llamar nunca al proveedor de pagos desde la app de restaurantes'. El código muestra que la app no llama hoy al proveedor, no que nunca deba hacerlo. Un equipo que nunca oyó la regla podría incumplirla, así que también se pone por escrito.
- La lista de endpoints del servicio de pedidos. Su archivo OpenAPI ya los expresa con precisión, y cambian a menudo, así que el pipeline genera la referencia a partir de ese archivo.
- El diagrama de las principales apps, servicios y bases de datos de la plataforma y de cómo se conectan. Ningún código base muestra por sí solo esa imagen completa, y sin embargo cambia cada vez que se añade o se elimina una parte. Así que se guarda en el repositorio, idealmente dibujado a partir de un modelo que está allí, y el pull request que añade o elimina una parte también lo modifica.
- Los comandos para compilar y ejecutar cada servicio. Están en los archivos de compilación y cambian poco, así que basta con un breve README que remita a ellos.
- **La lección:** dedica el esfuerzo de escritura a mano a lo que el código calla: razones, reglas, objetivos y la visión de conjunto. Genera o remite a todo lo que el código ya expresa: las copias hechas a mano se desvían.

Siete reglas para una documentación sólida

Views and Beyond establece siete reglas para una documentación sólida. Vale la pena saberlas de memoria, porque casi todos los fallos de documentación incumplen alguna de ellas.

1

1. Escribe para el lector

La documentación se escribe una vez y se lee muchas, así que optimízala para el lector. Decide quiénes son los lectores y qué intentarán hacer con el documento, organízalo para la consulta y no para la lectura lineal, y evita la jerga privada. Una señal de que se incumple esta regla: el documento está organizado en el orden en que al autor se le fueron ocurriendo las cosas.

2

2. Evita la repetición innecesaria

Cada información debe tener un único lugar, con enlaces desde otras partes. La repetición alarga los documentos y, peor aún, deja que las copias diverjan hasta que los lectores no pueden saber cuál es la verdadera. La repetición deliberada para comodidad del lector, como un resumen, está bien si se señala claramente.

3

3. Evita la ambigüedad

Un diagrama de cajas y líneas sin leyenda puede leerse de varias maneras: ¿una línea es una llamada, un flujo de datos o una dependencia de despliegue? Define la notación en una leyenda, usa un solo nombre para cada cosa e indica qué significa cada elemento y cada relación.

4

4. Usa una organización estándar

Una estructura conocida (una plantilla como arc42 o la plantilla de Views and Beyond) indica a los lectores dónde buscar y muestra a los autores lo que falta. Una sección vacía marcada como "no aplicable, porque..." informa más que un hueco silencioso.

5

5. Registra la justificación

Pon por escrito el porqué, incluidas las opciones rechazadas. Los futuros responsables del mantenimiento suelen poder averiguar qué se construyó; no pueden averiguar por qué, y sin el porqué o temen cambiar cualquier cosa o deshacen una decisión cuyas razones siguen siendo válidas. Para eso existen los registros de decisiones.

6

6. Mantenla al día, pero no demasiado al día

La documentación desactualizada induce a error. Pero actualizar todos los documentos con cada idea a medio formar genera agitación constante; registra las decisiones una vez tomadas y actualiza la descripción con un ritmo predecible, por ejemplo con cada lanzamiento o con cada cambio significativo.

7

7. Revísala para comprobar que es adecuada para su propósito

Solo los lectores previstos pueden decir si un documento funciona. Pide a un recién llegado que prepare y explique el sistema a partir de los documentos, o pide a un ingeniero de guardia que use la vista de despliegue en un simulacro, y corrige aquello con lo que tropiecen.

Documentación viva y documentación como código

El fallo tradicional de la documentación de arquitectura es la obsolescencia: la descripción se escribe al principio, el sistema cambia y nadie actualiza el documento porque vive en un lugar separado del trabajo. Dos ideas relacionadas atacan este problema de raíz.

La documentación como código, una práctica promovida por la comunidad Write the Docs, consiste en tratar la documentación como código fuente: escrita en formatos de texto plano como Markdown o AsciiDoc, guardada bajo control de versiones (normalmente en el mismo repositorio que el código que describe), modificada mediante los mismos pull requests, revisada por las mismas personas y compilada y publicada por el mismo pipeline. Así, se puede exigir que un pull request que cambia la forma de asignar los pedidos actualice la vista de ejecución en el mismo commit, y el revisor ve ambas cosas.

La documentación viva, un término popularizado por el libro homónimo de Cyrille Martraire (2019), va más allá: siempre que sea posible, la documentación debe derivarse del propio sistema para que no pueda desviarse. Los diagramas de dependencias generados a partir del código, las referencias de API generadas a partir de archivos OpenAPI, las reglas de arquitectura comprobadas por pruebas automatizadas y los diagramas generados a partir de un modelo guardado junto al código son ejemplos de ello. El texto escrito a mano se reserva para lo que no puede derivarse, principalmente la intención y la justificación.

Ninguna de las dos ideas elimina la necesidad de criterio sobre qué escribir. La Sección 4.7, Diagramas como código, trata en detalle las herramientas y los hábitos.

GUÍA DE DECISIÓN

Guía de decisión: ¿deberíamos poner esto por escrito?

Hazte estas preguntas en orden sobre una pieza de conocimiento arquitectónico.

Q1 **¿Cometería alguien un error costoso, o haría la misma pregunta una y otra vez, si esto no estuviera por escrito?**

- **Sí** Continúa: es candidato a documentarse.
- **No** Déjalo por ahora a la conversación y al código; vuelve a planteártelo si la pregunta empieza a repetirse.

Q2 **¿Puede leerse de forma fiable en el código, en la configuración o en una herramienta?**

- **Sí** Genéralo o enlaza al lugar donde está; no lo copies a mano.
- **No** Continúa: necesita que lo escriba una persona.

Q3 **¿Es una decisión con razones y opciones rechazadas?**

- **Sí** Escribe un registro de decisiones (consulta la Sección 4.5).
- **No** Continúa.

Q4 **¿Cambiará con la mayoría de los lanzamientos?**

- **Sí** Guárdalo en el repositorio junto al código que describe y haz que actualizarlo forme parte del pull request.
- **No** Ponlo en la visión general de la arquitectura o en la sección correspondiente de la plantilla y revísalo con un ritmo fijo.

EJEMPLO PRÁCTICO

Ejemplo práctico: un conjunto de documentación justa y necesaria para la plataforma de reparto de comida

Una empresa de reparto de comida opera una plataforma con una app de clientes, una app de restaurantes, una app de repartidores, servicios de pedidos, de despacho de repartidores y de notificaciones, y un proveedor de pagos externo. Tiene 30 ingenieros en cinco equipos y se prepara para una auditoría de un evaluador de pagos con tarjeta. El arquitecto propone este conjunto de documentación, guardado en su totalidad en un único repositorio de documentación o junto al código, y nada de él más largo de lo necesario.

Visión general de la arquitectura (de 2 a 3 páginas)	Propósito, usuarios principales, objetivos de calidad por orden de prioridad (por ejemplo: no perder nunca un pedido ni cobrarlo dos veces; confirmación del pedido con p99 inferior a 1.5 segundos en hora punta; app de restaurantes utilizable con conexiones deficientes), restricciones clave y un orden de lectura para los recién llegados. Responsable: el arquitecto. Se revisa cada trimestre.
Diagramas de contexto del sistema y de contenedores	Dos diagramas C4 (Sección 4.3) con leyenda, generados a partir de un modelo en el repositorio. Se actualizan en el pull request que añade o elimina un contenedor o una dependencia externa.
Bitácora de decisiones (decision log)	Unos 40 registros de decisiones breves, numerados, en el repositorio: eventos de pedido asíncronos, la elección del proveedor de pagos, una base de datos por servicio, etc. Nunca se reescriben; se sustituyen cuando cambia una decisión.
Vista de despliegue	Un diagrama y una tabla por entorno: regiones, clústeres, bases de datos gestionadas, el intermediario de mensajes (message broker) y el límite de red en torno a los datos de tarjetas. Responsable: el equipo de plataforma.
Escenarios de ejecución	Diagramas de secuencia de los cuatro flujos que más importan: hacer un pedido, rechazo del restaurante, asignación de repartidor, reembolso. Cada uno cabe en una pantalla.
Conceptos transversales	Una página para cada uno: autenticación, idempotencia y reintentos, logging y datos personales. Son las reglas que todo equipo debe seguir.
Runbooks	Uno por cada alerta que pueda despertar a alguien, enlazado desde la propia alerta.
No se escribe a mano	Diagramas de clases, listas de endpoints y esquemas de base de datos (generados en el pipeline); actas de reuniones; la historia de cómo creció el sistema (la bitácora de decisiones y el historial de versiones ya la cuentan).

Cada elemento tiene un lector concreto, un responsable y un desencadenante de actualización. La documentación sin responsable es la primera en degradarse.

La auditoría se atiende con la bitácora de decisiones, la vista de despliegue y el historial de versiones, no con un documento de auditoría aparte que habría que mantener coherente con ellos.

El conjunto completo puede leerse en unas dos horas, lo que hace verosímil que los recién llegados lo lean de verdad.

ATENCIÓN

Antipatrones de documentación

Estos son los fallos que más se ven en la práctica, y la regla que incumple cada uno.

- **La wiki obsoleta.** Cientos de páginas, editadas por última vez hace años, en parte ciertas y sin forma de saber qué parte. Los lectores aprenden a desconfiar de todo. Remedio: menos páginas, cada una con un responsable y una fecha de "última revisión", y eliminar lo que nadie mantiene.
- **El documento big-bang.** Una especificación de arquitectura de 120 páginas escrita al principio, aprobada una vez y nunca vuelta a leer. Incumple la regla de escribir para el lector y queda desactualizada en cuestión de meses.
- **Diagramas sin leyenda.** Cajas de distintos colores y líneas de distintos estilos sin ninguna leyenda. Cada lector inventa un significado, y dos lectores inventan significados distintos.
- **Niveles de abstracción mezclados.** Un solo diagrama que muestra una app móvil, una clase Java, un clúster de Kubernetes y un proceso de negocio. Usa vistas separadas (Sección 4.2).
- **Documentación solo en diapositivas.** Las presentaciones se hacen para una reunión y un público, rara vez se versionan y no se pueden buscar. Guarda la fuente de verdad en el repositorio y genera las diapositivas a partir de ella.
- **El qué sin el porqué.** Una descripción completa de la estructura sin ninguna justificación, de modo que los equipos futuros no pueden saber qué restricciones son esenciales y cuáles fueron accidentes.
- **Documentación de solo escritura.** Producida para marcar una casilla de gobernanza, nunca revisada por un lector real y, por tanto, nunca puesta a prueba para comprobar que es adecuada para su propósito.

Preguntas frecuentes

Nuestro código es limpio y está bien probado. ¿No es documentación suficiente?
El código limpio y las pruebas son una documentación excelente de *qué* hace el sistema a un nivel de detalle fino. No muestran la visión de conjunto entre servicios y repositorios, los objetivos de calidad, las restricciones ajenas al código ni por qué la estructura es como es. Eso es exactamente para lo que sirve la documentación de arquitectura.

¿Quién debería escribir la documentación de arquitectura?
Las personas que toman las decisiones y conviven con ellas, lo que en la mayoría de los equipos modernos significa tanto los ingenieros como los arquitectos. El arquitecto suele ser responsable de la estructura general y de la coherencia, y los equipos son responsables de las partes que describen sus servicios. Lo que más importa es que cada documento tenga un responsable concreto.

¿Cómo evitamos que la documentación se quede obsoleta?
Mantenla pequeña, mantenla junto al código, haz que actualizarla forme parte de la definición de terminado para los cambios que la afecten, genera lo que pueda generarse y revísala con un ritmo fijo. La Sección 4.7 trata estos hábitos en detalle.

Portales para desarrolladores: hacer que la documentación se encuentre

La documentación que nadie encuentra es como si no existiera. Un portal para desarrolladores da a los ingenieros un único lugar donde buscar. Backstage, el más conocido (Sección 10.7), gira en torno a un catálogo de software creado a partir de archivos de metadatos guardados con el código, de modo que la propiedad queda registrada y se actualiza mediante las revisiones normales. Su función TechDocs toma Markdown guardado junto al código y lo publica como un sitio de documentación, accesible desde la página del servicio en el catálogo. En la plataforma de reparto de comida, la página de Pedidos podría mostrar juntos el equipo propietario, sus registros de decisiones y sus runbooks. Un portal es un índice, no una cura: las páginas desactualizadas siguen desactualizadas, y mantener uno tiene un costo. A un equipo pequeño puede bastarle un repositorio bien cuidado.

AsyncAPI y los registros de esquemas como documentación de contratos

Es fácil perder de vista los flujos de eventos, porque ningún servicio los muestra por sí solo. Un documento AsyncAPI (Sección 6.5) los registra en JSON o YAML legible por máquina: los canales, qué aplicaciones envían o reciben en ellos y los formatos de los mensajes. Funciona con protocolos como Kafka, MQTT y AMQP; consulta la versión actual. Un registro de

esquemas añade control. El de Confluent, por ejemplo, comprueba cada nueva versión de un esquema con una regla de compatibilidad, hacia atrás por defecto, y rechaza las incompatibles, y Apicurio Registry, de código abierto, también puede almacenar documentos OpenAPI y AsyncAPI (Sección 7.7). Así, la documentación de `OrderPlaced` no puede alejarse en silencio de lo que se publica. Ninguno de los dos explica qué *significa* un campo; eso sigue necesitando palabras.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **Nombra los tres usos de la documentación de arquitectura que identifica Views and Beyond.**

La formación (presentar el sistema a las personas), la comunicación entre las partes interesadas y servir de base para el análisis y la construcción.

▼ **¿Significa "el software que funciona por encima de la documentación exhaustiva" que los equipos ágiles no deben documentar su arquitectura?**

No. El manifiesto afirma que los elementos de la derecha siguen teniendo valor; establece una prioridad para cuando ambos compiten y rechaza la documentación producida como fin en sí misma, no la documentación útil.

▼ **¿Por qué la justificación debe documentarse a mano y los diagramas de clases, en general, no?**

La justificación no puede recuperarse a partir del código y cambia poco, así que debe escribirla una persona. La estructura de clases puede leerse o generarse a partir del código y cambia a menudo, así que una copia dibujada a mano se desvía rápidamente.

▼ **¿Contra qué advierte "mantén la documentación al día, pero no demasiado al día"?**

Tanto contra la documentación obsoleta que induce a error a los lectores como contra la agitación constante de actualizar los documentos con cada idea sin asentar; actualiza con un ritmo predecible una vez tomadas las decisiones.

▼ **Inténtalo: Un equipo de venta de entradas tiene una wiki de 80 páginas, desactualizada, que nadie lee. Los desarrolladores preguntan con qué sustituirla y temen perder el tiempo. ¿Qué escribirías a mano, qué generarías o qué eliminarías, y cómo se mantendría al día?**

Escribe a mano lo que el código no puede decirte: contexto, objetivos de calidad, principios y decisiones con sus razones, en forma de ADR breves. Genera lo que el código ya indica y cambia a menudo, como las referencias de API. Elimina las páginas obsoletas en lugar de parchearlas. Mantenlo todo en el repositorio, revisado en el mismo pull request que el cambio de código, para que siga al día. Organízalo en torno a las preguntas de los lectores, por ejemplo las de un recién llegado o las de un operador, no en torno a la estructura antigua.

FUENTES

- Manifesto for Agile Software Development: https://agilemanifesto.org/
- Documenting Software Architectures: Views and Beyond, 2nd edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- Just Barely Good Enough models and documents (Scott Ambler, Agile Modeling): https://agilemodeling.com/essays/barelygoodenough.htm
- Docs as Code (Write the Docs): https://www.writethedocs.org/guide/docs-as-code/
- Principles of technical documentation (Gernot Starke, INNOQ): https://www.innoq.com/en/blog/2022/01/principles-of-technical-documentation/
- Backstage: TechDocs: https://backstage.io/docs/features/techdocs/
- Backstage: Software Catalog: https://backstage.io/docs/features/software-catalog/
- AsyncAPI specification: https://www.asyncapi.com/docs/reference/specification/latest
- Confluent: Schema evolution and compatibility: https://docs.confluent.io/platform/current/schema-registry/fundamentals/schema-evolution.html
- Apicurio Registry: https://www.apicur.io/registry/

SECCIÓN 4.2 DE 4.8 · INTERMEDIO · 22 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Vistas y puntos de vista: ISO/IEC/IEEE 42010, 4+1 y Rozanski y Woods

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.1 Definir la arquitectura
- 1.5 Partes interesadas e inquietudes

Términos anteriores que utiliza: Parte interesada, Inquietud, Descripción de arquitectura

LA IDEA EN BREVE

Una casa nunca se describe con un solo dibujo. El comprador mira el plano de la planta, el electricista el plano de la instalación eléctrica, el fontanero el trazado de las tuberías y el técnico municipal de urbanismo el alzado de la fachada. Cada dibujo omite a propósito la mayoría de las cosas para poder responder con claridad a las preguntas de una persona. La arquitectura de software funciona igual. En lugar de un único diagrama enorme, los arquitectos dibujan varias vistas del mismo sistema, cada una para un grupo concreto de personas, y eligen solo las vistas que esas personas realmente necesitan.

LO QUE NECESITAS SABER

- Una vista representa un sistema desde la perspectiva de unas inquietudes concretas; un punto de vista es el conjunto reutilizable de convenciones (inquietudes que enmarca, partes interesadas, tipos de modelo) para construir un tipo de vista.
- En ISO/IEC/IEEE 42010:2022, las partes interesadas tienen inquietudes, los puntos de vista enmarcan inquietudes y rigen las vistas, las vistas abordan inquietudes y constan de componentes de vista regidos por tipos de modelo, y las correspondencias relacionan elementos entre vistas.
- El modelo 4+1 de Kruchten (IEEE Software, 1995) usa las vistas lógica, de procesos, de desarrollo y física, unidas y validadas por un pequeño conjunto de escenarios.
- Rozanski y Woods definen siete puntos de vista (contexto, funcional, información, concurrencia, desarrollo, despliegue, operacional) y perspectivas, como la seguridad y el rendimiento, que se aplican a todas las vistas en lugar de dibujarse como vistas.
- El SEI agrupa las vistas en las categorías de módulos, de componentes y conectores y de asignación; un conjunto de documentación completo incluye al menos una de cada una.

- Elige el conjunto mínimo de vistas relacionando las partes interesadas y las inquietudes con las vistas, combinando vistas cuando sea posible y documentando las zonas de riesgo con más profundidad que las rutinarias.

Un sistema, muchas vistas

Ningún diagrama por sí solo puede describir un sistema real. Un dibujo que muestra cómo se divide en módulos el código de la plataforma de reparto de comida no dice nada sobre en qué servidores se ejecutan; un dibujo de servidores y redes no dice nada sobre lo que ocurre, paso a paso, cuando un repartidor acepta un encargo. Si intentas poner todo esto en una sola página, obtienes el conocido póster de arquitectura: decenas de cajas, líneas que significan seis cosas distintas y nadie capaz de responder a una pregunta precisa a partir de él.

La idea que resuelve esto es antigua y sencilla. Describe el sistema mediante varias vistas, cada una de las cuales muestra un conjunto de estructuras para un conjunto de inquietudes, igual que un edificio se describe con un plano de planta, un alzado, un plano eléctrico y un plano de fontanería. Cada una es una descripción parcial; juntas describen el todo. Esta idea sustenta la norma internacional, el modelo 4+1 de Kruchten, el catálogo de puntos de vista de Rozanski y Woods, el enfoque Views and Beyond del SEI y, de forma ligera, el modelo C4 de la sección siguiente. Esta sección explica el vocabulario que comparten y cómo elegir las vistas que necesita un sistema concreto.

Recuerda de la Sección 1.5 que una parte interesada es cualquiera que tenga un interés en el sistema y que una inquietud es cualquier asunto importante para una parte interesada. Las vistas existen para responder a inquietudes. Una vista que no responde a la inquietud de nadie es decoración.

El modelo conceptual de ISO/IEC/IEEE 42010:2022

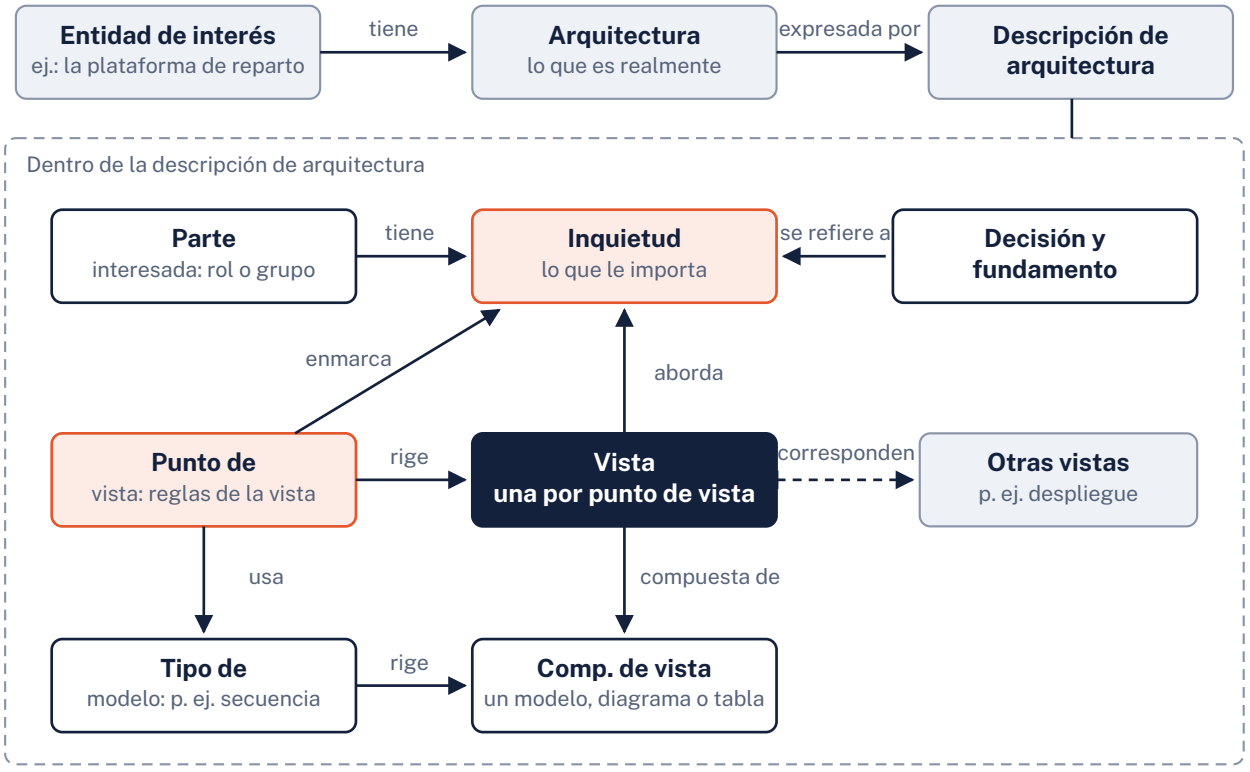
ISO/IEC/IEEE 42010 es la norma internacional para la descripción de arquitectura. Su edición actual se publicó en 2022 y sustituyó a la edición de 2011. No te dice cómo diseñar, qué notación usar ni qué vistas dibujar; define los conceptos que debe contener una descripción de arquitectura y cómo se relacionan, para que las descripciones producidas con métodos distintos puedan compararse y comprobarse. Su modelo conceptual es el esqueleto común de casi todos los enfoques de documentación de este capítulo.

El modelo parte de la entidad de interés: aquello cuya arquitectura se describe, que puede ser un sistema de software, un sistema de sistemas, una línea de productos o una empresa (la edición de 2011 lo llamaba sistema de interés). La entidad tiene una arquitectura, y una descripción de arquitectura la expresa. Dentro de la descripción, las **partes interesadas** tienen **inquietudes**. Un punto de vista es una especificación reutilizable de convenciones para

construir y usar un tipo de vista: nombra las inquietudes que enmarca, las partes interesadas típicas y los tipos de modelo (notaciones y convenciones de modelado) que deben usarse. Una **vista** aplica un punto de vista a una entidad concreta: aborda esas inquietudes para este sistema y está formada por componentes de vista, como diagramas, tablas y modelos, cada uno regido por un tipo de modelo (la edición de 2011 llamaba a los componentes de vista modelos de arquitectura).

En la práctica importan otros dos elementos. Las correspondencias registran relaciones entre elementos de la descripción, sobre todo, y es lo más útil, entre vistas: este contenedor de la vista de contenedores se ejecuta en aquel clúster de la vista de despliegue; este requisito lo aborda aquella decisión. Y las **decisiones de arquitectura y su justificación** forman parte de la descripción, vinculadas a las inquietudes que abordan. La edición de 2022 también define con más detalle dos conceptos que la edición de 2011 ya mencionaba, las perspectivas de las partes interesadas (formas de agrupar inquietudes) y los aspectos de arquitectura (características de la entidad que atraviesan las vistas), cambia el nombre de "marco de arquitectura" por el de marco de descripción de arquitectura y define lo que deben contener los marcos de descripción de arquitectura conformes, como la plantilla de documentación propia de una empresa.

El modelo conceptual de 42010, simplificado



Las inquietudes son el eje: los puntos de vista las enmarcan, las vistas las abordan y las decisiones se refieren a ellas.

SI ESTO ES NUEVO PARA TI

Si eres nuevo en esto: punto de vista frente a vista

Un punto de vista es como un tipo de mapa: "un mapa de carreteras" se define por sus convenciones (las carreteras en rojo, las autopistas más gruesas, una escala gráfica) y por sus usuarios (los conductores). Una vista es un mapa concreto dibujado con esas convenciones: el mapa de carreteras de Escocia. El punto de vista se define una vez, quizá para toda la empresa, y se dibuja una vista de él para cada sistema.

Un tipo de modelo es la técnica de dibujo que prescribe un punto de vista, como "diagrama de secuencia UML" o "diagrama de contenedores C4". La distinción permite que una organización diga: la vista de despliegue de cada sistema debe mostrar las regiones, las zonas de red y los servicios gestionados, dibujados como un diagrama de despliegue C4 con una tabla de entornos.

El modelo de vistas 4+1 de Kruchten

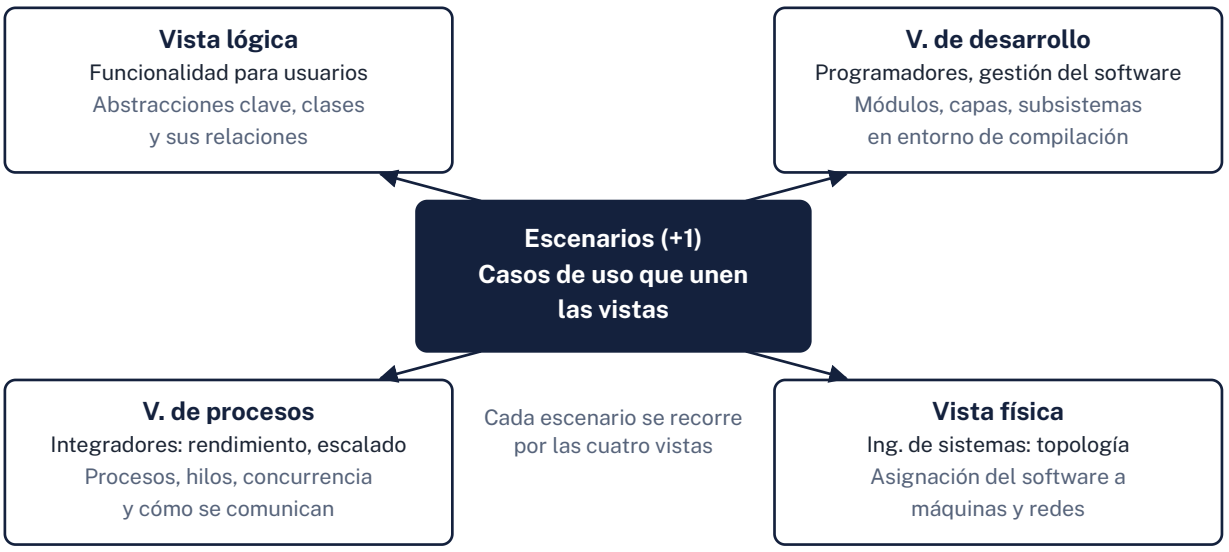
Philippe Kruchten, entonces en Rational Software, publicó *Architectural Blueprints: The "4+1" View Model of Software Architecture* en *IEEE Software* en noviembre de 1995. El modelo de vistas 4+1 fue el primer enfoque multivista ampliamente adoptado y pasó a formar parte del Rational Unified Process. El argumento de Kruchten era que los diagramas únicos de la época intentaban captar demasiado a la vez, y que las distintas partes interesadas (usuarios finales, desarrolladores, ingenieros de sistemas, jefes de proyecto) necesitan vistas distintas.

Las cuatro vistas son estas. La **vista lógica** describe la funcionalidad que el sistema proporciona a los usuarios finales, normalmente como las abstracciones clave y sus relaciones (en UML, diagramas de clases y de estados). La **vista de procesos** aborda las inquietudes de tiempo de ejecución: procesos e hilos, concurrencia, distribución, rendimiento, escalabilidad y tolerancia a fallos. La **vista de desarrollo** (también llamada vista de implementación) describe la organización del software en su entorno de desarrollo: módulos, subsistemas, capas y sus dependencias, que importan a los programadores y a los directivos que planifican el trabajo. La **vista física** (o vista de despliegue) asigna el software al hardware y a las redes. El "+1" son los **escenarios**: un pequeño conjunto de casos de uso importantes, cada uno recorrido a través de las cuatro vistas, que ilustran la arquitectura y a la vez validan que las vistas encajan entre sí.

Los escenarios son la parte que más se olvida y, posiblemente, la más valiosa. Una vista aislada puede parecer ordenada mientras oculta una laguna; recorrer un escenario real, como "el cliente hace un pedido", a través de las vistas lógica, de procesos, de desarrollo y física pone al

descubierto elementos que faltan y supuestos incoherentes. La misma idea reaparece en la vista de tiempo de ejecución de arc42, en los diagramas dinámicos de C4 y, en la Sección 4.8, en el consejo de explicar un sistema siguiendo una solicitud.

El modelo de vistas 4+1



Cuatro vistas para cuatro grupos de partes interesadas, unidas por un pequeño conjunto de escenarios.

Rozanski y Woods: puntos de vista y perspectivas

Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, de Nick Rozanski y Eóin Woods (2.ª edición, Addison-Wesley, 2012), ofrece un catálogo de siete puntos de vista orientado a grandes sistemas de información, junto con una segunda idea complementaria: las perspectivas.

Sus siete puntos de vista son: **contexto** (las relaciones del sistema con las personas, los sistemas y las entidades externas que lo rodean), **funcional** (los elementos funcionales de tiempo de ejecución, sus responsabilidades, interfaces e interacciones principales), **información** (cómo se almacenan, gestionan, poseen y distribuyen los datos), **concurrencia** (qué partes se ejecutan de forma concurrente y cómo se coordinan), **desarrollo** (lo que necesitan los desarrolladores, los testers y quienes compilan: la estructura de módulos, el procesamiento común, los estándares y la organización de las líneas de código), **despliegue** (el entorno de ejecución y cómo se asigna el software a él) y **operacional** (cómo se instala, se migra, se administra, se monitoriza y se da soporte al sistema en producción).

Una perspectiva es el nombre que dan a una inquietud de calidad que atraviesa todas las vistas, como la seguridad, el rendimiento y la escalabilidad, la disponibilidad y la resiliencia, o la evolución (su catálogo también incluye la accesibilidad, los recursos de desarrollo, la internacionalización, la ubicación, la regulación y la usabilidad). Una perspectiva no es otra vista que dibujar. Es un conjunto de preguntas, tácticas y comprobaciones que se aplican a todas las vistas: para la seguridad, examinas la vista de contexto en busca de actores externos y límites de confianza, la vista de información en busca de datos sensibles, la vista de despliegue en busca de zonas de red, y así sucesivamente. Esto capta algo que las vistas por separado no pueden: las cualidades surgen del sistema en su conjunto, así que hay que comprobarlas en todas partes (compara con los modelos de calidad del Capítulo 2).

El punto de vista operacional merece especial atención porque muchos enfoques lo infravaloran. Quién instala el sistema, cómo se migran los datos desde el antiguo, cómo se gestiona la configuración, qué monitorizan los operadores y cómo se les alerta: son preguntas arquitectónicas cuyas respuestas dan forma al sistema, y dejarlas para justo antes de la puesta en producción es una causa clásica de lanzamientos fallidos.

Las tres categorías del SEI: módulos, componentes y conectores, asignación

El Capítulo 1 presentó la agrupación de las estructuras del SEI en tres tipos (consulta la Sección 1.1). *Documenting Software Architectures: Views and Beyond* usa la misma agrupación para organizar las vistas, y es la forma más útil de comprobar que un conjunto de vistas está completo.

Las vistas de módulos muestran unidades de implementación (paquetes, bibliotecas, espacios de nombres) y relaciones como *es parte de*, *depende de* y *es un*. Los estilos típicos son las vistas de descomposición, de usos, de generalización, de capas y de modelo de datos. Las vistas de componentes y conectores (C&C) muestran los elementos presentes en tiempo de ejecución (servicios, clientes, bases de datos, colas) y los conectores mediante los que interactúan (llamadas HTTP, mensajes, datos compartidos); los estilos típicos incluyen cliente-servidor, publicación-suscripción, tuberías y filtros, datos compartidos y multinivel. Las vistas de asignación asignan los elementos de software a cosas que no son software: al hardware y a la infraestructura en la nube (despliegue), al sistema de archivos y a los artefactos de compilación (implementación e instalación) y a los equipos (asignación del trabajo).

Las tres categorías se corresponden de forma natural con los otros enfoques. La vista de desarrollo de Kruchten es una vista de módulos, su vista de procesos una vista C&C y su vista física una vista de asignación. Los puntos de vista funcional y de concurrencia de Rozanski y Woods son principalmente C&C, su punto de vista de desarrollo es de módulos, y sus puntos de vista de despliegue y operacional son de asignación. Los diagramas de contenedores y de

componentes de C4 de la sección siguiente son vistas C&C con cierto sabor de módulos, y los diagramas de despliegue de C4 son vistas de asignación. Cuando montes un conjunto de documentación, comprueba que contiene al menos una vista de cada categoría, porque cada una responde a preguntas que las otras no pueden responder.

Comparación de enfoques

Área de inquietud	4+1 (Kruchten)	Rozanski y Woods	Categoría del SEI	Diagrama C4
Límite del sistema, usuarios, sistemas externos	(implícito en los escenarios)	Contexto	C&C (contexto)	Contexto del sistema
Elementos de tiempo de ejecución e interacciones	Procesos (y lógica)	Funcional, concurrencia	Componentes y conectores	Contenedores, componentes, dinámico
Propiedad y flujo de los datos	(no separado)	Información	Módulos (modelo de datos) y C&C	(no separado)
Organización del código	Desarrollo	Desarrollo	Módulo	Componentes, código
Asignación a la infraestructura	Física	Despliegue	Asignación	Despliegue
Ejecución del sistema en producción	(no cubierto)	Operacional	Asignación (en parte)	(no cubierto)
Validación mediante casos de uso	Escenarios (+1)	(se aplica a todas las vistas)	Documentación del comportamiento	Dinámica

Relacionar las partes interesadas y las inquietudes con las vistas

La herramienta más práctica de la norma es una tabla sencilla: enumera las partes interesadas y sus principales inquietudes y, después, nombra la vista o las vistas que abordan cada inquietud. Los huecos (una inquietud que ninguna vista aborda) muestran dónde está incompleta la descripción; las vistas que nadie necesita son candidatas a eliminarse. La tabla siguiente es una primera aproximación para la plataforma de reparto de comida.

Partes interesadas, inquietudes y vistas de la plataforma de reparto de comida

Parte interesada	Inquietudes principales	Vistas que las atienden
Responsables de producto y patrocinadores	Alcance, socios externos, qué se compra frente a qué se construye, costo	Vista de contexto; visión general de una página con los objetivos de calidad y los principales riesgos
Desarrolladores de cada equipo	Dónde va el código, interfaces entre servicios, convenciones	Vista de contenedores; vista de componentes para los servicios complejos; conceptos transversales; decisiones
Operaciones y SRE	Dónde se ejecuta cada cosa, failover, capacidad, dependencias, recuperación	Vista de despliegue por entorno; vista operacional (monitorización, runbooks, copias de seguridad y restauración)
Equipo de seguridad	Límites de confianza, autenticación, datos de tarjetas, secretos	Vistas de contexto y de despliegue anotadas con zonas de confianza; concepto de seguridad (una perspectiva aplicada a todas las vistas)
Delegado de protección de datos	Datos personales de clientes y repartidores: por dónde fluyen y cuánto tiempo se conservan	Vista de información con un diagrama de flujo de datos y una tabla de conservación
Equipo de integración del proveedor de pagos	Qué llamadas se hacen, webhooks, idempotencia, versiones	Especificación de la interfaz y un diagrama de secuencia del flujo de pago
Evaluador de pagos con tarjeta (auditor)	Alcance del entorno de datos de tarjetas, controles, historial de cambios	Vista de despliegue que muestra el límite de los datos de tarjetas; bitácora de decisiones; historial de versiones

Elegir el conjunto mínimo de vistas

La tentación, una vez que sabes que existen muchos puntos de vista, es producirlos todos. Resístela. Cada vista tiene que mantenerse coherente con el sistema y con las demás vistas, así que cada una añade un costo de mantenimiento. El objetivo es el conjunto más pequeño de vistas que responda a las inquietudes importantes de las partes interesadas importantes, siguiendo la regla de la Sección 4.1 de escribir para el lector. Views and Beyond describe un procedimiento parecido: construir una tabla de partes interesadas y vistas, combinar vistas cuando un diagrama pueda servir a varias necesidades y, después, priorizar.

Un procedimiento para elegir las vistas

1. **1. Enumera las partes interesadas y las inquietudes**
   Usa el análisis de partes interesadas del Capítulo 1. Para cada inquietud, anota lo importante que es y cuánto está en juego si se malinterpreta.

2. **2. Relaciona las inquietudes con vistas candidatas**
   Usa un catálogo (4+1, Rozanski y Woods o las categorías del SEI) como lista de comprobación para no olvidar áreas enteras, normalmente la información y las operaciones.

3. **3. Combina cuando puedas**
   Un diagrama de contenedores anotado con tecnologías y protocolos suele servir a la vez a desarrolladores y a operaciones; un diagrama de despliegue con zonas de confianza sirve a operaciones y a seguridad. Combinar reduce el número de artefactos que hay que mantener.

4. **4. Comprueba la cobertura de las tres categorías**
   Asegúrate de que el conjunto incluye al menos una vista de módulos, una de componentes y conectores y una de asignación, además de unos pocos escenarios que las recorran.

5. **5. Decide la profundidad de cada vista**
   No todas las vistas necesitan el mismo detalle. Documenta en profundidad las partes arriesgadas o complejas y brevemente las rutinarias.

6. **6. Registra las correspondencias**
   Indica cómo se relacionan los elementos de una vista con los de otra (contenedor con nodo de despliegue, módulo con contenedor) para que los lectores puedan moverse entre vistas. Las discrepancias entre vistas son donde se esconden muchos errores de diseño.

EJEMPLO PRÁCTICO

Ejemplo práctico: el conjunto mínimo de vistas para la plataforma de reparto de comida

Aplicar el procedimiento a la plataforma descrita en la Sección 4.1 (cinco equipos, pagos con tarjeta a través de un proveedor externo, una próxima evaluación de seguridad de pagos) da el siguiente conjunto.

Vista de contexto	Un diagrama de contexto del sistema C4: clientes, personal de restaurantes, repartidores, agentes de soporte, el proveedor de pagos, el servicio de mapas, los servicios de notificaciones push. Sirve a todo el mundo; es lo primero que lee un recién llegado.
Vista C&C	Un diagrama de contenedores C4 para toda la plataforma, más diagramas de componentes solo para los servicios de pedidos y de despacho de repartidores, que son los más complejos y los que más cambian.
Vista de información	Una tabla de las principales entidades de datos, qué servicio es propietario de cada una, por dónde fluyen los datos personales y los plazos de conservación; un diagrama de flujo de datos para los datos de clientes. Se añadió porque, de otro modo, las inquietudes del delegado de protección de datos quedaban sin abordar.
Vista de asignación	Un diagrama de despliegue C4 para producción, con el límite de los datos de tarjetas y las zonas de red marcados, y una breve tabla de diferencias en el entorno de preproducción.
Vista de módulos	No se dibuja a mano. La estructura interna de módulos de cada servicio se genera a partir del código y se comprueba con pruebas de arquitectura.
Escenarios	Cuatro diagramas de secuencia: hacer un pedido, el restaurante rechaza el pedido, asignación de repartidor, reembolso. Cada uno se contrasta con las vistas de contenedores y de despliegue.
Perspectivas	La seguridad y la disponibilidad se aplican como listas de comprobación a todas las vistas durante la revisión, en lugar de dibujarse como vistas separadas.

Siete artefactos, la mayoría de ellos diagramas únicos, responden a las inquietudes de los siete grupos de partes interesadas de la tabla anterior.

La vista de información se añadió solo porque la tabla de partes interesadas y vistas puso de manifiesto un hueco. Sin la tabla se habría pasado por alto, algo habitual.

La vista de módulos se genera, no se dibuja, porque puede recuperarse a partir del código y cambia a menudo (Sección 4.1).

PARA PROFUNDIZAR

Para profundizar: documentar bien una vista

Views and Beyond recomienda una estructura estándar para cada vista, que además es una buena lista de comprobación para las revisiones: una **presentación principal** (normalmente el diagrama, que muestra los elementos y las relaciones), un **catálogo de elementos** (qué es cada elemento, sus responsabilidades, interfaces y propiedades), un **diagrama de contexto** que muestra lo que queda fuera de la parte representada, una **guía de variabilidad** (lo que puede variar, como los componentes opcionales o la configuración) y la **justificación** del diseño mostrado.

También insiste en la documentación que va *más allá* de las vistas, que da nombre al libro: una hoja de ruta que explica cómo está organizada la documentación, una visión general del sistema, la correspondencia entre vistas, la justificación de las decisiones que atraviesan varias vistas y un directorio o glosario. La mayoría de los antipatrones de la Sección 4.1 proceden de saltarse estas partes.

ATENCIÓN

Errores habituales con las vistas

- Mezclar categorías en un mismo diagrama, por ejemplo dibujar paquetes de código, servicios en ejecución y servidores como cajas en el mismo lienzo, con líneas que significan cosas distintas.
- Dibujar vistas que nadie ha pedido porque un marco las enumera, y omitir la única vista que necesita el equipo de operaciones.
- Dejar que las vistas se contradigan: la vista de contenedores muestra seis servicios, pero la vista de despliegue muestra cinco. Las correspondencias y las revisiones lo detectan.
- Tratar los atributos de calidad como una vista más en lugar de como una perspectiva que hay que comprobar en todas las vistas.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **¿Qué diferencia hay entre un punto de vista y una vista?**

Un punto de vista es una especificación reutilizable de convenciones para un tipo de vista (inquietudes que enmarca, partes interesadas, tipos de modelo). Una vista aplica un punto de vista a un sistema concreto y aborda esas inquietudes para él.

▼ **¿Qué papel desempeñan los escenarios en el modelo 4+1?**

Son un pequeño conjunto de casos de uso importantes que se recorren a través de las cuatro vistas, que ilustran la arquitectura y validan que las vistas son coherentes y completas.

▼ **¿En qué se diferencia una perspectiva de Rozanski y Woods de un punto de vista?**

Un punto de vista produce una vista. Una perspectiva es una inquietud de calidad transversal, como la seguridad, que se aplica como preguntas y tácticas a todas las vistas en lugar de dibujarse como una vista propia.

▼ **¿A qué categoría del SEI pertenece un diagrama de despliegue y por qué?**

A la de asignación, porque asigna elementos de software a cosas que no son software, en este caso el hardware y la infraestructura en la nube.

▼ **Inténtalo: En una plataforma de venta de entradas importan tres lectores: los desarrolladores nuevos, los operadores del día de la venta y un evaluador de seguridad de pagos. ¿Qué conjunto mínimo de vistas producirías y cómo lo justifica la inquietud de cada lector?**

Una vista de contexto para todos. Una vista de contenedores compartida por desarrolladores y operadores, de modo que un diagrama sirve a dos inquietudes. Una vista de despliegue u operativa para los operadores, que muestre las zonas, el escalado y la gestión de fallos. Una perspectiva de seguridad aplicada a esas vistas, que muestre al evaluador los límites de confianza y los flujos de datos de tarjetas. Añade una vista de ejecución del recorrido de compra, porque es la zona de riesgo. Omite las vistas detalladas de las partes rutinarias. Menos vistas se mantienen actualizadas.

FUENTES

- ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description (ISO): https://www.iso.org/standard/74393.html
- ISO/IEC/IEEE 42010 summary (arc42 quality model): https://quality.arc42.org/standards/iso-42010
- Philippe Kruchten, Architectural Blueprints: The 4+1 View Model of Software Architecture (arXiv copy of the 1995 IEEE Software paper): https://arxiv.org/abs/2006.04975
- Viewpoints catalogue, Rozanski and Woods: https://www.viewpoints-and-perspectives.info/home/viewpoints/

- Documenting Software Architectures: Views and Beyond, 2nd edition (SEI): https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/

SECCIÓN 4.3 DE 4.8 · INTERMEDIO · 24 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

El modelo C4: contexto, contenedores, componentes y código

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.4 Comunicación en red
- 0.7 Servidores y la nube
- 4.2 Vistas y puntos de vista

Términos anteriores que utiliza: API (interfaz de programación de aplicaciones), HTTP, Punto de vista

LA IDEA EN BREVE

Piensa en un mapa en línea. Con la vista alejada, ves un país entero y las autopistas entre ciudades. Acerca la vista y ves los distritos de una ciudad, luego las calles de un distrito y después un solo edificio. Cada nivel de zoom es útil para personas distintas, y siempre sabes en qué nivel estás mirando. El modelo C4 hace lo mismo con el software: una imagen muestra el sistema completo y quién lo usa, la siguiente muestra las apps y bases de datos que contiene, la siguiente muestra las partes dentro de una app y la última, que rara vez se necesita, muestra el propio código.

LO QUE NECESITAS SABER

- El modelo C4, creado por Simon Brown, describe el software con una jerarquía de cuatro diagramas (contexto del sistema, contenedores, componentes y código) construida sobre las abstracciones persona, sistema de software, contenedor y componente.
- Un contenedor C4 es una unidad que se ejecuta o se despliega por separado y que ejecuta código o almacena datos, como una aplicación web, una app móvil, un servicio, un esquema de base de datos o un topic de mensajes; no es lo mismo que un contenedor Docker.
- Los diagramas de contexto del sistema y de contenedores se recomiendan para todos los equipos; los de componentes, solo donde aporten valor, y los de código, rara vez y, preferiblemente, generados a partir del código.
- Los diagramas complementarios son el panorama de sistemas (todos los sistemas de una organización), el dinámico (la colaboración en tiempo de ejecución para un escenario) y el de despliegue (los contenedores asignados a la infraestructura, uno por entorno).

- Todo diagrama necesita un título con su tipo y su alcance, y una leyenda; todo elemento, un nombre, un tipo, una descripción y (en contenedores y componentes) una tecnología; toda relación, una única dirección, una etiqueta concreta y, entre contenedores, un protocolo.
- Las herramientas de modelado como Structurizr generan muchos diagramas coherentes a partir de un solo modelo, mientras que las herramientas de dibujo producen imágenes independientes que se van distanciando entre sí.

Mapas de tu código

El modelo C4 es un enfoque para diagramar la arquitectura de software creado por Simon Brown. Surgió de la formación en arquitectura de software que impartía desde aproximadamente 2006: según el sitio de C4, sus raíces se remontan a algún momento entre 2006 y 2009, los tipos de diagrama recibieron nombre a principios de 2010 y el nombre "C4" se usó por primera vez a principios de 2011. Brown había observado que los equipos a los que se pedía dibujar su arquitectura producían diagramas difíciles de entender: formas y colores sin explicar, cajas que podían ser cualquier cosa, desde una clase hasta un centro de datos, líneas sin etiqueta y varios niveles de detalle mezclados en una misma página. UML, que podría haber resuelto el problema, había caído en gran medida en desuso en el trabajo diario.

Su respuesta no fue una notación nueva, sino un pequeño conjunto de *abstracciones* y una jerarquía de diagramas construida sobre ellas, que él compara con los niveles de zoom de un mapa en línea. Empiezas con el país entero, te acercas a una ciudad, luego a un distrito y después a una calle. Cada nivel muestra una cantidad distinta de detalle con un propósito distinto, y siempre sabes en qué nivel estás mirando. C4 es deliberadamente independiente de la notación y de la herramienta: puedes dibujarlo en una pizarra, en una herramienta de dibujo o con una herramienta de modelado, siempre que sigas unas pocas reglas.

C4 se ha convertido en uno de los enfoques más usados para los diagramas de arquitectura porque es fácil de aprender, encaja de forma natural con el pensamiento de múltiples vistas de la Sección 4.2 y produce diagramas que pueden seguir tanto los desarrolladores como los lectores no técnicos. En términos de 42010, cada tipo de diagrama C4 es un punto de vista con su propio tipo de modelo; C4 simplemente mantiene pequeño el conjunto.

Las abstracciones: personas, sistemas de software, contenedores, componentes

C4 se basa en cuatro abstracciones, más el código. Una **persona** es un usuario humano del sistema: un cliente, un repartidor, un agente de soporte. Un sistema de software es el nivel más alto de abstracción, algo que aporta valor a sus usuarios y que normalmente pertenece a un

equipo u organización, que lo construye; la plataforma de reparto de comida es uno, y también lo son el proveedor de pagos y el servicio de mapas de los que depende.

Un contenedor es algo que tiene que estar en ejecución para que el sistema funcione: en palabras de Brown, esencialmente un límite de ejecución alrededor de un código que se ejecuta o de unos datos que se almacenan. Son ejemplos una aplicación web del lado del servidor, una aplicación de una sola página que se ejecuta en un navegador, una app móvil, una aplicación de escritorio, una función serverless, un esquema de base de datos, un almacén de archivos o un topic de mensajes. Cada uno se puede desplegar o ejecutar por separado, y los contenedores se comunican entre sí a través de los límites de proceso, normalmente por una red.

Un componente es una agrupación de funcionalidad relacionada detrás de una interfaz bien definida, que vive *dentro* de un contenedor y se ejecuta en el mismo proceso que los demás componentes de ese contenedor. En un servicio Java puede ser un conjunto de clases detrás de una interfaz; en un servicio Node.js, un módulo. Los componentes no se pueden desplegar por separado; la unidad de despliegue es el contenedor. Por último, el **código** son las clases, funciones y tablas que implementan un componente.

ATENCIÓN

Un contenedor no es un contenedor Docker

El malentendido más común sobre C4 es leer "contenedor" como "contenedor Docker". El término de C4 es anterior a la popularidad de Docker y significa cualquier unidad que se ejecuta o se despliega por separado: una app móvil en un teléfono, una aplicación JavaScript en un navegador y un esquema de base de datos son todos contenedores, aunque ninguno se ejecute en Docker. A la inversa, un contenedor C4 puede desplegarse como veinte contenedores Docker detrás de un balanceador de carga; cuántas copias se ejecutan, y dónde, corresponde a un diagrama de despliegue, no a un diagrama de contenedores.

Si la palabra causa confusión en tu organización, la mayoría de los equipos la mantienen y la explican en la leyenda o en el glosario. El sitio de C4 reconoce el choque, señala que a veces ambos coinciden de forma útil y dice que los equipos pueden cambiar la terminología si lo necesitan; si lo haces, cámbiala en todas partes.

Los cuatro diagramas principales

Nivel	Alcance	Muestra	Público principal	¿Recomendado?
1. Contexto del sistema	Un sistema de software	El sistema como una sola caja, las personas que lo usan y los demás sistemas de software con los que interactúa	Todo el mundo, técnicos y no técnicos, dentro y fuera del equipo	Sí, para todos los equipos
2. Contenedores	Un sistema de software	Los contenedores (apps, servicios, almacenes de datos, topics) que contiene, sus responsabilidades, sus tecnologías y cómo se comunican	Arquitectos, desarrolladores y personal de operaciones y soporte	Sí, para todos los equipos
3. Componentes	Un contenedor	Los componentes que contiene, sus responsabilidades y tecnologías, y sus relaciones	Arquitectos y desarrolladores	Solo donde aporte valor; genéralo si hay que mantenerlo a largo plazo
4. Código	Un componente	Clases, interfaces, funciones o tablas, normalmente como diagrama de clases UML o diagrama entidad-relación	Desarrolladores	Rara vez; generarlo bajo demanda a partir del código para las partes más complejas

SI ESTO ES NUEVO PARA TI

Paso a paso: ¿sistema, contenedor o componente?

El punto en el que más gente tropieza aquí es decidir qué abstracción de C4 es una parte de un sistema. Los nombres se parecen, así que recorre la plataforma de reparto de comida de fuera hacia dentro y pregúntate, de cada parte, si se puede desplegar y ejecutar por sí sola.

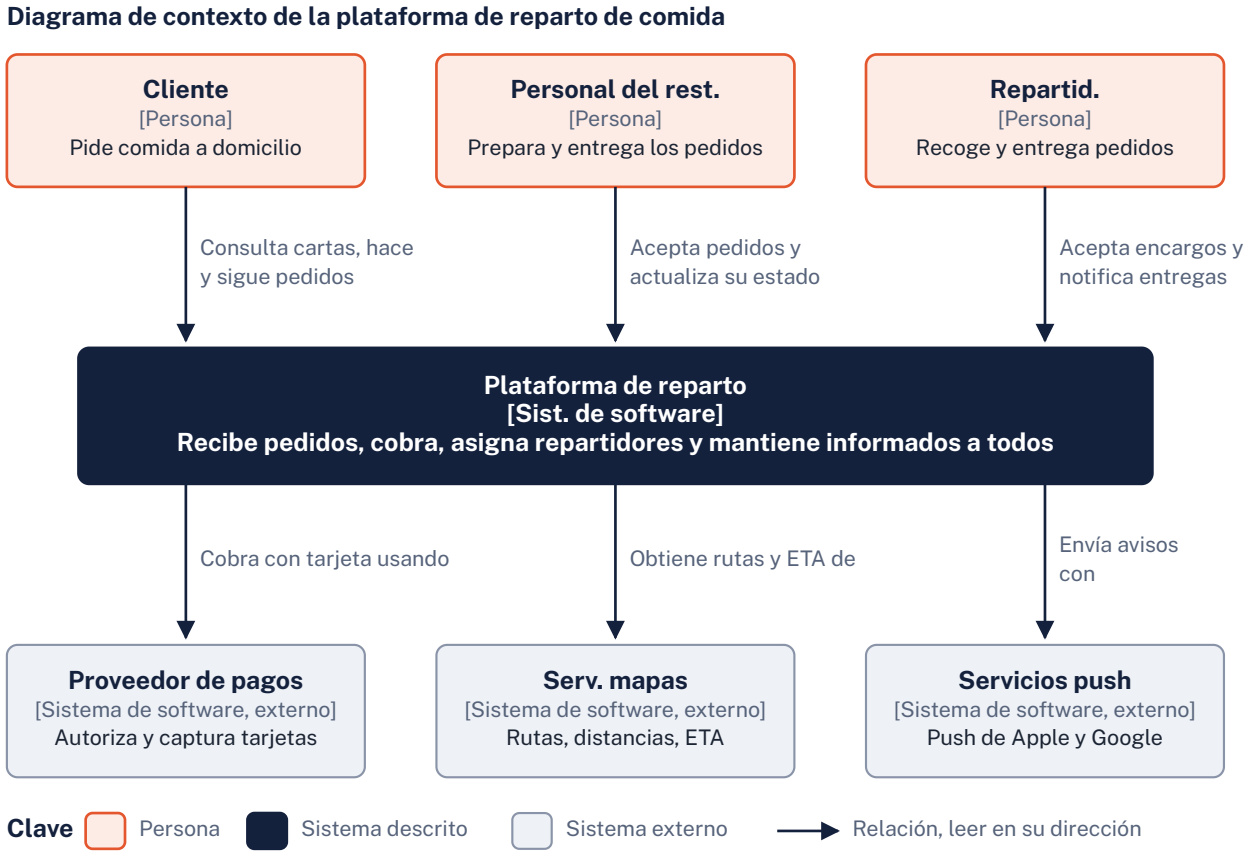
- Empieza por arriba. La plataforma de reparto de comida en su conjunto es un sistema de software: aporta valor a los clientes, al personal de los restaurantes y a los repartidores. El proveedor de pagos es otro sistema de software, que construye y gestiona otra organización.
- Ahora mira dentro de la plataforma. El servicio de pedidos se ejecuta como un programa propio y se comunica con las demás partes por la red, así que es un contenedor. También lo son la app del cliente en un teléfono y la app del restaurante que se ejecuta en el navegador de una tableta.
- Un contenedor no tiene por qué ejecutar tu propio código. La base de datos de pedidos almacena datos, y la plataforma no puede funcionar si no está en ejecución, así que también es un contenedor. Los diagramas de la sección tratan de la misma forma el topic de eventos de pedidos, que guarda mensajes.
- Acércate al servicio de pedidos. Su flujo de pedidos, que aplica las reglas de los pedidos, y su cliente de pagos, que llama al proveedor de pagos, son componentes. Se ejecutan en el mismo proceso que el resto del servicio y se despliegan con él; el cliente de pagos no se puede desplegar por sí solo.
- **La lección:** una parte que se puede desplegar y ejecutar por sí misma es un contenedor; una parte dentro de uno, que comparte su proceso, es un componente; el conjunto que aporta valor a sus usuarios, que normalmente pertenece a un equipo u organización, es un sistema de software. Decide esto antes de dibujar, y cada caja quedará en el nivel de zoom correcto.

Nivel 1: el diagrama de contexto del sistema

El diagrama de contexto del sistema es el punto de partida. Coloca el sistema que estás describiendo en el centro, como una sola caja, y lo rodea de las personas que lo usan y de los sistemas de software de los que depende o que dependen de él. Oculta deliberadamente todo el detalle interno, incluidas las tecnologías, los protocolos y los servicios internos. Responde a las preguntas que cualquier recién llegado, responsable de producto o revisor de seguridad hace primero: ¿qué es este sistema, quién lo usa y con qué se conecta?

Como oculta lo interno, el diagrama de contexto es también el diagrama adecuado para las conversaciones con partes interesadas no técnicas y para delimitar el alcance: ¿dónde termina nuestra responsabilidad? En el ejemplo de reparto de comida de abajo muestra de inmediato que los pagos con tarjeta, el cálculo de rutas y el envío de notificaciones push se contratan fuera, lo cual es arquitectónicamente significativo (cada uno es una dependencia externa con su propia disponibilidad, su costo y su contrato) e invisible en la mayoría de las vistas a nivel de código.

Plataforma de reparto de comida: contexto del sistema



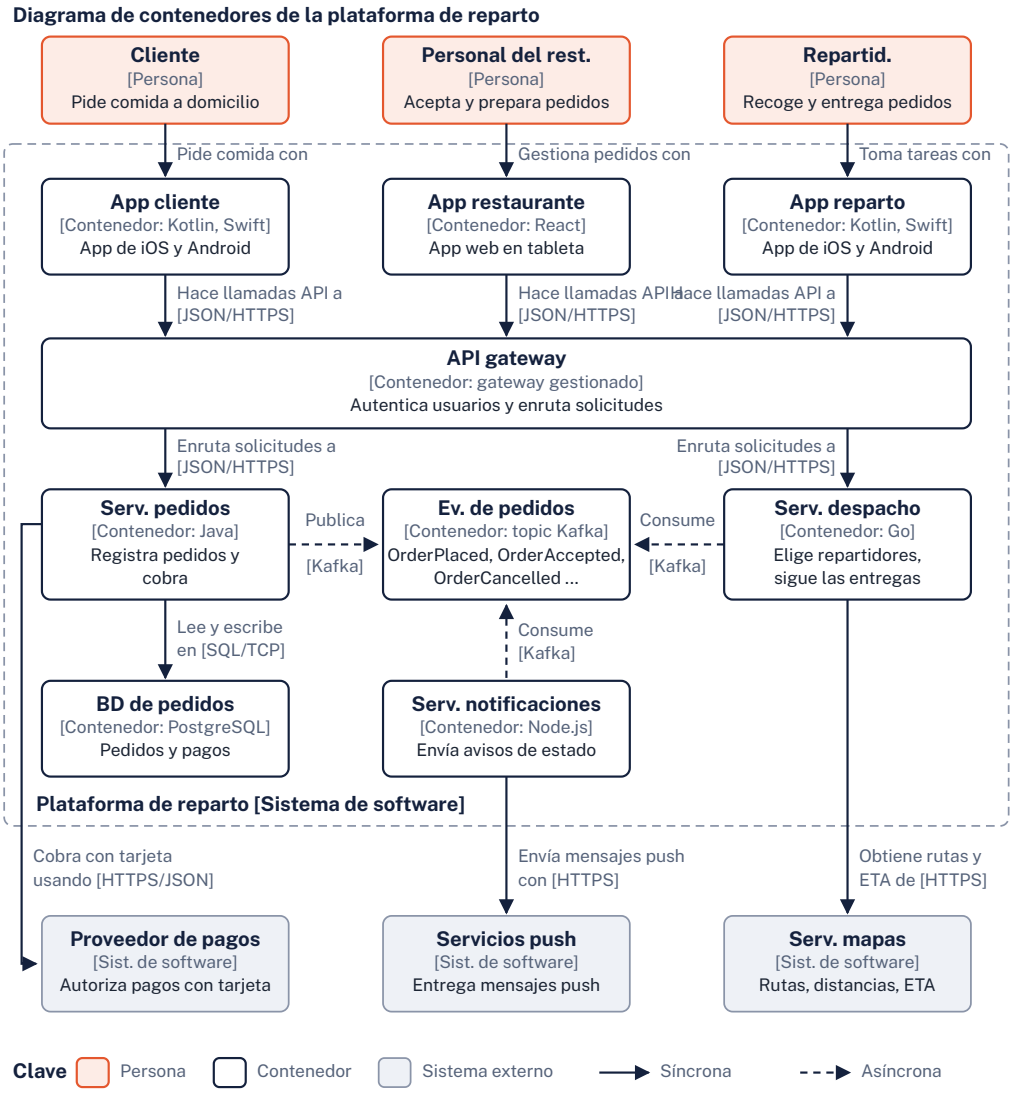
Nivel 1: la plataforma como una sola caja, con sus tres tipos de usuarios y tres sistemas externos; cada elemento tiene un nombre, un tipo y una descripción, y cada flecha tiene una etiqueta.

Nivel 2: el diagrama de contenedores

El diagrama de contenedores se acerca a la caja del sistema y muestra los contenedores que hay dentro: las aplicaciones y los almacenes de datos que forman el sistema, de qué es responsable cada uno, qué tecnología usa cada uno y cómo se comunican. Para la mayoría de los equipos es el diagrama más valioso que tienen. Muestra las principales decisiones estructurales (cuántas unidades desplegables hay, cuáles se comunican de forma síncrona y cuáles mediante mensajería, cuáles son dueñas de datos) y es lo bastante concreto para los desarrolladores y el personal de operaciones, sin dejar de caber en una página.

Fíjate en dos decisiones en el diagrama de abajo. El topic de mensajes de los eventos de pedidos se dibuja como un contenedor por derecho propio, porque es un almacén en el que los productores escriben y del que los consumidores leen; la guía de C4 ofrece este estilo explícito o, como alternativa, omitir el topic y escribir "a través del topic de eventos de pedidos" en las flechas. Dibujarlo muestra con claridad el acoplamiento a través del topic. Y las relaciones asíncronas usan una línea discontinua, que explica la leyenda. El diagrama de contenedores no dice nada sobre cuántas copias de cada servicio se ejecutan, en qué región o detrás de qué balanceador de carga; eso corresponde a un diagrama de despliegue.

Plataforma de reparto de comida: contenedores



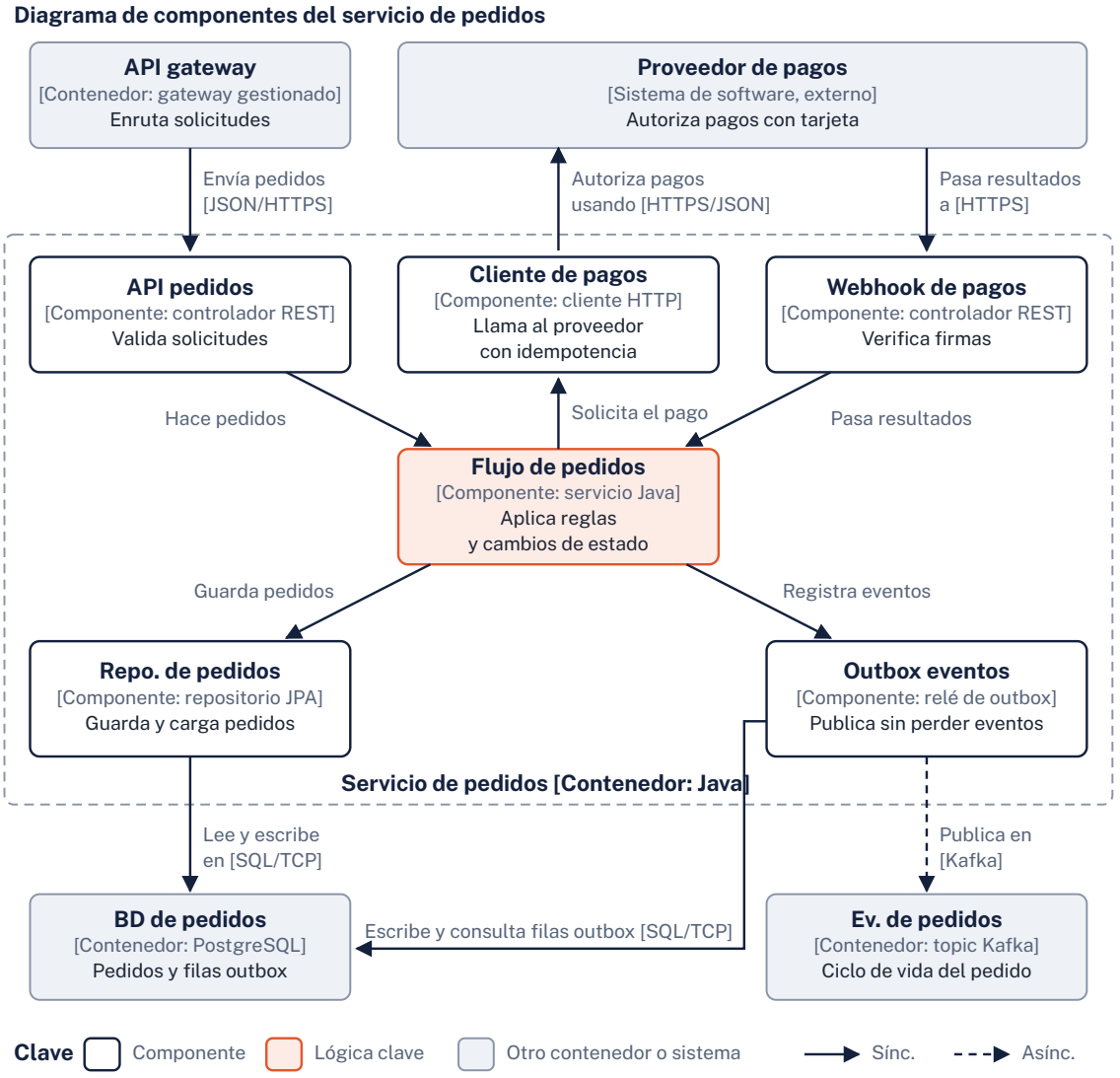
Nivel 2: las apps, los servicios y los almacenes de datos dentro del límite de la plataforma, con tecnologías en cada elemento y protocolos en cada flecha.

Nivel 3: el diagrama de componentes

El diagrama de componentes se acerca a un contenedor y muestra los componentes que hay dentro. Está dirigido a los arquitectos y desarrolladores que trabajan en ese contenedor, y es donde se hace visible la estructura interna de un servicio: qué parte atiende las solicitudes entrantes, dónde viven las reglas de negocio, qué parte habla con la base de datos y cómo salen los eventos del servicio.

La guía de C4 es dibujar diagramas de componentes solo donde aporten valor, normalmente para contenedores complejos, arriesgados o que modifican muchas personas, y generarlos automáticamente si se van a mantener durante mucho tiempo, porque la estructura de componentes cambia tan a menudo como el código. En la plataforma de reparto de comida, el servicio de pedidos cumple esos requisitos: maneja dinero, habla con un proveedor externo y debe publicar eventos de forma fiable. El diagrama de abajo muestra una estructura típica, que incluye un componente de bandeja de salida (outbox) que escribe los eventos en la base de datos en la misma transacción que el pedido y después los retransmite a Kafka (el patrón de bandeja de salida se trata en el Capítulo 6).

Servicio de pedidos: componentes



Nivel 3: dentro de un contenedor, con los contenedores y sistemas vecinos fuera del límite discontinuo para que cada relación tenga sus dos extremos.

Nivel 4: el código y los diagramas complementarios

El nivel de código se acerca a un componente y muestra sus clases, interfaces o tablas, normalmente como diagrama de clases UML o diagrama entidad-relación. La guía de C4 lo considera opcional y, en general, no vale la pena dibujarlo a mano: la mayoría de los entornos de desarrollo pueden generarlo bajo demanda, y una copia dibujada a mano queda desactualizada en cuanto alguien refactoriza. Dibújalo solo para las partes más importantes o intrincadas, como un modelo de dominio en el corazón del negocio (la Sección 4.4 muestra un diagrama de clases así).

Tres **diagramas complementarios** cubren los huecos que deja la jerarquía. Un **diagrama de panorama de sistemas** (system landscape) es un diagrama de contexto sin un único sistema en el foco: muestra todos los sistemas de software y personas de una organización o dominio, lo que resulta útil para los arquitectos empresariales y de plataforma. Un diagrama dinámico muestra cómo colaboran los elementos en tiempo de ejecución en un escenario, como interacciones numeradas sobre un diagrama de contenedores o de componentes (o como un diagrama de secuencia UML); es el equivalente en C4 de los escenarios de 4+1. Un diagrama de despliegue muestra cómo se asignan los contenedores a la infraestructura en un entorno: nodos de despliegue como regiones de la nube, clústeres de Kubernetes, máquinas virtuales, servicios de bases de datos gestionados y dispositivos, con el número de instancias. C4 recomienda un diagrama de despliegue por cada entorno que importe, normalmente producción y quizá preproducción (staging).

Reglas de notación que hacen legibles los diagramas

C4 no prescribe formas ni colores, pero sí prescribe lo que debe decir todo diagrama. Estas reglas, resumidas de la guía de notación de C4, son las que hacen que los diagramas anteriores se puedan leer sin una reunión para explicarlos.

Lista de comprobación: un diagrama C4 que se explica solo

- [ ] **Un título que indique el tipo y el alcance**
  Por ejemplo, "Diagrama de contenedores de la plataforma de reparto de comida", para que los lectores sepan de inmediato el nivel de zoom.

- [ ] **Una leyenda (clave)**
  Explica cada forma, color, estilo de borde, estilo de línea y punta de flecha que uses, aunque la notación parezca obvia. Los lectores de otros equipos no compartirán tus convenciones.

- [ ] **Cada elemento tiene un nombre, un tipo y una descripción**
  El tipo entre corchetes (Persona, Sistema de software, Contenedor, Componente) elimina la ambigüedad; una frase breve sobre su responsabilidad dice para qué sirve el elemento.

- [ ] **Tecnología en cada contenedor y componente**
  "Contenedor: Java", "Contenedor: PostgreSQL". Las elecciones tecnológicas son arquitectura y los lectores las necesitan.

- [ ] **Cada relación tiene una sola dirección y una etiqueta**
  Usa una flecha por relación, etiquetada con lo que hace en la dirección de la flecha ("Publica eventos de pedidos en"), nunca solo "usa" ni una flecha de doble punta.

- [ ] **Protocolos en las relaciones entre contenedores**
  JSON/HTTPS, SQL/TCP, Kafka. Que una relación sea síncrona o asíncrona es uno de los datos más importantes del diagrama.

- [ ] **Acrónimos explicados**
  Escríbelos completos o defínelos en la leyenda o en el glosario.

- [ ] **Un nivel de abstracción por diagrama**
  Nada de clases en un diagrama de contenedores, nada de servidores en un diagrama de componentes.

- [ ] **El color como complemento, no como única señal**
  Los diagramas se imprimen en blanco y negro y los leen personas daltónicas; usa también el estilo de línea y las etiquetas, y mantén el significado de los colores coherente entre diagramas.

EJEMPLO PRÁCTICO

Ejemplo práctico: mejorar un diagrama típico de un equipo

Un equipo de la empresa de reparto de comida comparte esta descripción del diagrama de su pizarra: cinco cajas con las etiquetas "App", "Backend", "Orders", "DB" y "Stripe"; líneas entre ellas sin puntas de flecha; la caja Orders es verde y las demás azules; no hay título ni leyenda. Aplicando las reglas de C4 paso a paso:

Título y nivel	Decide qué es el diagrama. Mezcla una app móvil, un servicio y una base de datos, así que es un diagrama de contenedores. Ponle el título "Diagrama de contenedores de la plataforma de reparto de comida".
Nombre, tipo, descripción	"App" pasa a ser "App del cliente [Contenedor: Kotlin, Swift] App de iOS y Android para pedir comida". "Backend" resulta ser el API gateway; "DB" pasa a ser "Base de datos de pedidos [Contenedor: PostgreSQL]".
Sistemas externos	"Stripe" es el nombre de un producto de un sistema externo. Muéstralo como "Proveedor de pagos [Sistema de software]" fuera del límite del sistema, con el nombre del proveedor en la descripción si eso importa a los lectores.
Relaciones	Sustituye cada línea simple por una flecha etiquetada: "Llama a [JSON/HTTPS]", "Lee y escribe en [SQL/TCP]", "Autoriza pagos con tarjeta usando [HTTPS]". El equipo descubre que una línea significaba dos cosas distintas (una llamada en una dirección y un webhook en la otra), así que se convierte en dos flechas.
Elementos que faltan	Al etiquetar las flechas se descubre que los eventos de pedidos llegan al servicio de despacho a través de un topic de Kafka que no estaba dibujado. Añade el topic como contenedor, con flechas asíncronas discontinuas.
Leyenda y color	La caja verde resulta significar "es de nuestro equipo". Añádelo a una leyenda, o sustituye el color por un límite discontinuo con el nombre del equipo.
Resultado	El diagrama ha ganado un título, una leyenda, tres elementos y el doble de flechas, y aun así necesita menos explicación, porque cada pregunta que haría un lector tiene respuesta en la página.

La mayor parte del valor vino de etiquetar las flechas: las líneas sin etiqueta ocultan los datos arquitectónicos más importantes (dirección, protocolo, síncrono o asíncrono).

El diagrama mejorado reveló una dependencia (el topic) que el propio equipo había olvidado, un efecto secundario típico de aplicar las reglas.

ATENCIÓN

Errores comunes con C4

- Dibujar cada microservicio como un sistema de software aparte cuando todos pertenecen a un mismo equipo. La guía de C4 dice que los servicios de un solo equipo son contenedores dentro del sistema de ese equipo; los servicios de equipos distintos pueden modelarse cada uno como un sistema de software.
- Mostrar un bus de mensajes compartido o una malla de servicios como un gran contenedor por el que pasa todo, lo que oculta quién depende realmente de quién. Modela los topics o colas individuales, o escríbelos en las flechas.
- Dibujar a mano diagramas de componentes para todos los contenedores. Se desactualizan rápido; dibújalos de forma selectiva o genéralos.
- Poner detalles de despliegue (réplicas, balanceadores de carga, zonas de disponibilidad) en el diagrama de contenedores en lugar de en un diagrama de despliegue.
- Flechas etiquetadas solo con "usa" o "datos", o flechas que apuntan en los dos sentidos.
- Omitir a las personas en el diagrama de contexto, lo que elimina la razón por la que existe el sistema.

Herramientas: dibujo frente a modelado

Los diagramas C4 pueden hacerse con cualquier herramienta de dibujo, y muchos equipos empiezan con una pizarra o una aplicación general de diagramas. El sitio de C4 distingue dos familias de herramientas. Las **herramientas de diagramación** dibujan cajas y líneas; son flexibles y conocidas, pero los diagramas son imágenes independientes, así que cambiar el nombre de un elemento obliga a editar todos los diagramas que lo muestran, y nada comprueba que los diagramas coincidan. Las **herramientas de modelado** mantienen un único modelo de personas, sistemas, contenedores, componentes y relaciones, y generan cada diagrama como una vista de ese modelo; el cambio de nombre se hace una sola vez, los diagramas siguen siendo coherentes y el modelo se puede consultar y validar.

Structurizr, creada por Simon Brown, es la herramienta de modelado de referencia para C4. Su Structurizr DSL, basado en texto, describe el modelo y las vistas en un archivo que vive en el control de versiones junto al código. Otras opciones son C4-PlantUML (formas C4 para PlantUML), el soporte de diagramas C4 de Mermaid y diversas herramientas de modelado comerciales; la Sección 4.7, Diagramas como código, muestra ejemplos. La elección importa menos que el hábito: guarda la fuente de los diagramas donde trabaja el equipo y actualízala en el mismo pull request que el cambio que describe.

Preguntas frecuentes

¿Sustituye C4 a UML?

No exactamente. C4 define abstracciones y tipos de diagrama, no una notación, y no dice nada sobre muchas cosas que UML puede expresar con precisión, como las máquinas de estados o las secuencias detalladas de mensajes. Muchos equipos usan C4 para la estructura y toman prestados de UML los diagramas de secuencia, de estados y de despliegue donde hace falta precisión (Sección 4.4).

¿Cómo encajan las funciones serverless en C4?

Cada función, o cada grupo de funciones que se despliegan juntas, es un contenedor: una unidad desplegable por separado que ejecuta código. Los almacenes de datos y las colas compartidos que usan también son contenedores.

Nuestro sistema es un único monolito desplegable. ¿El diagrama de contenedores no sirve para nada?

No. Un monolito normalmente sigue teniendo un cliente web o móvil, la aplicación de servidor, una base de datos y quizá un almacén de archivos y una tarea programada, y cada uno es un contenedor. El diagrama de componentes del monolito pasa entonces a ser la vista más interesante de sus módulos internos.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Qué es un contenedor en el modelo C4? Da dos ejemplos que no tengan que ver con Docker.

Una unidad que se ejecuta o se despliega por separado y que ejecuta código o almacena datos. Ejemplos: una app móvil en un teléfono, una aplicación de una sola página en un navegador, un esquema de base de datos, una función serverless.

▼ ¿Qué dos diagramas C4 recomienda la guía de C4 para todos los equipos?

El diagrama de contexto del sistema y el diagrama de contenedores.

▼ ¿Dónde deben mostrarse el número de réplicas, las regiones y los balanceadores de carga?

En un diagrama de despliegue, uno por entorno, no en el diagrama de contenedores.

▼ **¿Por qué insiste C4 en flechas de una sola dirección y con etiqueta?**

Las líneas sin etiqueta o de doble sentido ocultan qué es la relación (llamada, flujo de datos, evento), en qué dirección va y qué protocolo usa, que están entre los datos arquitectónicos más importantes del diagrama.

▼ **Inténtalo: Un diagrama muestra las cajas "Web", "API", "Kafka", "Postgres" y "Payment" unidas por líneas sin flechas, sin título ni leyenda. Una caja es roja. Aplica las reglas de C4: ¿qué está mal y qué cambiarías?**

No tiene un título que indique tipo y alcance, ni una leyenda, así que nadie sabe qué significa el rojo. Los cuadros necesitan nombre, tipo, descripción y, en el caso de los contenedores, tecnología. Payment es un sistema de software externo y debe verse diferente. Las líneas necesitan una sola dirección, una etiqueta específica y un protocolo, como "publica OrderPlaced [Kafka]". Este es un diagrama de contenedores, así que llámalo así. Conserva el rojo solo si la leyenda lo explica; de lo contrario, quítalo.

FUENTES

- The C4 model for visualising software architecture (Simon Brown): https://c4model.com/
- C4 model: Container (not Docker!): https://c4model.com/abstractions/container
- C4 model: Notation: https://c4model.com/diagrams/notation
- C4 model: System context diagram: https://c4model.com/diagrams/system-context
- Structurizr DSL documentation: https://docs.structurizr.com/dsl

SECCIÓN 4.4 DE 4.8 · INTERMEDIO · 23 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Los diagramas UML que los arquitectos todavía usan

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.6 API e integración
- 4.2 Vistas y puntos de vista
- 4.3 El modelo C4

Términos anteriores que utiliza: Interfaz, Llamada síncrona, Vista de asignación, Inversión de dependencias

LA IDEA EN BREVE

Músicos de todo el mundo pueden leer la misma partitura, porque las notas y los símbolos significan lo mismo en todas partes. UML es un conjunto compartido de símbolos parecido para dibujar software: una flecha concreta significa "espera una respuesta", otra significa "envía y sigue adelante", un rombo relleno significa "está formado por". No necesitas conocerlo entero. Los arquitectos usan sobre todo un puñado de sus dibujos: uno que muestra quién habla con quién y en qué orden, otro que muestra las etapas por las que pasa algo, otro que muestra dónde se ejecuta todo y unos pocos más, cada uno usado solo cuando su precisión ayuda de verdad.

LO QUE NECESITAS SABER

- UML es un lenguaje de modelado de propósito general estandarizado por el OMG; la versión actual es UML 2.5.1 (diciembre de 2017), que define catorce tipos de diagrama en las familias de estructura y de comportamiento.
- Fowler distingue entre UML como boceto (comunicación selectiva e informal), como plano (modelos completos y precisos) y como lenguaje de programación (modelos ejecutables); los arquitectos sobre todo hacen bocetos.
- Los diagramas de componentes muestran interfaces proporcionadas (bola) y requeridas (enchufe); los diagramas de despliegue muestran nodos, artefactos y vías de comunicación.
- Los diagramas de secuencia muestran líneas de vida y activaciones; una punta de flecha rellena es una llamada síncrona, una punta abierta sobre una línea continua es un mensaje asíncrono y una línea discontinua es una respuesta; fragmentos como alt, opt y loop añaden flujo de control.

- Los diagramas de máquina de estados describen ciclos de vida con estados y transiciones etiquetadas como evento [guarda] / acción; los diagramas de actividades describen flujos de trabajo con decisiones, bifurcaciones, uniones y calles (swimlanes).
- Los diagramas de clases conviene reservarlos para modelos de dominio pequeños con asociaciones, multiplicidades y composición; los diagramas de clases completos deben generarse, no dibujarse a mano.
- La precisión merece la pena en las interfaces entre organizaciones, en el trabajo regulado y crítico para la seguridad (safety), en los ciclos de vida que comparten varios equipos y en el desarrollo distribuido.

Qué es UML y en qué punto está hoy

El Lenguaje Unificado de Modelado (UML) es un lenguaje gráfico de propósito general para describir la estructura y el comportamiento del software. Surgió de la fusión, en Rational Software a mediados de la década de 1990, de los métodos de Grady Booch, James Rumbaugh e Ivar Jacobson, y lo adoptó como estándar el Object Management Group (OMG), un consorcio de estándares del sector, en 1997. UML 2.0 (2005) fue una revisión importante. La versión actual es UML 2.5.1, publicada por el OMG en diciembre de 2017; la anterior, UML 2.4.1, también la publicó ISO como ISO/IEC 19505 (2012). La especificación define catorce tipos de diagrama, divididos en diagramas de estructura (como los de clases, componentes y despliegue) y diagramas de comportamiento (como los de actividades, máquina de estados y secuencia).

La reputación de UML ha cambiado considerablemente. A finales de la década de 1990 y principios de la de 2000 se promovió como la base del desarrollo dirigido por modelos, en el que modelos detallados se transformarían en código. Esa ambición fracasó en gran medida fuera de los dominios especializados, y muchos equipos asocian UML con procesos pesados y diagramas que nadie leía. Cuando Marian Petre entrevistó a 50 desarrolladores de software profesionales para un estudio presentado en ICSE 2013, 35 dijeron que no usaban UML en absoluto y la mayoría de los demás lo usaban de forma selectiva e informal; la mayoría de los equipos dibujan cajas y líneas informales en lugar de UML estricto. Aun así, un puñado de tipos de diagrama UML siguen siendo las mejores herramientas disponibles para ciertas preguntas, sobre todo las relativas al comportamiento a lo largo del tiempo, y su notación se entiende ampliamente. Un arquitecto no necesita los catorce tipos de diagrama; necesita quizá seis, usados de forma deliberada.

Esta sección trata esos seis, cada uno con un pequeño ejemplo de la plataforma de reparto de comida que se usa a lo largo de este capítulo. Da por sabido el vocabulario de C4 de la Sección 4.3; los dos se combinan bien, con C4 para la visión general de la estructura y UML donde la precisión sobre interfaces, interacciones o estados compensa su costo.

Boceto, plano o lenguaje de programación

UML Distilled de Martin Fowler (3.ª edición, Addison-Wesley, 2003) y sus ensayos del mismo año distinguen tres formas de usar UML, una clasificación a la que Steve Mellor llegó de manera independiente. Como **boceto**, UML se usa de forma informal y selectiva para comunicar ideas: en una pizarra antes de escribir código (boceto hacia delante) o para explicar parte de un sistema existente (boceto inverso). Solo aparecen los elementos importantes, la notación se sigue con flexibilidad y el diagrama a menudo se tira. El propio resumen de Fowler es “la exhaustividad es enemiga de la comprensibilidad”.

Como **plano**, UML se usa para producir modelos completos y precisos que otra persona implementa, o que se generan a partir del código existente para documentarlo por completo; lo que importa es la completitud, y normalmente hacen falta herramientas. Como **lenguaje de programación**, el modelo UML es lo bastante detallado para compilarse en código ejecutable, el objetivo de la arquitectura dirigida por modelos (MDA). Este modo sobrevive en algunos nichos embebidos y críticos para la seguridad (safety), a menudo mediante perfiles y herramientas especializados, pero no se ha generalizado.

Para los arquitectos, el modo boceto es el predeterminado y el modo plano, la excepción. La pregunta útil para cada diagrama es: ¿quién lo leerá y qué decisión o comprensión respalda? Un diagrama de secuencia esbozado para acordar cómo se gestionan los fallos de pago es valioso aunque ignore la mitad de la notación. Un modelo de clases completo de un servicio de 200 clases, mantenido a mano, normalmente no lo es.

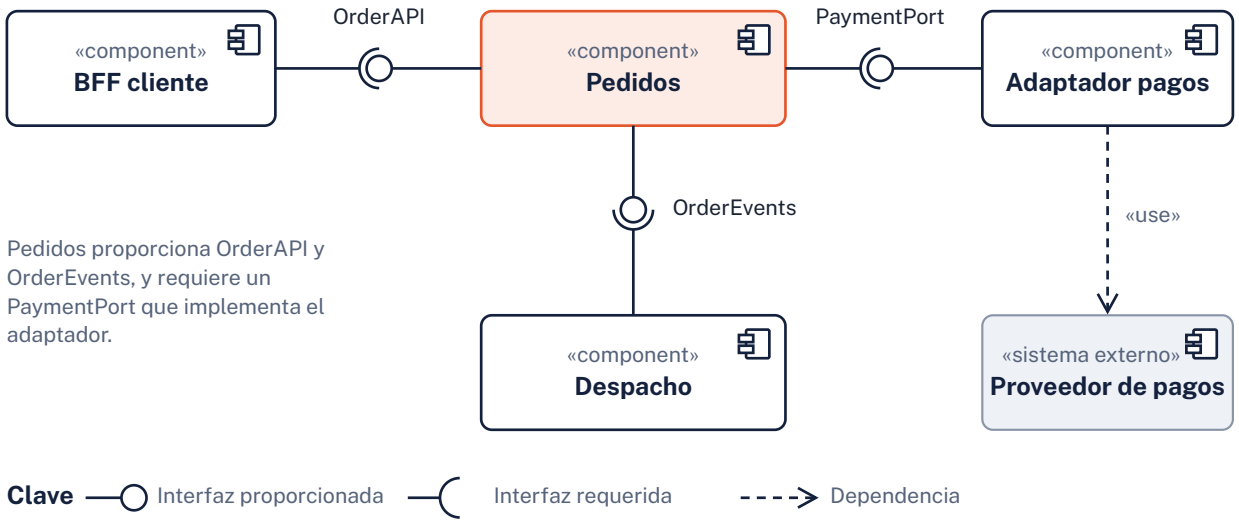
Diagramas de componentes: las partes y sus interfaces

Un diagrama de componentes UML muestra las partes modulares de un sistema y, sobre todo, sus **interfaces**. Su notación característica es la de "bola y enchufe": un pequeño círculo sobre un palo (una interfaz proporcionada, que el componente ofrece) y un semicírculo (una interfaz requerida, que necesita que otro le proporcione). Cuando una bola encaja en un enchufe, los dos componentes quedan conectados a través de esa interfaz. Las dependencias son flechas discontinuas, que pueden llevar opcionalmente un estereotipo como «use».

Esto es exactamente lo que les importa a los arquitectos cuando quieren mostrar la dirección de las dependencias y la sustituibilidad. En el ejemplo, el componente de pedidos no depende directamente del proveedor de pagos; requiere una interfaz `PaymentPort`, que implementa un adaptador de pagos aparte. Es la inversión de dependencias y el enfoque de puertos y adaptadores del Capítulo 3 hechos visibles: cambiar de proveedor de pagos significa escribir un adaptador nuevo, no modificar el componente de pedidos. Un diagrama de componentes C4 puede mostrar las mismas partes, pero no distingue las interfaces proporcionadas de las requeridas.

Diagrama de componentes

Diagrama de componentes: pedidos y sus interfaces



Las bolas son interfaces que un componente proporciona; los enchufes son interfaces que requiere. Pedidos depende de un puerto, no del proveedor. (BFF: backend for frontend, un componente del lado del servidor dedicado a una app cliente concreta.)

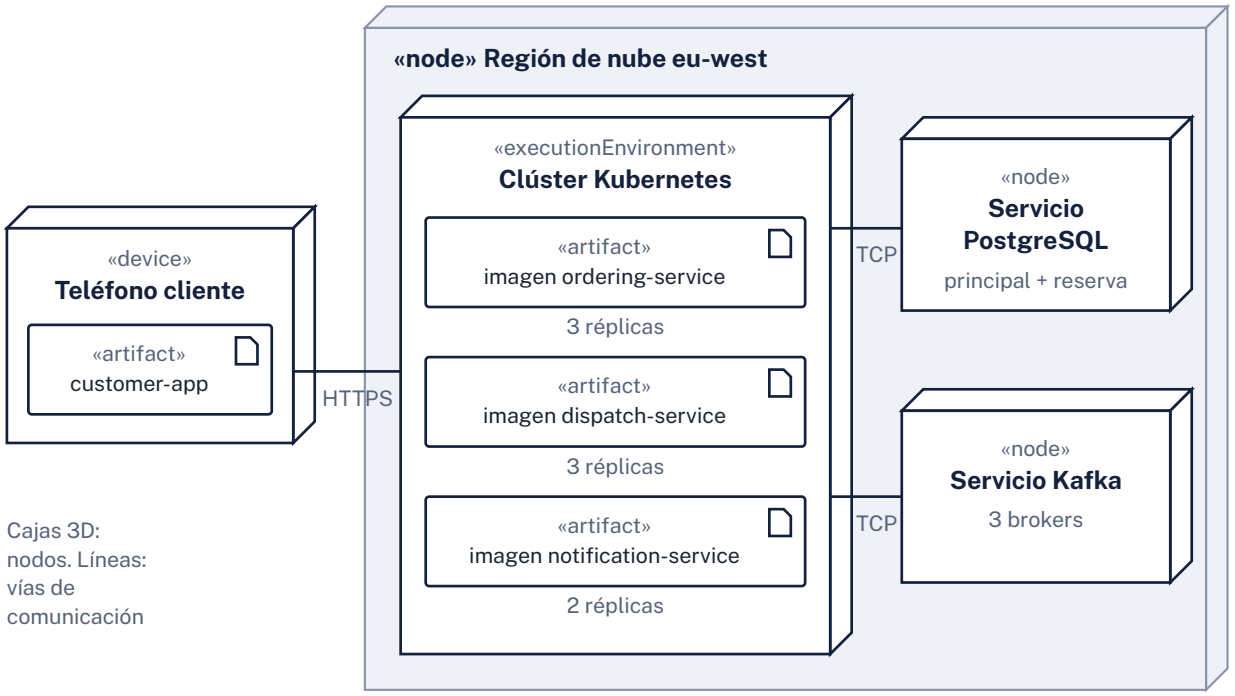
Diagramas de despliegue: nodos y artefactos

Un diagrama de despliegue UML muestra la infraestructura física o virtual y lo que se ejecuta en ella. Sus elementos son los **nodos**, dibujados como cajas tridimensionales, que son recursos de cómputo: un «device», como un teléfono o un servidor, o un «executionEnvironment», como un sistema operativo, un entorno de ejecución de contenedores o un clúster de Kubernetes; los nodos pueden anidarse. Los **artefactos**, dibujados como rectángulos con un icono de documento y el estereotipo «artifact», son los archivos físicos que se despliegan: un JAR, una imagen de contenedor, un paquete de app móvil, un archivo de configuración. Las líneas entre nodos son **vías de comunicación**, etiquetadas con protocolos.

Los diagramas de despliegue son vistas de asignación en los términos del SEI (Sección 4.2). Responden a las preguntas del equipo de operaciones: dónde se ejecuta cada parte, cuántas copias hay, en qué región, qué rutas de red existen. El diagrama de despliegue de C4 cubre el mismo terreno con una notación menos formal; cualquiera de los dos sirve, siempre que haya uno por cada entorno importante y muestre las réplicas, los servicios gestionados y los límites de red.

Diagrama de despliegue

Diagrama de despliegue: producción



Los nodos son donde se ejecutan las cosas; los artefactos son lo que se despliega; las líneas son vías de comunicación.

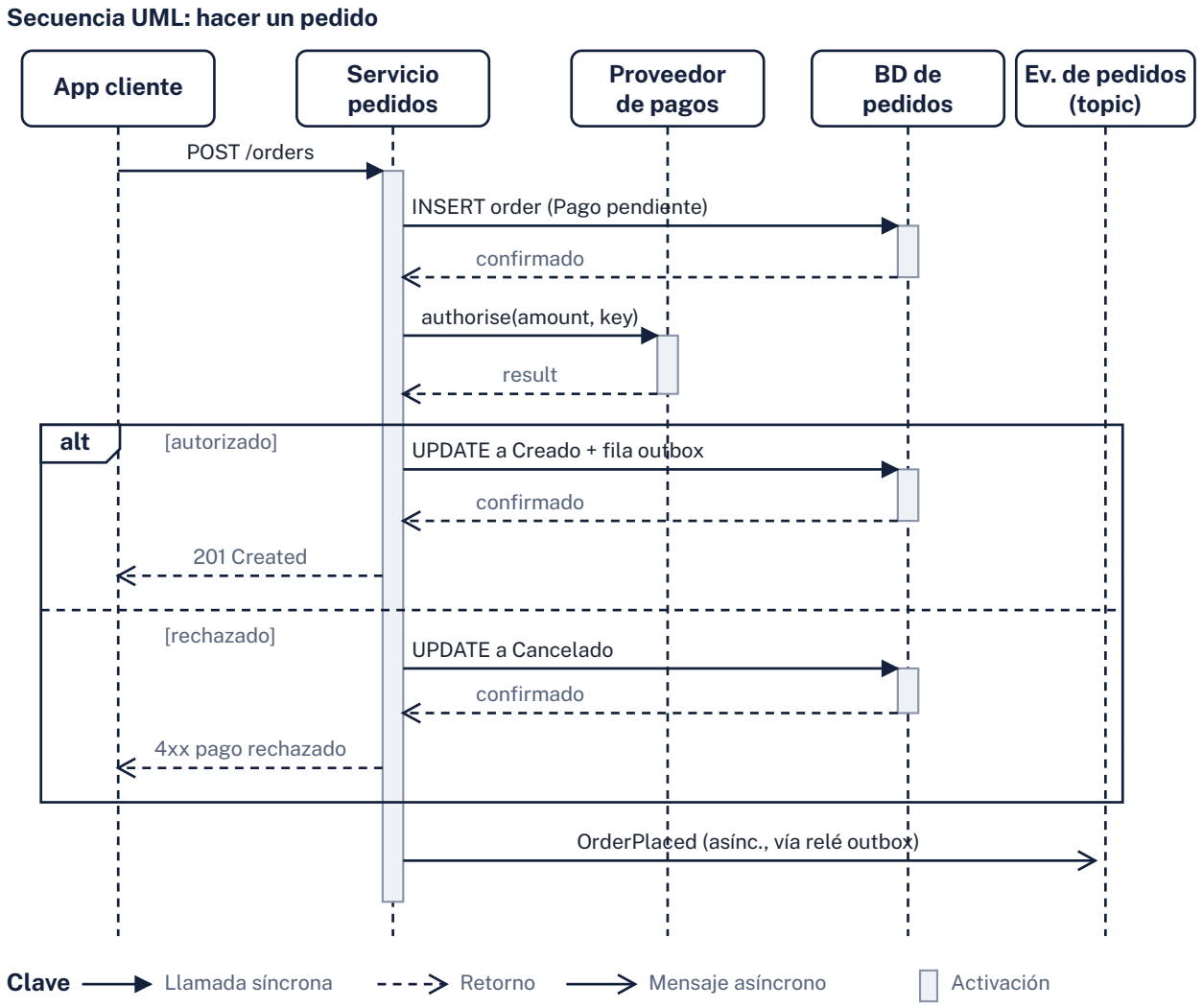
Diagramas de secuencia: quién llama a quién y en qué orden

Si los arquitectos se quedan con un solo diagrama UML, suele ser el diagrama de secuencia. Muestra cómo interactúan los participantes a lo largo del tiempo en un escenario. Los participantes aparecen en la parte superior, cada uno con una línea de vida vertical discontinua; el tiempo avanza hacia abajo. Un rectángulo estrecho sobre una línea de vida, la **activación** (especificación de ejecución), muestra cuándo ese participante está ocupado atendiendo un mensaje.

Las puntas de flecha tienen significado. Un **mensaje síncrono**, en el que el emisor espera una respuesta, tiene una línea continua y una punta de flecha rellena. Una **respuesta** (retorno) tiene una línea discontinua y una punta abierta. Un **mensaje asíncrono**, en el que el emisor sigue adelante sin esperar, tiene una línea continua y una punta abierta. Acertar con esto importa para la arquitectura: que el servicio de pedidos espere al proveedor de pagos o que publique y siga adelante determina su latencia, sus modos de fallo y su acoplamiento (consulta la Sección 0.6). El orden de los pasos también importa: el diagrama de abajo guarda el pedido antes de llamar al proveedor, así que una caída entre ambos nunca puede dejar a un cliente con un cobro sin registro del pedido, y la clave de idempotencia hace que reintentar una autorización sea seguro.

Los **fragmentos combinados** añaden flujo de control: una caja con la etiqueta `alt` y secciones con guarda separadas por una línea discontinua muestra alternativas (if/else), `opt` muestra una parte opcional, `loop` una repetición y `par` secciones en paralelo. Úsalos con moderación; un diagrama de secuencia con bucles y alternativas anidados es código mal dibujado. Dibuja un diagrama por cada escenario importante, incluidos los escenarios de fallo, que es donde ocurre la mayor parte del razonamiento arquitectónico.

Diagrama de secuencia



Hacer un pedido con seguridad: guardar primero el pedido, autorizar la tarjeta con una clave de idempotencia y luego actualizar el pedido. Un fragmento alt muestra los dos resultados posibles y un evento asíncrono cierra el flujo.

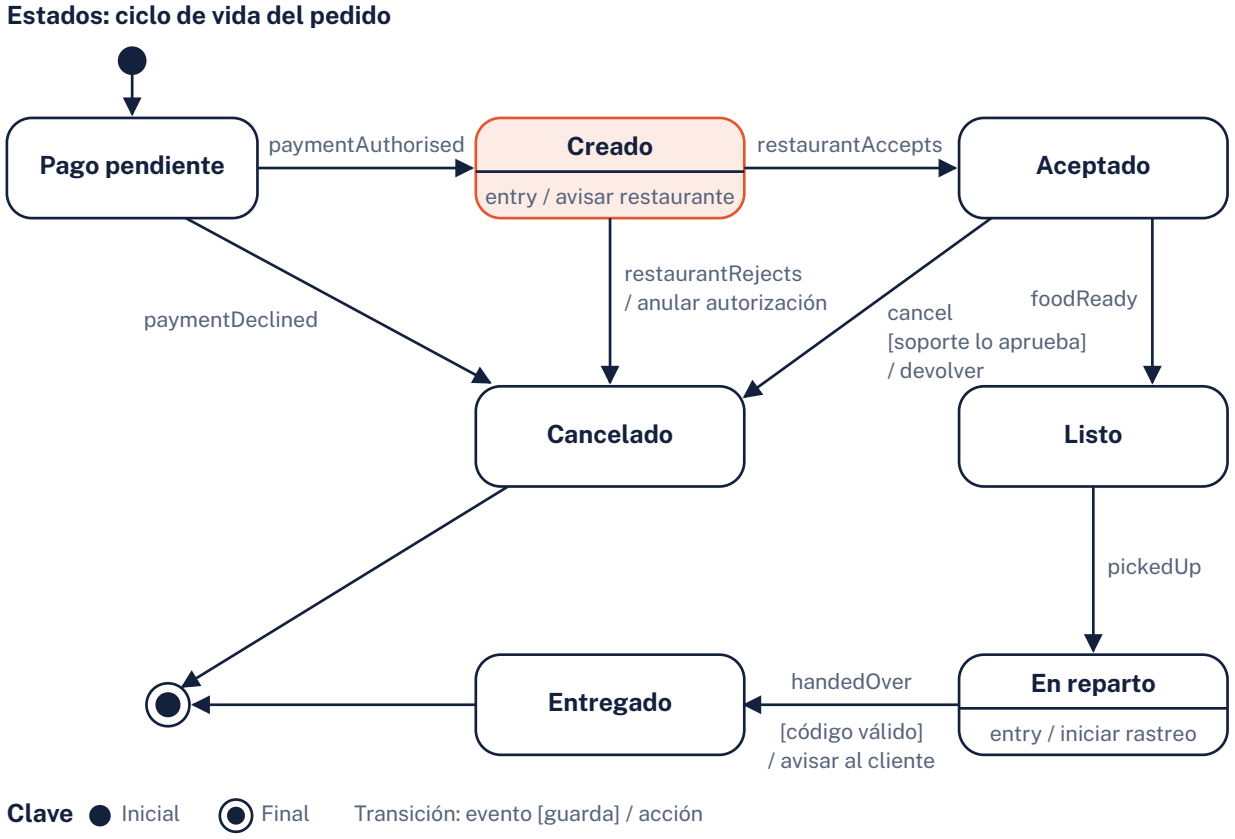
Diagramas de máquina de estados: ciclos de vida y reglas

Muchos objetos de negocio tienen un ciclo de vida: un pedido, un pago, una suscripción, un siniestro de seguros. Un diagrama de máquina de estados lo describe con precisión. Los **estados** son rectángulos redondeados; un círculo relleno marca el pseudoestado **inicial** y una

diana el estado **final**. Las **transiciones** son flechas con la etiqueta `event [guard] / action`: el evento que desencadena la transición, una condición de guarda opcional que debe cumplirse y una acción opcional que se realiza por el camino. Los estados pueden tener comportamientos internos como `entry / notify restaurant`.

Las máquinas de estados son arquitectónicas más a menudo de lo que se cree. El ciclo de vida del pedido de abajo lo comparten varios servicios: pedidos es su dueño, despacho y notificaciones reaccionan a sus transiciones y el personal de soporte puede desencadenar algunas de ellas. Acordar los estados y las transiciones válidas, en un solo diagrama, evita toda una clase de bugs en los que un servicio cree que un pedido está cancelado mientras otro envía a un repartidor. El diagrama también saca a la luz preguntas que el texto oculta: ¿se puede cancelar un pedido cuando ya está en reparto? (Aquí, no: no hay ninguna transición desde ese estado a Cancelado.) ¿Un reembolso es una acción de una transición o un ciclo de vida aparte? El dinero también sigue a los estados: la tarjeta solo se autoriza cuando se hace el pedido y se captura cuando el restaurante acepta, así que un rechazo anula la autorización y solo una cancelación posterior necesita un reembolso.

Diagrama de máquina de estados



El ciclo de vida del pedido: todas las transiciones válidas, con eventos, guardas y acciones, en una sola página.

SI ESTO ES NUEVO PARA TI

Paso a paso: leer la etiqueta de una transición

La idea más difícil de esta sección es el pequeño lenguaje escrito en cada flecha de un diagrama de máquina de estados: evento [guarda] / acción. Sigue un pedido a lo largo del ciclo de vida del pedido de la sección y lee cada etiqueta por turno.

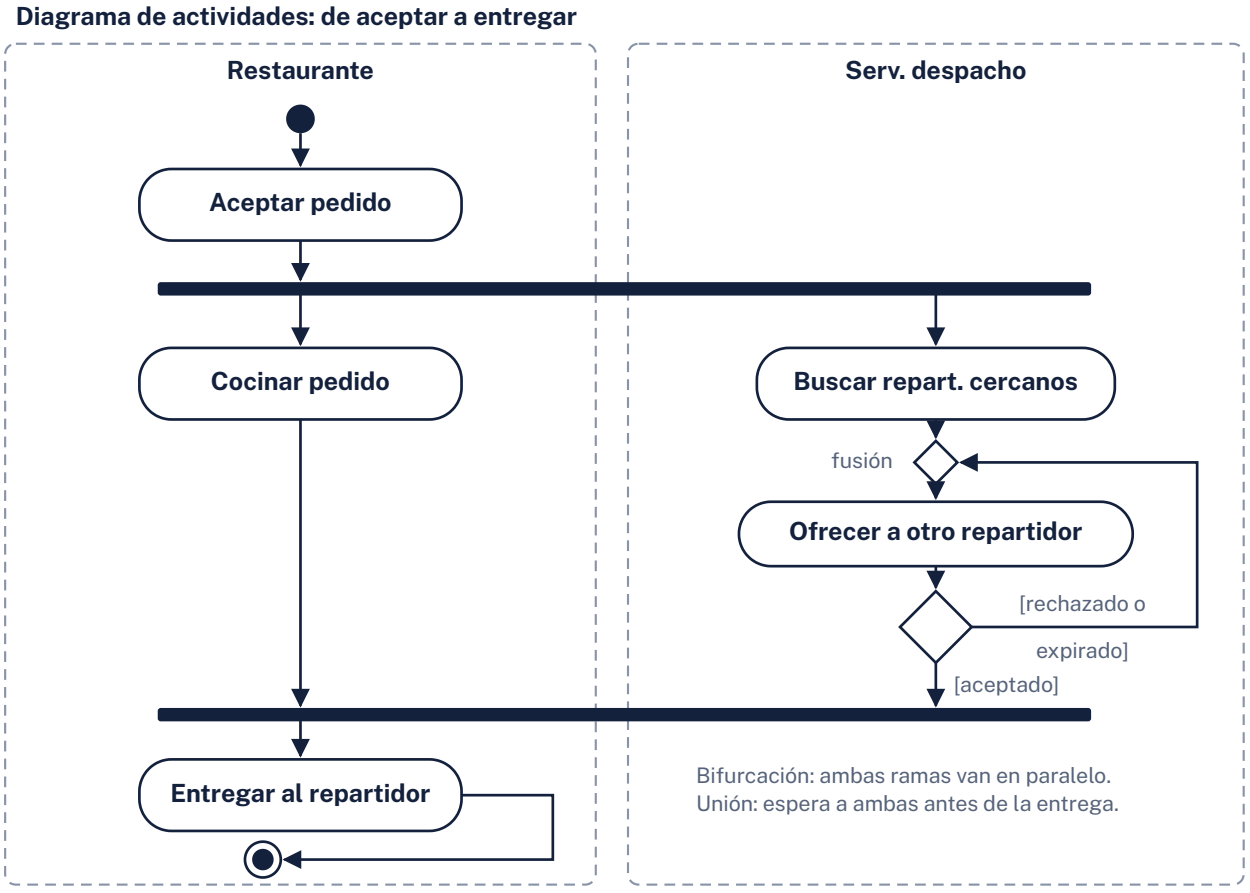
- Un pedido nuevo empieza en Pago pendiente, adonde apunta el círculo relleno. Se queda ahí hasta que llega un evento que nombra una de sus flechas de salida.
- Llega el evento paymentAuthorised, y su flecha lleva a Creado. Esa etiqueta es solo un evento: sin condición y sin acción. Al llegar, se ejecuta el comportamiento de entrada de Creado, avisar restaurante, para que el restaurante se entere del pedido.
- El evento restaurantAccepts lleva el pedido a Aceptado. Ahora llega una solicitud de cancelación. La flecha que sale de Aceptado dice cancel [soporte lo aprueba] / devolver: cancel es el evento, la guarda entre corchetes es la condición que debe cumplirse y devolver (el reembolso), tras la barra, es la acción que se realiza por el camino.
- Si soporte no lo ha aprobado, la guarda es falsa: la transición no se dispara y el pedido simplemente sigue en Aceptado.
- Si soporte lo ha aprobado, la transición se dispara: el reembolso se hace por el camino y el pedido termina en Cancelado.
- Si el pedido ya hubiera estado En reparto, ninguna flecha con la etiqueta cancel sale de ese estado, así que la solicitud no podría cancelarlo. Una flecha que falta también es una regla.
- **La lección:** lee cada flecha como 'cuando ocurra este evento, y si esta condición se cumple, haz esto y pasa allí'. Acordar estas etiquetas en un solo diagrama ayuda a que varios servicios respeten las mismas reglas para un pedido.

Diagramas de actividades: flujos de trabajo y paralelismo

Un diagrama de actividades es un diagrama de flujo con una semántica precisa. Los rectángulos redondeados son **acciones**; los rombos son **decisiones** (con guardas en los flujos de salida) y fusiones; las barras gruesas son **bifurcaciones**, que dividen un flujo en flujos paralelos, y **uniones**, que esperan a todos los flujos de entrada antes de continuar. Las **particiones**, a menudo llamadas calles (swimlanes), muestran quién o qué realiza cada acción. Un bucle debe volver a través de un rombo de fusión: dos flujos que entraran directamente en una misma acción significarían que la acción espera a ambos, como una unión.

Los arquitectos usan los diagramas de actividades para procesos de negocio y flujos de trabajo que abarcan varias partes de un sistema, sobre todo cuando importan el paralelismo y las responsabilidades. El ejemplo muestra que la preparación de la comida y la asignación del repartidor se ejecutan en paralelo, que la asignación se repite en bucle hasta que un repartidor acepta y que la entrega espera a ambas. Eso plantea de inmediato una pregunta arquitectónica: ¿y si ningún repartidor acepta en veinte minutos mientras la comida ya está lista? El diagrama hace visible el hueco; la respuesta podría ser una transición por timeout, un paso de compensación o un cambio en el ciclo de vida del pedido. Los flujos de trabajo de larga duración de este tipo suelen implementarse como sagas o flujos orquestados, que se tratan en el Capítulo 6.

Diagrama de actividades



La preparación de la comida y la asignación del repartidor se ejecutan en paralelo entre una bifurcación y una unión; la decisión se repite en bucle hasta que un repartidor acepta.

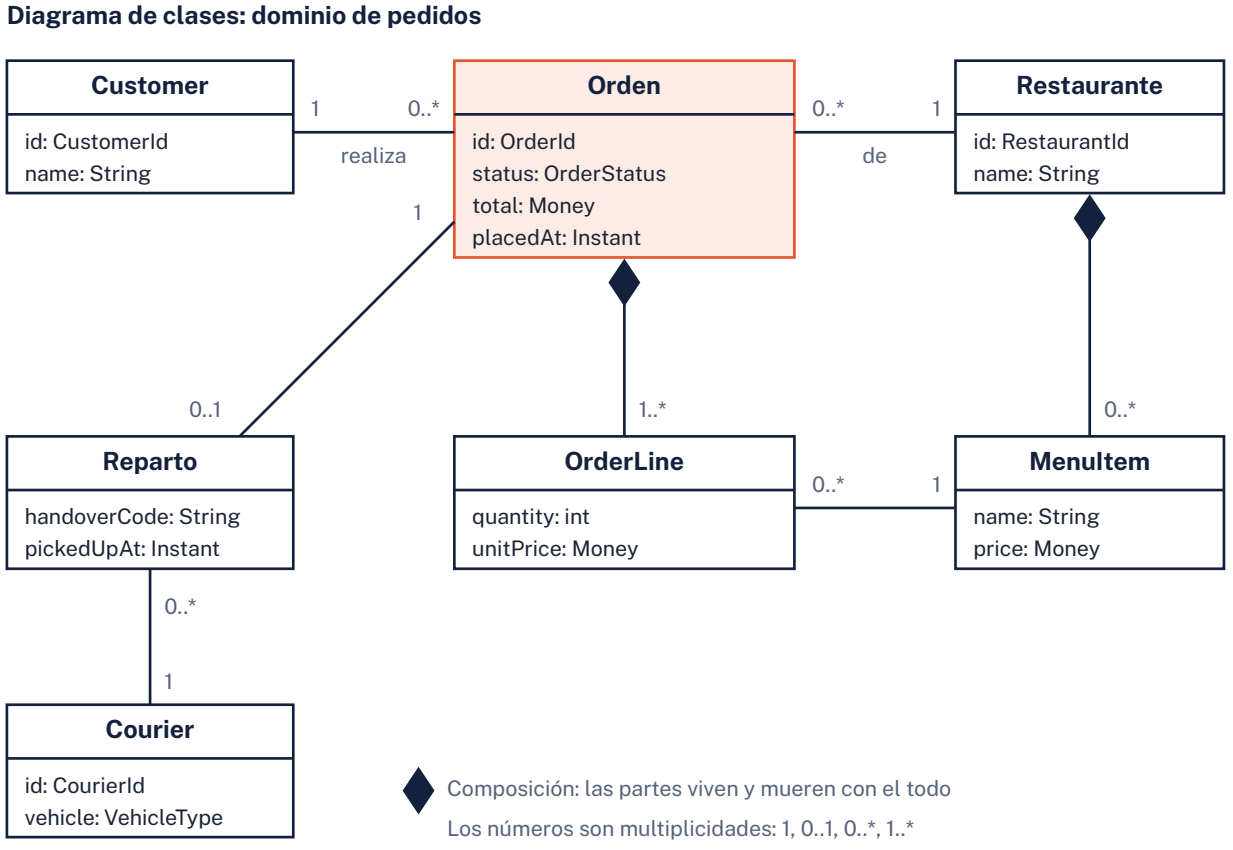
Diagramas de clases, con moderación: el modelo de dominio

El diagrama de clases es el diagrama UML más conocido y el que los arquitectos deberían usar con más moderación. Dibujar a mano todas las clases de un servicio es el modo plano en su versión menos útil: queda desactualizado en semanas y el código dice lo mismo con más

precisión. Donde los diagramas de clases se ganan su sitio es en el modelo de dominio: un diagrama pequeño de los conceptos de negocio clave, sus atributos y sus relaciones, que desarrolladores, responsables de producto y expertos del dominio pueden discutir juntos. En términos de diseño guiado por el dominio (Capítulo 5), recoge el lenguaje ubicuo de un contexto delimitado.

Lo esencial es poco. Una caja de clase tiene un nombre, atributos y, opcionalmente, operaciones. Una línea simple es una **asociación**, con nombre opcional y con multiplicidades en cada extremo (`1`, `0..1`, `0..*`, `1..*`). Un rombo relleno marca la **composición**: las partes pertenecen exactamente a un todo y no le sobreviven, como las líneas de pedido pertenecen a un pedido. Un triángulo hueco marca la generalización (herencia) y una línea discontinua con punta abierta, una dependencia. Las multiplicidades encierran decisiones de diseño reales: `Order 1 to 0..1 Delivery` dice que un pedido tiene como mucho una entrega, lo que descarta repartir un pedido grande entre dos repartidores. Si el negocio pudiera querer eso, el diagrama es donde aflora la pregunta.

Diagrama de clases del modelo de dominio



Un modelo de dominio lo bastante pequeño para discutirlo con el negocio: conceptos clave, atributos, asociaciones y multiplicidades.

Los seis diagramas de un vistazo

Diagrama	Responde a	Notación clave	Uso típico por parte de los arquitectos
Componente	¿Cuáles son las partes y a través de qué interfaces dependen unas de otras?	Cajas de componentes, interfaces proporcionadas (bola) y requeridas (enchufe), dependencias	Mostrar la dirección de las dependencias y las partes sustituibles; puertos y adaptadores
Despliegue	¿Dónde se ejecuta cada parte, cuántas copias hay y cómo se conectan?	Nodos (cajas 3D), artefactos, vías de comunicación	Infraestructura de producción y preproducción (staging), zonas de red
Secuencia	¿En qué orden interactúan los participantes en este escenario?	Líneas de vida, activaciones, mensajes síncronos y asíncronos, respuestas, fragmentos	Escenarios clave en tiempo de ejecución, sobre todo fallos, reintentos e integraciones
Máquina de estados	¿En qué estados puede estar este objeto y qué lo hace pasar de uno a otro?	Estados, estados inicial y final, transiciones con evento [guarda] / acción	Ciclos de vida compartidos entre servicios: pedidos, pagos, siniestros
Actividades	¿Cuál es el flujo de trabajo, quién hace cada paso y qué se ejecuta en paralelo?	Acciones, decisiones, bifurcaciones y uniones, particiones	Procesos de negocio, flujos de trabajo de larga duración, sagas
Clase	¿Cuáles son los conceptos clave y cómo se relacionan?	Clases, asociaciones, multiplicidades, composición, generalización	Modelos de dominio pequeños; diagramas generados a partir del código

Cuándo UML es excesivo y cuándo importa la precisión

UML es excesivo cuando el público no lo conoce y la pregunta no lo necesita: un diagrama de contexto C4 comunica mejor con los directivos que un diagrama de casos de uso UML, y un boceto en la pizarra es la herramienta adecuada para una discusión de diseño que habrá terminado en una hora. Es excesivo cuando se mantienen a mano diagramas que podrían generarse y cuando se impone una notación estricta en un boceto cuyo propósito es explorar. La observación de Fowler de que la precisión añadida en UML 2 sirve sobre todo a quienes lo usan como plano y como lenguaje de programación, y supone una carga para quienes hacen bocetos, sigue siendo válida.

La precisión compensa cuando un malentendido sale caro. Las **interfaces entre empresas** son el caso más claro: cuando la empresa de reparto de comida se integra con un proveedor de pagos o con el sistema de pedidos de una gran cadena de restaurantes, cada parte construye a partir de la descripción de la otra y solo descubre los desajustes en las pruebas o en producción. Un diagrama de secuencia preciso del flujo de pago, con timeouts, reintentos, claves de idempotencia y webhooks, adjunto a la especificación de la interfaz, es un seguro barato. Las **industrias reguladas y críticas para la seguridad (safety)** (dispositivos médicos, ferrocarril, aviación, automoción, algunos sistemas financieros) a menudo exigen documentación de diseño que los auditores puedan rastrear desde los requisitos hasta la implementación; allí es habitual el UML formal o su pariente de la ingeniería de sistemas, SysML. Las **reglas de concurrencia y de ciclo de vida** que comparten varios equipos, como los estados del pedido de antes, se benefician de una notación que obliga a declarar cada transición. Y el **desarrollo externalizado o distribuido**, en el que quienes implementan un diseño no pueden preguntar fácilmente a su autor, necesita descripciones que dejen menos margen a la interpretación.

ATENCIÓN

Errores comunes con UML

- Usar puntas de flecha equivocadas en los diagramas de secuencia, de modo que un evento asíncrono parezca una llamada bloqueante, o al revés. Los lectores sacan conclusiones arquitectónicas de ellas.
- Dibujar a mano diagramas de clases de todo el sistema. Genéralos si hacen falta y dibuja solo el modelo de dominio.
- Omitir la leyenda porque "es UML estándar". Muchos lectores no conocen la notación; una leyenda de dos líneas no cuesta nada.
- Un diagrama de secuencia gigante con todas las ramas. Dibuja un diagrama por escenario y haz diagramas aparte para los escenarios de fallo.
- Exigir UML completo en un boceto, lo que ralentiza la discusión y quita a la gente las ganas de hacer diagramas.

Preguntas frecuentes

¿Debería nuestro equipo aprender UML?
Aprende lo esencial de la notación de los seis diagramas de esta sección; lleva unas pocas horas y da al equipo un vocabulario compartido para interfaces, interacciones y ciclos de vida. Apenas hace falta aprender los demás tipos de diagrama o la especificación completa, salvo que trabajes en un dominio que los exija.

¿Qué relación hay entre UML y C4?
Son complementarios. C4 ofrece un pequeño conjunto de abstracciones y una jerarquía de diagramas estructurales; UML ofrece notaciones precisas, sobre todo para el comportamiento. Muchos equipos usan C4 para las vistas de contexto, contenedores y componentes, y UML para los diagramas de secuencia, de estados y, de vez en cuando, de despliegue.

¿Puedo dibujar UML con texto en lugar de con una herramienta de dibujo?
Sí. Tanto PlantUML como Mermaid generan diagramas de secuencia, de estados, de clases y de actividades a partir de archivos de texto breves que pueden vivir en el control de versiones. La Sección 4.7 muestra ejemplos.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ **En un diagrama de secuencia, ¿cómo distingues una llamada síncrona de un mensaje asíncrono?**

Una llamada síncrona tiene una línea continua con una punta de flecha rellena (el emisor espera); un mensaje asíncrono tiene una línea continua con una punta abierta (el emisor continúa). Las respuestas son líneas discontinuas con puntas abiertas.

▼ **¿Qué significa la etiqueta `handedOver [code matches] / notify customer` en una transición?**

Cuando ocurre el evento handedOver y la guarda "el código coincide" (code matches) es verdadera, la transición se dispara y se realiza la acción "avisar al cliente" (notify customer).

▼ **¿Por qué los arquitectos rara vez deberían dibujar a mano diagramas de clases de servicios enteros?**

Duplican lo que el código expresa con más precisión y se desactualizan rápido; vale la pena dibujar modelos de dominio pequeños, y los diagramas completos pueden generarse cuando hagan falta.

▼ **Da dos situaciones en las que el UML preciso merece el esfuerzo.**

Por ejemplo, las interfaces entre empresas, el desarrollo regulado o crítico para la seguridad (safety), los ciclos de vida que comparten varios equipos o el desarrollo externalizado en el que quienes implementan no pueden preguntar fácilmente a los diseñadores.

▼ **Inténtalo: Tres equipos de reparto de comida no paran de discrepar sobre cuándo se puede cancelar un pedido. ¿Qué diagrama UML dibujarías? Da un ejemplo de transición en la notación correcta y di por qué aquí vale la pena pagar por más precisión.**

Un diagrama de máquina de estados, porque la discusión trata del ciclo de vida de un objeto. Ejemplo: de Confirmed a Cancelled con cancelRequested [la cocina no ha empezado] / refundPayment, siguiendo evento [guarda] / acción. Un diagrama de secuencia mostraría una ruta, pero no todos los estados permitidos. La precisión vale la pena porque varios equipos comparten este ciclo de vida y los errores cuestan dinero real. Un boceto cuidadoso de una docena de estados zanja las discusiones sin necesidad de un modelo UML completo del sistema.

FUENTES

- UML specification versions (Object Management Group): https://www.omg.org/spec/UML/
- About the Unified Modeling Language 2.5.1 (OMG): https://www.omg.org/spec/UML/2.5.1/About-UML/
- UmlMode (Martin Fowler): https://martinfowler.com/bliki/UmlMode.html
- UmlAsSketch (Martin Fowler): https://martinfowler.com/bliki/UmlAsSketch.html

SECCIÓN 4.5 DE 4.8 · INTERMEDIO · 25 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Registros de decisiones de arquitectura (ADR)

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.8 Cómo se hace el software
- 1.2 Arquitectura frente a diseño
- 1.6 Decisiones y compromisos

Términos anteriores que utiliza: Arquitectónicamente significativo, Atributo de calidad, Compromiso, Pull request

LA IDEA EN BREVE

Imagina que te mudas a una casa y encuentras una pared que parece no servir para nada. ¿La derribas? Si el constructor hubiera dejado una nota que dijera "esta pared sostiene el baño de arriba", lo sabrías. Un registro de decisiones de arquitectura es esa nota para el software: una página breve y fechada que explica una decisión importante, las alternativas que se consideraron y lo que cuesta esa decisión. Las notas nunca se reescriben; si los propietarios cambian de opinión, añaden una nota nueva que dice qué cambió y por qué.

LO QUE NECESITAS SABER

- Un registro de decisiones de arquitectura (ADR) recoge una decisión significativa con su contexto, las opciones consideradas, la decisión y sus consecuencias; Michael Nygard popularizó la forma ligera en una entrada de blog de 2011.
- El formato de Nygard tiene título, contexto, decisión, estado y consecuencias; el contexto se escribe de forma neutral, la decisión en voz activa ("Haremos...") y las consecuencias incluyen los costos además de los beneficios.
- MADR (versión 4.0.0, 2024) añade los impulsores de decisión, las opciones consideradas con sus pros y contras, y la confirmación; una declaración Y (Y-statement) comprime una decisión en una sola frase: en el contexto de, ante, optamos por, y descartamos, para lograr, aceptando que.
- Los registros pasan de propuestos a aceptados o rechazados, y más tarde a obsoletos o reemplazados; el contenido de un registro aceptado nunca se reescribe, porque un nuevo registro que lo reemplaza conserva la historia.

- Los ADR viven en el repositorio de código como archivos Markdown numerados (por ejemplo, docs/decisions/0012-title.md), se revisan en pull requests y se indexan en una bitácora de decisiones.
- Escribe un ADR para las decisiones costosas de revertir, que afectan a cualidades clave, restringen a otros o fueron objeto de debate; evita los registros escritos a posteriori, sin alternativas reales o solo con consecuencias positivas.

Qué es un ADR y qué problema resuelve

Un registro de decisiones de arquitectura (ADR) es un documento de texto breve que recoge una decisión arquitectónicamente significativa: la situación que la impuso, las opciones que se sopesaron, lo que se eligió y lo que se deriva de esa elección. Un conjunto de ellos, guardados en orden, forma la bitácora de decisiones de un sistema. La idea de registrar las decisiones es antigua (la investigación sobre la justificación del diseño se remonta a varias décadas, y Jeff Tyree y Art Akerman publicaron una plantilla de decisión detallada en *IEEE Software* en 2005), pero la forma ligera que usan hoy la mayoría de los equipos procede de la entrada de blog de Michael Nygard *Documenting Architecture Decisions*, publicada el 15 de noviembre de 2011.

El punto de partida de Nygard era un problema que reconoce todo equipo que lleva mucho tiempo con un sistema. Un nuevo desarrollador encuentra en el código una decisión que parece extraña: una cola de más, una tabla duplicada, una biblioteca que nadie elegiría hoy. Sin saber por qué se tomó, el recién llegado tiene dos malas opciones. Puede aceptarla a ciegas, conservando algo cuyas razones quizá ya no se apliquen, o cambiarla a ciegas, rompiendo algo cuyas razones siguen vigentes. El remedio de Nygard fue poner por escrito las razones en el momento de tomar la decisión, en un pequeño archivo guardado con el código, para que la motivación viaje con el sistema. Los grandes documentos de arquitectura rara vez lo logran porque nadie los actualiza; un registro breve de una sola decisión, escrito una vez y nunca reescrito, sigue siendo cierto porque solo pretende describir lo que se sabía y se decidió en una fecha concreta.

Esto se basa directamente en la anatomía de una decisión descrita en el Capítulo 1: decisiones y compromisos: contexto, opciones, criterios, decisión con su justificación, consecuencias y estado. Un ADR es simplemente esa anatomía puesta por escrito en un formato predecible. También responde a la pregunta que arquitectura frente a diseño deja abierta en la práctica: si la arquitectura es el conjunto de decisiones significativas, la bitácora de ADR es lo más parecido que tiene un equipo a una lista de su arquitectura.

El formato de Nygard

Nygard propuso cinco partes. La mayoría de los equipos aún las usan, a menudo moviendo el estado al principio para que los lectores vean de inmediato si el registro está en vigor. Sus indicaciones sobre cómo escribir cada parte importan tanto como los encabezados.

Las cinco partes de un ADR al estilo de Nygard

Parte	Qué contiene	Cómo escribirlo bien
Título	Una frase nominal breve que nombra la decisión, con un número de secuencia, como “ADR 12: Eventos de pedido a través de un intermediario de mensajes”.	Nombra la decisión, no el problema: quien recorra la bitácora debería saber qué se decidió solo con el título.
Contexto	Las fuerzas en juego: el problema, los requisitos de calidad y las restricciones pertinentes, los hechos técnicos y organizativos, y la tensión entre ellos.	Nygard pide un lenguaje neutral en cuanto a valores: describe hechos y fuerzas, no la respuesta preferida. Incluye cifras cuando existan (carga pico, objetivos, fechas de incidentes).
Decisión	La respuesta a esas fuerzas.	Frases completas en voz activa, que empiecen por “Haremos…”, para que no haya duda sobre qué se decidió y quién lo decidió.
Estado	Propuesto, aceptado y, más adelante, posiblemente obsoleto o reemplazado, con una referencia al registro que lo sustituye.	Mantenlo al día; es el único campo que cambia tras la aceptación.
Consecuencias	El contexto resultante una vez aplicada la decisión.	Enumera todas las consecuencias, no solo las positivas: qué se vuelve más fácil, qué se vuelve más difícil, los nuevos riesgos, el nuevo trabajo y las opciones que ahora quedan cerradas.

Nygard también sugirió que un registro debería caber en una o dos páginas, que los registros se numeren de forma secuencial y que los números nunca se reutilicen, y que vivan en el repositorio del proyecto como texto plano. Cada una de esas pequeñas reglas existe por una razón: los registros breves se leen, los números estables pueden citarse en comentarios de código y mensajes de commit, y el texto plano en el repositorio puede revisarse y versionarse como el código.

Otras plantillas: MADR, declaraciones Y y formas más completas

El formato de Nygard no dice nada explícito sobre las opciones que se rechazaron, y muchos equipos descubrieron que las alternativas son precisamente lo que más quiere saber un lector posterior. Varias plantillas las añaden.

MADR (Markdown Architectural Decision Records, mantenida en adr.github.io) es la plantilla estructurada más usada. Su versión vigente en el momento de escribir esto es la 4.0.0, publicada en septiembre de 2024. La plantilla completa tiene un título y, después, *Contexto y planteamiento del problema*, *Impulsores de decisión* (opcional), *Opciones consideradas*, *Resultado de la decisión* con las subsecciones opcionales *Consecuencias* y *Confirmación*, *Pros y contras de las opciones* (opcional) y *Más información* (opcional). Los metadatos, como el estado, la fecha, los responsables de la decisión y a quién se consultó y a quién se informó, pueden ir en una cabecera YAML. Dos ideas de MADR merecen adoptarse aunque uses otro formato. Los impulsores de decisión hacen explícitos los criterios antes de juzgar las opciones, que es la disciplina de la matriz ponderada del Capítulo 1 en forma compacta. La *confirmación* indica cómo comprobará el equipo que la decisión se implementa realmente y sigue siendo válida, por ejemplo mediante una revisión, una prueba o una comprobación automatizada de la arquitectura.

La declaración Y, propuesta por Olaf Zimmermann y sus colegas y presentada por primera vez en la conferencia SATURN en 2012, comprime una decisión en una sola frase estructurada: *en el contexto de* (un caso de uso o un componente), *ante* (una inquietud o un requisito de calidad), *optamos por* (la opción) *y descartamos* (las alternativas), *para lograr* (los beneficios), *aceptando que* (los inconvenientes). El nombre es un juego de palabras en inglés: la Y suena como “why” (por qué). Una declaración Y es ideal para el resumen de una bitácora de decisiones, una diapositiva, un comentario de código o la descripción de un pull request; para una puerta de un solo sentido que es objeto de debate suele quedarse demasiado comprimida, y los equipos la amplían hasta un registro completo.

Existen plantillas más pesadas para entornos regulados o muy grandes. La plantilla de Tyree y Akerman de 2005 añade campos como supuestos, restricciones, decisiones relacionadas, requisitos relacionados e implicaciones. Son útiles cuando las decisiones deben poder trazarse hasta los requisitos para un auditor; para la mayoría de los equipos de producto añaden un esfuerzo que nadie lee.

Elegir una plantilla

Plantilla	Tamaño	¿Registra las opciones rechazadas?	Más adecuada para
Nygard (2011)	De media página a dos páginas	Solo si las incluyes en el contexto o en las consecuencias	Equipos que empiezan; el hábito más pequeño que funciona.
MADR 4.0.0 (2024)	De una a tres páginas	Sí: opciones consideradas, con pros y contras opcionales de cada una	La mayoría de los equipos de producto; decisiones debatidas en las que las alternativas deben quedar a la vista.
Declaración Y (2012)	Una frase	Sí, en la cláusula *y descartamos*	Resúmenes de la bitácora de decisiones, decisiones pequeñas, descripciones de pull requests, diapositivas.
Tyree y Akerman (2005)	Varias páginas	Sí, además de supuestos, restricciones y trazabilidad	Entornos regulados que necesitan trazabilidad hasta los requisitos.

Estado, ciclo de vida y por qué no se editan los registros aceptados

Un ADR pasa por un número reducido de estados. Empieza como **propuesto**, normalmente como un pull request que añade el archivo, para que el debate tenga lugar en los comentarios de la revisión. Cuando las personas responsables están de acuerdo, se fusiona como **aceptado**. Una propuesta que no prospera puede conservarse con el estado **rechazado**, lo cual es valioso: la próxima persona que sugiera la misma idea encontrará el razonamiento anterior. Más adelante, un registro aceptado puede pasar a **obsoleto** (la decisión ya no se aplica, quizá porque se retiró el componente) o quedar **reemplazado** por un registro más reciente que lo sustituye, con enlaces en ambas direcciones. MADR enumera el mismo conjunto: propuesto, rechazado, aceptado, obsoleto y reemplazado.

La regla central es que el contenido de un registro aceptado no se reescribe. Si el equipo cambia de opinión, escribe un nuevo ADR que reemplaza al anterior y solo cambia la línea de estado del registro antiguo para que apunte al nuevo. Las razones son prácticas. Primero, el registro es una prueba de lo que se sabía y se decidió en un momento dado; editarlo a posteriori convierte la historia en ficción y destruye la confianza que hace útil la bitácora. Segundo, el registro antiguo explica por qué el sistema era como era durante un periodo, lo que importa al depurar datos antiguos o al leer código antiguo. Tercero, un nuevo registro obliga al equipo a explicar qué cambió en el contexto, que suele ser la idea más importante. Las correcciones

pequeñas (una errata, un enlace roto) son inofensivas; cambiar la decisión, las razones o las consecuencias no lo es. La descripción de la práctica que hace Martin Fowler dice lo mismo: una vez aceptado, un registro no se reabre; en su lugar se escribe un registro que lo reemplaza.

El ciclo de vida de un ADR



Los registros se proponen y se revisan, y luego se aceptan o se rechazan; los aceptados pasan más tarde a obsoletos o son reemplazados, nunca reescritos.

SI ESTO ES NUEVO PARA TI

Paso a paso: cambiar de opinión sin reescribir la historia

La idea que requiere más atención aquí es por qué un registro aceptado nunca se reescribe, ni siquiera cuando el equipo cambia de opinión. El Capítulo 1 enunció la regla; sigue un cambio real en la bitácora de decisiones de la plataforma de reparto de comida para ver lo que significa en la práctica.

- En 2025 el equipo acepta el ADR-0009: el servicio de notificaciones envía las notificaciones push a través de un SDK de terceros. Su contexto, su decisión y sus consecuencias describen lo que el equipo sabe y elige en ese momento.
- En 2026, después de que el ADR-0012 haya introducido un intermediario de mensajes, el equipo decide sustituir el SDK por su propia entrega push, que se ejecuta sobre el intermediario.
- El atajo tentador es abrir el ADR-0009 y reescribirlo para que describa el nuevo diseño. Pero entonces la bitácora afirma que el equipo eligió su propia entrega push ya en 2025, y quien lea código antiguo o datos antiguos de notificaciones no encontrará ningún registro de que el SDK se usara alguna vez, ni de por qué.
- En su lugar, el equipo escribe un registro nuevo, el ADR-0015, y lo propone en un pull request como cualquier otro. Su contexto dice qué ha cambiado desde el ADR-0009, que a menudo es la parte más útil del registro.
- Cuando se acepta el ADR-0015, el único cambio en el ADR-0009 es su línea de estado, que ahora dice 'Reemplazado por ADR-0015'. El ADR-0015 enlaza de vuelta con él, y la bitácora de decisiones incluye ambos.
- Quien llega nuevo y lee la bitácora ve ahora toda la historia: qué se decidió en 2025, qué lo reemplazó en 2026 y por qué.
- **La lección:** un registro aceptado describe lo que se sabía y se decidió en su fecha, y por eso sigue siendo cierto. Las ideas nuevas van en un registro nuevo; aparte de pequeñas correcciones, como una errata, el antiguo solo cambia su línea de estado.

Dónde guardar los ADR

Guarda los ADR en el repositorio del código que gobiernan, como archivos de texto plano (normalmente Markdown), una decisión por archivo, con un número de secuencia y un slug breve en el nombre del archivo. Hay dos convenciones de directorio habituales: `doc/adr`, la predeterminada de la herramienta de línea de comandos `adr-tools` de Nat Pryce, y `docs/decisions`, recomendada por MADR con nombres como `0012-publish-order-events-`

`through-broker.md`. Thoughtworks colocó los ADR ligeros en el anillo *Adopt* (adoptar) de su Technology Radar (la última vez en 2018) y recomendó expresamente el control de código fuente en lugar de una wiki, para que los registros vayan al paso del código que describen.

Guardar los registros junto al código tiene ventajas concretas. Se revisan en los mismos pull requests que el código que los implementa, se versionan y se ramifican con él, un desarrollador puede encontrarlos con las mismas herramientas de búsqueda, y el código puede apuntar a ellos: un comentario como "ver ADR-0012" encima del retransmisor de la bandeja de salida (outbox relay) le indica al siguiente lector dónde están las razones. Las decisiones que abarcan muchos repositorios, como la elección de un proveedor de identidad para toda la organización, necesitan un lugar propio; un repositorio de arquitectura dedicado con las mismas convenciones funciona bien, con enlaces desde cada base de código afectada.

```
docs/
  decisions/
    README.md                                  ← the decision log (index)
    0001-record-architecture-decisions.md
    0004-use-hosted-card-fields.md
    0007-use-managed-postgresql-for-orders.md
    0012-publish-order-events-through-broker.md
    template.md
src/
  ordering/
    outbox_relay.py      # comment: see docs/decisions/0012-...
```

Una estructura típica: archivos ADR numerados junto al código, un índice, una plantilla y un comentario de código que apunta de vuelta al registro

Qué decisiones merecen un ADR

Escribir un ADR para cada elección entierra las importantes; no escribir ninguno pierde las razones. El criterio es que la decisión sea significativa, tal como se definió en el Capítulo 1: una decisión merece un registro cuando es cara de revertir, afecta a un atributo de calidad clave, tiene un alcance amplio, restringe a otros equipos o fue objeto de debate. En la práctica, eso significa elecciones como el almacén de datos principal, el estilo de comunicación entre servicios, el proveedor de identidad, la política de versionado de API, la plataforma de alojamiento, una decisión de construir o comprar, y las convenciones cuyo incumplimiento sale caro (cómo se representa el dinero, cómo se almacenan las zonas horarias). También incluye las decisiones de *no* hacer algo que la gente sigue proponiendo, como no introducir todavía una caché.

Las elecciones locales, baratas de revertir e invisibles fuera de un módulo no suelen necesitar registro: la elección de una biblioteca de aserciones para pruebas, el nombre de una clase, la organización interna de un componente. Una señal útil es la pregunta que haría alguien recién

llegado: si es probable que alguien pregunte “¿por qué demonios hicieron esto?” antes de un año, escribe el registro ahora, mientras las razones están frescas.

GUÍA DE DECISIÓN

Guía de decisión: ¿necesita esta decisión un ADR?

Hazte estas preguntas en orden cuando surja una elección técnica en un debate de diseño o en una revisión de código.

Q1 **¿Revertir esta decisión dentro de un año llevaría más de unos pocos días de trabajo, una migración de datos o cambios por parte de otros equipos?**

- **Sí** Escribe un ADR. Es una puerta de un solo sentido o casi.
- **No** Pasa a la siguiente pregunta.

Q2 **¿Afecta de forma apreciable a un atributo de calidad que importa a las partes interesadas (por ejemplo, la disponibilidad, la latencia, la seguridad o el costo) o a una restricción como la normativa?**

- **Sí** Escribe un ADR y vincúlalo al escenario de calidad al que sirve.
- **No** Pasa a la siguiente pregunta.

Q3 **¿Fue objeto de debate, o rechazó el equipo una opción que probablemente se vuelva a proponer?**

- **Sí** Escribe un ADR breve (puede bastar una declaración Y) para que el razonamiento no se vuelva a discutir desde cero.
- **No** Pasa a la siguiente pregunta.

Q4 **¿Es probable que alguien recién llegado mire el resultado y pregunte por qué se hizo así?**

- **Sí** Escribe un ADR breve o, como mínimo, un comentario de código y un mensaje de commit que expliquen el motivo.
- **No** No hace falta un ADR. Bastan un mensaje de commit claro y el propio código.

Un ADR completo como ejemplo práctico

La plataforma de reparto de comida que se usa a lo largo de este capítulo tiene aplicaciones para clientes, restaurantes y repartidores, un servicio de pedidos, pagos a través de un proveedor de pagos externo, el despacho de repartidores y las notificaciones. El registro de abajo sigue la estructura de MADR, ligeramente abreviada. Fíjate en cómo el contexto da cifras y fechas en lugar de adjetivos, en cómo se enuncian los impulsores antes que las opciones, en cómo se trata con justicia a las opciones rechazadas y en cómo las consecuencias incluyen

costos reales. La decisión en sí (mensajería asíncrona para los eventos de pedido) se trata a fondo en el Capítulo 6: arquitectura dirigida por eventos; aquí lo que importa es cómo se registra.

```
# ADR-0012: Publish order events through a managed message broker

- Status: Accepted
- Date: 2026-09-14
- Deciders: ordering team lead, platform lead, principal architect
- Consulted: restaurant, courier and notifications teams; SRE
- Related: ADR-0007 (managed PostgreSQL for the order service)

## Context and problem statement

When a customer's payment succeeds, the ordering service calls the restaurant,
courier-dispatch and notification services synchronously over HTTP before it
confirms the order. On Friday evenings (peak about 40 orders/s) p99 order
confirmation time reaches 4.2 s against our target of 1.5 s (scenario QS-3).
On Friday 21 August a notification-service outage blocked all new orders for
23 minutes. Two new consumers (loyalty points, analytics) want order data.

## Decision drivers

- Order confirmation must not depend on notification or dispatch availability.
- p99 confirmation under 1.5 s at three times today's peak.
- No order event may be lost, even if a service crashes mid-request.
- New consumers should not require changes to the ordering service.
- The platform team has three people; operational load must stay small.

## Considered options

1. Keep synchronous calls; add timeouts, retries and circuit breakers.
2. Publish events to a managed message broker using a transactional outbox.
3. Let consumers poll a shared "order_events" table in the order database.

## Decision outcome

Chosen option: 2. We will publish OrderPlaced, OrderAccepted, OrderPickedUp,
OrderDelivered and OrderCancelled events to our cloud provider's managed
broker. The ordering service writes each event to an outbox table in the
same database transaction as the order; a relay publishes it. Payment
authorisation stays synchronous, because the customer must see the result.

### Consequences

- Good: confirmation no longer waits for downstream services.
- Good: loyalty and analytics can subscribe without ordering-team work.
- Bad: services become eventually consistent; the restaurant tablet may
  show a new order a second or two after the customer's confirmation.
- Bad: a broker to pay for and monitor; consumers must be idempotent
  because delivery is at-least-once.
- Neutral: event schemas become public contracts and need versioning.

### Confirmation
```

```
Load test at 120 orders/s in staging before go-live; a build check fails
if the ordering service gains an HTTP client for notification or dispatch.

## Pros and cons of the other options

- Option 1: least new infrastructure, but a slow dependency still delays
  every order, and each new consumer means another call to add.
- Option 3: no new infrastructure, but couples every consumer to our schema
  and adds polling load to the order database (see ADR-0007).
```

Código fuente Markdown del ADR-0012 en docs/decisions/0012-publish-order-events-through-broker.md

EJEMPLO PRÁCTICO

Ejemplo práctico: la misma decisión como declaración Y

Para la bitácora de decisiones, la descripción de un pull request o una diapositiva, el equipo también escribe la decisión como una declaración Y.

Declaración Y (Y-statement)	En el contexto de **la confirmación de pedidos en la plataforma de reparto de comida**, ante **servicios posteriores lentos o no disponibles que bloquean los pedidos en las horas pico**, optamos por **publicar los eventos de pedido a través de un intermediario de mensajes gestionado con una bandeja de salida transaccional** y descartamos **las llamadas síncronas con circuit breakers y los consumidores que consultan periódicamente una tabla compartida**, para lograr **una confirmación de pedidos que no dependa de los servicios de notificaciones, de despacho de repartidores ni de restaurantes, y nuevos consumidores que no requieran cambios en el servicio de pedidos**, aceptando que **los servicios pasan a ser eventualmente consistentes, hay que operar y pagar el intermediario, y cada consumidor debe gestionar eventos duplicados**.
Lo que conserva	El contexto, la inquietud que impulsa la decisión, la elección, las opciones rechazadas, los beneficios y los costos aceptados: todos los elementos del registro completo, de un vistazo.
Lo que pierde	Las cifras (p99 de 4.2 s, caída de 23 minutos), los responsables de la decisión, el plan de confirmación y el razonamiento contra cada opción rechazada. Para una decisión tan significativa, sigue haciendo falta el registro completo.

Una buena prueba para cualquier ADR es si puedes comprimirlo en una declaración Y sensata. Si no puedes completar las cláusulas *y descartamos* o *aceptando que*, probablemente el registro no consideró alternativas reales u ocultó sus costos.

Revisar los ADR

Como un ADR es un archivo del repositorio, el mecanismo natural de revisión es el pull request. Quien lo propone lo abre con el estado *propuesto*; las personas afectadas comentan; el registro se corrige; cuando las personas responsables están de acuerdo, el estado cambia a *aceptado* y se fusiona. Así el debate queda unido al registro y visible para cualquiera que lea más tarde el historial.

Quién debe revisar depende de cómo se gobiernen las decisiones. En un modelo centralizado, una junta de arquitectura aprueba los registros significativos. En el proceso de consejo (advice process) descrito por Andrew Harmel-Law (martinfowler.com, 2021), cualquiera puede tomar una decisión arquitectónica, siempre que antes pida consejo a todas las personas afectadas de forma significativa y a quienes tengan la experiencia pertinente, y que registre ese consejo; quien decide no está obligado a seguir el consejo, pero sí a escucharlo. Los ADR son la columna vertebral de ese proceso: el consejo aparece en el registro, y un foro asesor abierto y periódico revisa los registros propuestos. Los modelos de gobernanza se tratan en el Capítulo 11: gobernanza en organizaciones ágiles.

La lista de comprobación *ecADR* de Olaf Zimmermann (2020) ofrece a los revisores una definición de terminado compacta para una decisión. **E**videncia: hay motivos para creer que la opción funcionará, a partir de un spike, una prueba de concepto o una experiencia fiable. **C**riterios: se compararon al menos dos opciones con las inquietudes de las partes interesadas. **A**cuerdo: la decisión ha sido cuestionada por los pares y el equipo la acepta. **D**ocumentación: está registrada y compartida con los afectados. **R**ealización y revisión: la implementación está planificada y se ha fijado una fecha o un desencadenante para volver a revisarla. Un revisor que comprueba esos cinco puntos, además de la honestidad de las consecuencias, detecta la mayoría de los registros flojos.

Antipatrones

Los ADR fallan de formas reconocibles. Zimmermann catalogó varias en 2023 con nombres fáciles de recordar, y conviene conocer las más dañinas.

ATENCIÓN

Cómo fallan los ADR

- **Escrito a posteriori para justificar.** La decisión se tomó en un pasillo y el registro se reconstruye para que parezca racional. Sus opciones son inventadas y su contexto es selectivo. Escribe el registro mientras decides, como una propuesta, no como un comunicado de prensa posterior.
- **Ninguna opción considerada** (el *Sprint* de Zimmermann), o una *Dummy Alternative* (alternativa ficticia): una opción hecha para perder que nunca podría funcionar, añadida para que la favorita parezca examinada. Si no existe ninguna alternativa creíble, dilo y explica por qué.
- **Sin consecuencias reales** (el *Free Lunch Coupon*, o vale de comida gratis) o solo con beneficios (el *Fairy Tale*, o cuento de hadas). Toda decisión significativa tiene costos; un registro que los oculta acabará descubierto, y los lectores dejarán de confiar en la bitácora.
- **El Mega-ADR.** Diagramas, especificaciones de componentes y código metidos en un registro de varias páginas. Enlaza en su lugar con la documentación de arquitectura; el ADR registra la decisión y sus razones.
- **Editar registros aceptados** para que coincidan con el diseño actual, lo que destruye la historia que la bitácora existe para conservar.
- **ADR para todo,** incluidas las puertas de doble sentido triviales, hasta que nadie los lee; o ADR que nadie encuentra, dispersos en wikis, hilos de chat y presentaciones.

La bitácora de decisiones como índice

Los registros individuales son útiles; la bitácora en su conjunto lo es más. Un breve archivo de índice (a menudo el `README.md` de la carpeta de decisiones) enumera cada registro con su número, título, estado y fecha. Quien llega nuevo lo lee en cinco minutos y obtiene una historia comprimida de la arquitectura: qué se decidió, qué se rechazó, qué se reemplazó y cuándo. Herramientas como `adr-tools` pueden generar el índice y crear registros de reemplazo que actualizan automáticamente el estado del registro antiguo, y herramientas de documentación como el DSL de Structurizr pueden importar una carpeta de ADR a la documentación publicada.

Trata la bitácora como parte de la descripción de la arquitectura. La plantilla arc42 tiene una sección para las decisiones de arquitectura que puede limitarse a enlazar con ella, y un pipeline de documentación como código puede publicarla junto con el resto de la documentación.

```
# Decision log

| ADR  | Title                                                | Status                | Date       |
|------|------------------------------------------------------|-----------------------|------------|
| 0001 | Record architecture decisions                        | Accepted              | 2025-03-02 |
| 0004 | Use the payment provider's hosted card fields        | Accepted              | 2025-04-18 |
| 0007 | Use managed PostgreSQL for the order service         | Accepted              | 2025-06-11 |
| 0009 | Send push notifications through a third-party SDK    | Superseded by 0015    | 2025-09-30 |
| 0012 | Publish order events through a managed broker        | Accepted              | 2026-09-14 |
| 0013 | Store courier locations in a time-series database    | Rejected              | 2026-09-20 |
| 0015 | Replace the SDK with our own push delivery on broker | Accepted              | 2026-10-08 |
```

docs/decisions/README.md: la bitácora de decisiones de la plataforma de reparto de comida

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Por qué la sección de contexto de un ADR debe evitar argumentar a favor de la opción elegida?

Porque su función es registrar las fuerzas y los hechos tal como eran, para que un lector posterior pueda juzgar si siguen siendo válidos. Si el contexto se escribe como un alegato, el lector no puede separar la situación de la conclusión.

▼ Tu equipo cambia de opinión sobre una decisión aceptada. ¿Qué haces con el ADR antiguo?

Dejar su contenido sin cambios, cambiar su estado a reemplazado con un enlace al nuevo registro y escribir un nuevo ADR que explique qué cambió en el contexto y qué se decide ahora.

▼ ¿Qué aporta la sección de confirmación de MADR que le falta al formato de Nygard?

Una indicación de cómo comprobará el equipo que la decisión se ha implementado y sigue siendo válida, como una prueba de carga, una revisión o una comprobación automatizada de dependencias.

▼ **¿Cuál es una forma rápida de comprobar si un ADR consideró de verdad las alternativas y los costos?**

Intenta escribirlo como una declaración Y. Si las cláusulas "y descartamos" o "aceptando que" no pueden completarse con honestidad, al registro le faltan opciones reales o consecuencias reales.

▼ **Inténtalo: Critica este ADR: "Elegimos Kafka porque es el mejor broker. Consecuencias: rápido y escalable". Seis meses después, alguien lo edita para que diga RabbitMQ. ¿Qué está mal y qué debería haberse hecho cada vez?**

No da contexto, ni alternativas, ni criterios, y "lo mejor" no es una razón. Las consecuencias enumeran solo beneficios, sin costos como operar el clúster. Debería plantear el problema con números, comparar con equidad al menos dos opciones, luego decir "Haremos..." y nombrar a qué se renuncia. Editarlo después destruye el historial: escribe un nuevo ADR que lo sustituya y marca el antiguo como sustituido.

FUENTES

- Documenting Architecture Decisions, Michael Nygard (2011): https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
- MADR: Markdown Architectural Decision Records: https://adr.github.io/madr/
- Architecture Decision Record, Martin Fowler (bliki): https://martinfowler.com/bliki/ArchitectureDecisionRecord.html
- Scaling the Practice of Architecture, Conversationally, Andrew Harmel-Law: https://martinfowler.com/articles/scaling-architecture-conversationally.html
- A Definition of Done for Architectural Decision Making, Olaf Zimmermann: https://ozimmer.ch/practices/2020/05/22/ADDefinitionOfDone.html
- Jeff Tyree and Art Akerman: Architecture Decisions: Demystifying Architecture (IEEE Software, 2005): https://doi.org/10.1109/MS.2005.27

SECCIÓN 4.6 DE 4.8 · INTERMEDIO · 22 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

arc42 y otras plantillas de documentación

ANTES DE LEER ESTO

Esta sección se basa en:

- 4.2 Vistas y puntos de vista
- 4.3 El modelo C4
- 4.5 Registros de decisiones

Términos anteriores que utiliza: Descripción de arquitectura, Punto de vista, Correspondencia, Escenario de atributo de calidad, Registro de decisiones de arquitectura (ADR)

LA IDEA EN BREVE

Una plantilla es como la estructura estándar del manual de un vehículo: todos los manuales tienen una sección sobre los neumáticos, otra sobre los testigos de advertencia y otra sobre el mantenimiento, así que sabes dónde buscar en cualquier vehículo. arc42 hace lo mismo para la arquitectura de software. Ofrece doce secciones estándar, como objetivos, contexto, estructura y decisiones, para que quien escribe sepa qué cubrir y quien lee sepa dónde encontrarlo. No tienes que completar todas las secciones; un sistema pequeño puede necesitar solo cuatro o cinco breves.

LO QUE NECESITAS SABER

- arc42, creada por Gernot Starke y Peter Hruschka y gratuita bajo una licencia Creative Commons, es una plantilla de doce secciones; prescribe dónde va la información, no cómo diseñar ni qué notación usar.
- Las secciones son introducción y objetivos, restricciones, contexto y alcance, estrategia de solución, vista de bloques de construcción, vista de tiempo de ejecución, vista de despliegue, conceptos transversales, decisiones de arquitectura, requisitos de calidad, riesgos y deuda técnica, y glosario.
- Todas las secciones son opcionales; una descripción mínima cubre de tres a cinco escenarios de calidad, el contexto, una breve estrategia de solución, la vista de bloques de construcción de nivel 1 y los conceptos transversales clave.
- Usa arc42 de forma ligera: enlaza con los requisitos, los ADR y el código de infraestructura en lugar de copiarlos, y deja constancia expresa cuando una sección no sea necesaria.

> - La plantilla Views and Beyond del SEI es más exhaustiva, con una hoja de ruta, paquetes por vista y una correspondencia entre vistas; IEEE 1016-2009 especifica las descripciones del diseño de software y está en estado inactive-reserved (inactiva reservada) desde 2020.
> - ISO/IEC/IEEE 42010 prescribe el contenido de una descripción de arquitectura, no una plantilla; una descripción arc42 es conforme cuando se hacen explícitos las partes interesadas, las inquietudes, los puntos de vista, las vistas, las correspondencias y la justificación.

Qué es arc42

arc42 es una plantilla gratuita para describir y comunicar la arquitectura de software, creada por Gernot Starke y Peter Hruschka y usada en la práctica desde 2005. Se publica bajo una licencia Creative Commons Attribution-ShareAlike (Reconocimiento-CompartirIgual), en más de una docena de idiomas y en formatos como AsciiDoc, Markdown y Word, con una versión "simple" y otra con texto de ayuda integrado. Está especialmente extendida en los países de habla alemana y en la formación de iSAQB, pero se usa en todo el mundo.

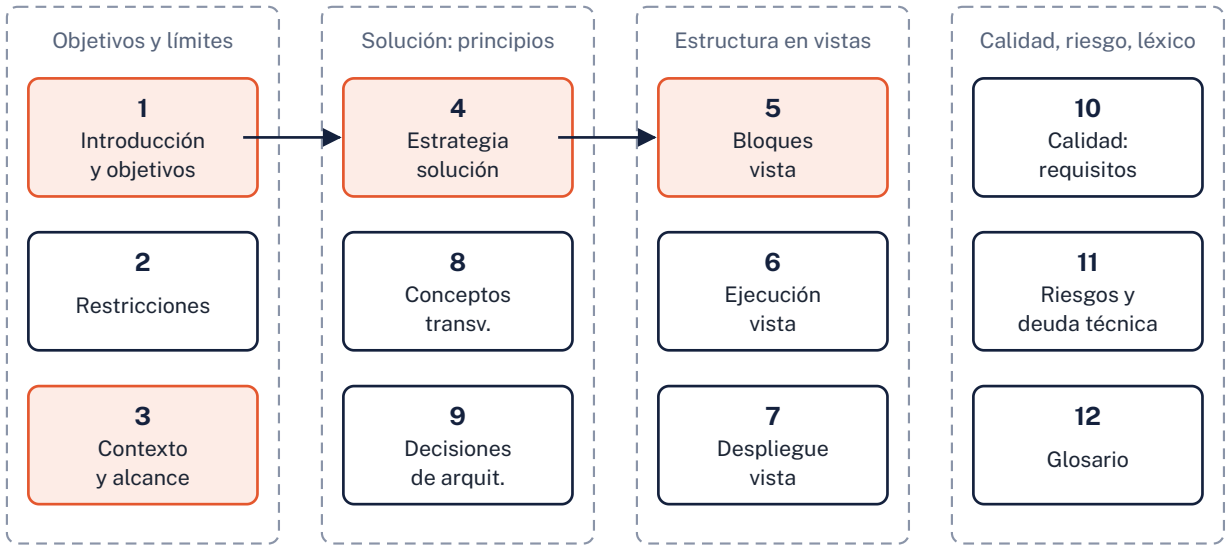
arc42 no es, deliberadamente, ni un método ni una notación. No te dice cómo diseñar ni prescribe UML, C4 ni ningún otro estilo de diagrama. Es un conjunto de doce cajones etiquetados: para cada tipo de información arquitectónica, dice dónde ponerla, para que todo lector sepa dónde buscar y todo autor sepa qué falta. Sus autores dejan claro que todas las secciones son opcionales; un sistema pequeño puede completar cuatro secciones en unas pocas páginas, y uno grande puede completar las doce en muchas.

La plantilla se basa en ideas vistas antes en este capítulo. Sus tres vistas (bloques de construcción, tiempo de ejecución y despliegue) se corresponden de cerca con las categorías de módulos, de componentes y conectores y de asignación del SEI, y con las vistas de desarrollo, de procesos y física de Kruchten de vistas y puntos de vista. Su sección de decisiones es donde van los registros de decisiones de arquitectura, y su sección de calidad es donde van los escenarios de atributos de calidad del Capítulo 2.

Las doce secciones

El diagrama agrupa las secciones según la pregunta que responden. Las secciones 1 a 3 fijan los objetivos y los límites: para qué sirve el sistema, qué limita la solución y dónde termina el sistema. La sección 4 enuncia las ideas fundamentales de la solución, y las secciones 8 y 9 contienen los principios y las decisiones que se aplican a toda ella. Las secciones 5 a 7 muestran la estructura desde tres ángulos. Las secciones 10 a 12 hacen explícitos la calidad, el riesgo y el vocabulario.

La estructura de arc42



Sombreado: el núcleo de toda descripción mínima (objetivos, contexto, estrategia y bloques de nivel 1); las FAQ de arc42 añaden de 3 a 5 escenarios de calidad y conceptos clave (8).

Las doce secciones de arc42 agrupadas por finalidad; las secciones resaltadas son el núcleo incluso de una descripción mínima.

Qué va en cada sección, y cuánto

Sección	Qué va en ella	Sistema pequeño (un equipo)	Sistema grande (muchos equipos)
1. Introducción y objetivos	Un breve resumen de los requisitos, los tres a cinco objetivos de calidad principales por orden de prioridad y una tabla de partes interesadas con las expectativas de cada grupo.	Media página: propósito, tres objetivos de calidad, cinco o seis partes interesadas.	Dos o tres páginas, que enlacen con los documentos de requisitos en lugar de copiarlos.
2. Restricciones	Todo lo que limite la libertad de diseño: normativa, tecnologías impuestas, reglas organizativas, plazos, presupuesto, convenciones.	Una lista de viñetas.	Una tabla agrupada en restricciones técnicas, organizativas y legales, cada una con su origen.
3. Contexto y alcance	El sistema como caja negra: los usuarios y los sistemas vecinos, lo que fluye entre ellos (contexto de negocio) y los canales y protocolos usados (contexto técnico).	Un diagrama de contexto (a menudo un diagrama de contexto del sistema de C4) y una tabla breve de interfaces externas.	Diagramas de contexto de negocio y técnico, más una tabla de interfaces con responsables y contratos.
4. Estrategia de solución	Las pocas decisiones y enfoques fundamentales que dan forma a todo lo demás: el estilo de descomposición, las tecnologías clave, cómo se logran los principales objetivos de calidad.	Unas pocas viñetas, cada una vinculada a un objetivo de calidad.	Una o dos páginas; una tabla que relaciona cada objetivo de calidad con el enfoque que lo sirve.
5. Vista de bloques de construcción	La descomposición estática en bloques de construcción (subsistemas, servicios, componentes, módulos), refinada nivel a nivel, cada uno con sus responsabilidades e interfaces. Suele ser la sección más extensa.	Solo el nivel 1: un diagrama (sirve un diagrama de contenedores de C4) y una frase por bloque.	Nivel 1 para todo el sistema; niveles 2 y 3 solo para las partes complejas o arriesgadas.
6. Vista de tiempo de ejecución	Cómo cooperan los bloques de construcción en escenarios importantes: casos de uso clave, arranque, gestión de errores, failover.	Uno o dos diagramas de secuencia para los flujos más importantes.	Los escenarios críticos y los sorprendentes; nunca todos los casos de uso.

Sección	Qué va en ella	Sistema pequeño (un equipo)	Sistema grande (muchos equipos)
7. Vista de despliegue	La infraestructura técnica: entornos, nodos, redes y qué bloques de construcción se ejecutan en cada sitio.	Un diagrama o una breve lista de servicios alojados.	Diagramas por entorno y región, enlazados con la infraestructura como código que los define.
8. Conceptos transversales	Conceptos que se aplican a muchos bloques de construcción: modelo de dominio, seguridad, gestión de errores, logging y observabilidad, persistencia, internacionalización, pruebas.	Un párrafo breve para cada uno de los dos o tres conceptos que importan.	Una página o más por concepto, a menudo a cargo de un equipo de plataforma o de un equipo facilitador.
9. Decisiones de arquitectura	Las decisiones importantes con sus razones, salvo que se registren en otro lugar.	Un enlace a la carpeta de ADR.	Un enlace a la bitácora de decisiones, quizá con las cinco o seis decisiones que todo recién llegado debe conocer.
10. Requisitos de calidad	Un árbol de calidad (el árbol de utilidad del Capítulo 2) y escenarios de calidad concretos que refinan los objetivos de calidad de la sección 1.	De tres a cinco escenarios.	Un árbol de utilidad con escenarios priorizados, vinculados a funciones de aptitud.
11. Riesgos y deuda técnica	Los riesgos técnicos y la deuda conocidos, idealmente priorizados, con planes de mitigación.	Una lista breve, revisada cada trimestre.	Un enlace al registro de deuda, más los riesgos principales con sus responsables.
12. Glosario	Los términos importantes del dominio y técnicos, con definiciones que todos aceptan.	Unos diez términos.	Un glosario compartido, a veces por contexto delimitado.

Usar arc42 de forma ligera

La forma más común de fracasar con arc42 es tratarla como un formulario que hay que rellenar: se crean doce encabezados, cada uno se llena con algo, y el resultado es largo, genérico y nunca se lee. Las propias indicaciones de la plantilla apuntan en la dirección

contraria. Las FAQ de arc42 dicen que no hay una regla general para el mínimo, porque depende de la criticidad, el tamaño, la complejidad y el riesgo, pero sugieren que incluso una descripción mínima debería contener de tres a cinco requisitos de calidad centrales en forma de escenarios, el contexto y las interfaces externas, una breve estrategia de solución, la vista de bloques de construcción de nivel 1 y los conceptos transversales más importantes.

Cuatro hábitos mantienen arc42 ligera. **Elimina las secciones vacías o márcalas de forma deliberada** ("Despliegue: una única plataforma gestionada; ver el repositorio de infraestructura"), para que el lector sepa que el hueco es intencionado. **Enlaza en lugar de copiar**: los requisitos viven en el backlog, las decisiones en la carpeta de ADR, el despliegue en la infraestructura como código; arc42 apunta a ellos. **Prefiere diagramas con tablas breves de elementos** a páginas de prosa. **Escribe para las preguntas del próximo lector**, no por exhaustividad: si nadie va a preguntar por el comportamiento en tiempo de ejecución de las pantallas de administración, no lo describas.

arc42 se combina bien con el modelo C4. Una combinación habitual es un diagrama de contexto del sistema de C4 en la sección 3, un diagrama de contenedores como nivel 1 de la vista de bloques de construcción en la sección 5, con diagramas de componentes como nivel 2, y un diagrama de despliegue de C4 en la sección 7. Si se mantienen como diagramas como código en el mismo repositorio que el texto de arc42, toda la descripción puede versionarse y revisarse como el código.

SI ESTO ES NUEVO PARA TI

Paso a paso: una referencia, no una copia

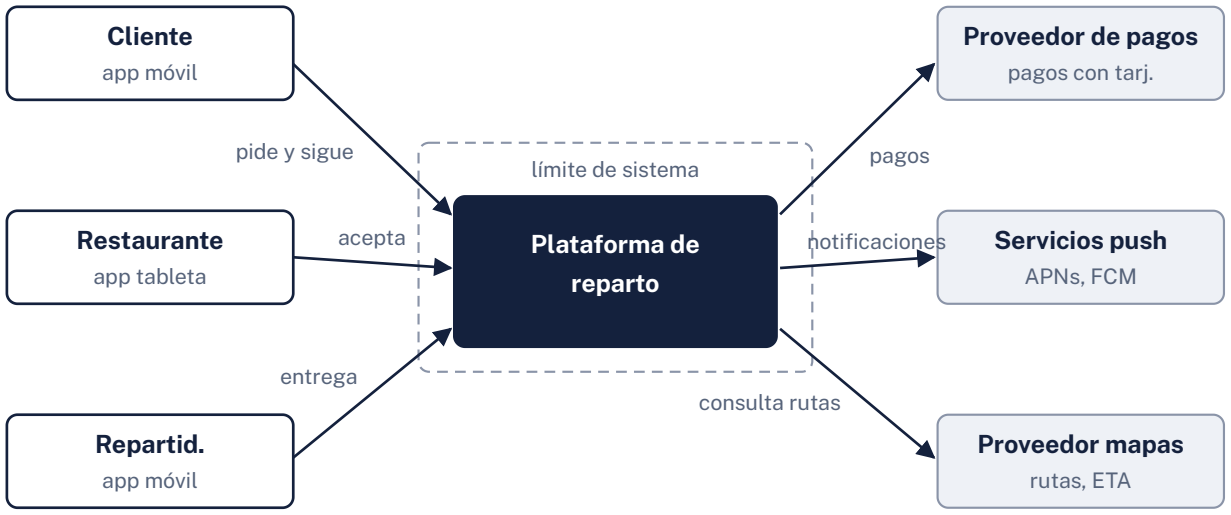
El punto en el que más gente tropieza aquí es que arc42 es un conjunto de cajones etiquetados, no un formulario que rellenar. Muchas secciones deberían contener una breve referencia al lugar donde ya está la información real, no una copia de ella.

- El equipo de reparto de comida ya mantiene dos diagramas C4 (contexto del sistema y contenedores), una bitácora de decisiones en docs/decisions y sus escenarios de calidad en docs/quality.md. Ahora los organiza con arc42.
- Los diagramas van a sus cajones: el diagrama de contexto, a la sección 3, y el diagrama de contenedores, como nivel 1 de la vista de bloques de construcción, a la sección 5.
- Para la sección 9, decisiones de arquitectura, alguien propone pegar el texto completo de cada registro de decisiones, para que todo esté en un solo documento.
- Después, el ADR-0015 reemplaza al ADR-0009, como en la sección anterior. El pull request que añade el ADR-0015 también cambia la bitácora de decisiones, pero nadie se acuerda de la copia pegada, que sigue presentando el ADR-0009 como aceptado.
- Ahora un lector encuentra dos respuestas: la descripción arc42 dice que las notificaciones push pasan por un SDK de terceros, mientras que la bitácora de decisiones dice que el equipo gestiona ahora su propia entrega push. No puede saber cuál es la verdadera, y empieza a desconfiar de toda la descripción.
- Así que la sección 9 contiene un enlace a docs/decisions y los números de unos pocos registros de lectura obligada. Todas las demás secciones funcionan igual: los escenarios de calidad se quedan en docs/quality.md, el despliegue en el repositorio de infraestructura, y una sección sin nada que aportar se elimina o explica por qué en una línea.
- **La lección:** arc42 indica a los lectores dónde buscar; no tiene por qué contenerlo todo. Da a cada dato un único lugar y deja que la plantilla apunte a él, y la descripción seguirá siendo breve y veraz.

Ejemplo práctico: arc42-lite para la plataforma de reparto de comida

La página de abajo es una descripción arc42-lite completa de la plataforma de reparto de comida que se usa a lo largo de este capítulo. Cabe en una página impresa más el diagrama de contexto y, aun así, responde a las preguntas que haría primero un nuevo ingeniero, un responsable de operaciones o un auditor, y señala dónde está el detalle.

Sección 3: contexto de negocio



El contexto de la plataforma: tres grupos de usuarios a la izquierda y tres servicios externos a la derecha, cada flecha etiquetada con lo que se intercambia; los canales y los protocolos pertenecen al contexto técnico.

EJEMPLO PRÁCTICO

Ejemplo práctico: una descripción arc42-lite de una página

Plataforma de reparto de comida, descripción de arquitectura, versión revisada el 2026-10-01 por el arquitecto principal. Las secciones que no aportarían nada se indican como tales en lugar de dejarse vacías.

1. Objetivos	Permitir a los clientes pedir en restaurantes locales y seguir la entrega en tiempo real. Objetivos de calidad por orden de prioridad: (1) ningún pedido se pierde ni se cobra dos veces; (2) p99 de la confirmación de pedidos por debajo de 1.5 s a 120 pedidos/s; (3) se puede añadir un nuevo consumidor de los datos de pedidos sin cambiar el servicio de pedidos. Partes interesadas: clientes, restaurantes, repartidores, operaciones, finanzas, el delegado de protección de datos, el proveedor de pagos.
2. Restricciones	Los datos de tarjeta nunca tocan nuestros sistemas (campos de tarjeta alojados por el proveedor de pagos, para limitar el alcance de PCI DSS). Los datos personales del Reino Unido y de la UE se quedan en regiones de la UE. Cinco squads; el código del lado del servidor, solo en Java, Go o TypeScript.
3. Contexto	Ver el diagrama de contexto. Interfaces externas: API REST y webhooks del proveedor de pagos; APNs y FCM para las notificaciones push; proveedor de mapas para rutas y ETA. Los responsables y los contratos de las interfaces se enumeran en `docs/interfaces.md`.
4. Estrategia de solución	Servicios separados para pedidos, despacho de repartidores y notificaciones (objetivo 3); eventos de pedido asíncronos a través de un intermediario gestionado con una bandeja de salida transaccional (objetivos 1 y 2, ADR-0012); PostgreSQL gestionado para los pedidos (objetivo 1, ADR-0007); claves de idempotencia en cada llamada de pago (objetivo 1).
5. Bloques de construcción	El nivel 1 es el diagrama de contenedores de C4 en `docs/diagrams/containers.dsl`: tres apps cliente, API gateway, servicios de pedidos, de despacho de repartidores y de notificaciones, base de datos de pedidos y el topic de eventos de pedido en el intermediario gestionado. El nivel 2 solo existe para pedidos, el servicio más arriesgado.
6. Ejecución	Dos escenarios: *hacer y pagar un pedido* (diagrama de secuencia) y *timeout del proveedor de pagos* (muestra el reintento con clave de idempotencia y el estado pendiente).
7. Despliegue	Un proveedor de nube, dos regiones de la UE, Kubernetes gestionado y bases de datos gestionadas; definido en el repositorio de infraestructura, que es la fuente de verdad.
8. Conceptos	Modelo de dominio (estados del ciclo de vida del pedido); idempotencia y reintentos; observabilidad (ID de traza transportados en cada evento); tratamiento y conservación de los datos personales.

9. Decisiones	Ver la bitácora de decisiones en `docs/decisions/`. Registros de lectura obligada: ADR-0004, ADR-0007, ADR-0012.
10. Calidad	Escenarios QS-1 a QS-5 en `docs/quality.md`, cada uno con una medida de la respuesta; QS-3 (latencia de confirmación) y QS-1 (ningún pedido perdido) tienen funciones de aptitud automatizadas.
11. Riesgos y deuda	Un único proveedor del intermediario (dependencia del proveedor, aceptada en el ADR-0012); el despacho de repartidores todavía lee directamente la base de datos de pedidos (deuda, eliminación prevista para el T1); límites de tasa del proveedor de mapas en las horas pico (riesgo, responsable: squad de despacho).
12. Glosario	Pedido, carrito, aceptación, recogida, entrega, despacho, ETA, bandeja de salida (outbox), clave de idempotencia: definidos en `docs/glossary.md`.

Cada sección está completada brevemente o apunta a una única fuente de verdad; no se copia nada, así que nada puede desincronizarse.

Los objetivos de calidad de la sección 1 impulsan la estrategia de la sección 4, que cita las decisiones de la sección 9: un lector puede trazar cada decisión importante hasta un objetivo.

Otras plantillas

La plantilla *Views and Beyond* del SEI. El Software Engineering Institute publica una plantilla de documentación que sigue *Documenting Software Architectures: Views and Beyond*, de Paul Clements y sus colegas. Su nivel superior tiene seis partes: una hoja de ruta y una visión general de la documentación, los antecedentes de la arquitectura, las vistas, la correspondencia entre vistas, los materiales de referencia y un glosario con siglas. Cada vista se documenta de la misma manera, con una presentación principal, un catálogo de elementos, un diagrama de contexto, una guía de variabilidad y la justificación (descritos en el recuadro "Para profundizar" de la Sección 4.2). Es más exhaustiva que arc42 y conviene a sistemas grandes, de larga vida o especificados por contrato, en los que cada vista debe estar completa y con referencias cruzadas; su correspondencia entre vistas es un punto fuerte que arc42 deja implícito.

IEEE 1016. IEEE Std 1016-2009, *Software Design Descriptions*, especifica el contenido y la organización de una descripción del diseño de software (SDD), el documento de diseño detallado que tradicionalmente se exige en defensa, en la industria aeroespacial y en otros proyectos regidos por contrato. Tomó de IEEE 1471 (el predecesor de 42010) los conceptos de parte interesada, inquietud, punto de vista y vista, y enumera puntos de vista de diseño como contexto, composición, lógico, dependencia, información, interfaz, interacción, dinámica de

estados, algoritmo y recursos. IEEE la pasó al estado *inactive-reserved* (inactiva reservada) en 2020, pero todavía se cita en contratos y en la docencia, y programas más antiguos siguen entregando SDD con su forma. Está más cerca del diseño detallado que de la arquitectura.

Los documentos de diseño y los RFC, que usan muchas organizaciones de ingeniería, son otro tipo de plantilla: una propuesta para un cambio concreto en lugar de una descripción de todo un sistema. Se tratan dos secciones más adelante, en comunicar la arquitectura.

Conformidad con ISO/IEC/IEEE 42010

ISO/IEC/IEEE 42010:2022 establece requisitos para las descripciones de arquitectura y para los marcos de descripción de arquitectura (ADF): convenciones establecidas, como arc42 o el enfoque Views and Beyond, para producir descripciones en una comunidad determinada. No prescribe una plantilla, una notación ni un proceso. Una descripción es *conforme* cuando contiene la información que exige la norma, sea cual sea la plantilla que la organice.

En términos generales, eso significa identificar el sistema descrito y su entorno, identificar las partes interesadas y sus inquietudes, indicar qué puntos de vista se usan y qué inquietudes enmarca cada uno, proporcionar vistas que sigan esos puntos de vista, registrar las correspondencias (cómo se relacionan los elementos de distintas vistas) y registrar la justificación de las decisiones clave. arc42 no afirma ser conforme de serie, y las descripciones arc42 sin más a menudo carecen de un vínculo explícito entre cada inquietud y la vista que la aborda. Las carencias son pequeñas y fáciles de cubrir, como muestra la tabla. La mayoría de los equipos no necesitan una conformidad formal; los de sectores regulados o vinculados por contrato a veces sí, y es tranquilizador que una plantilla ligera pueda cumplir la norma con unos pocos añadidos.

Cómo hacer que una descripción arc42 sea conforme con 42010

Lo que pide 42010	Dónde lo tiene arc42	Qué añadir
Las partes interesadas y sus inquietudes	Tabla de partes interesadas de la sección 1.3; objetivos de calidad en 1.2 y 10	Enumera las inquietudes de forma explícita por parte interesada, no solo las expectativas.
Puntos de vista, cada uno con las inquietudes concretas que enmarca	Implícitos en las secciones 3, 5, 6 y 7	Una tabla breve que nombre cada punto de vista, las inquietudes que enmarca y su notación (por ejemplo, diagramas de contenedores de C4).
Vistas que siguen los puntos de vista	Secciones 3, 5, 6 y 7	Nada, si cada vista indica qué punto de vista sigue.
Correspondencias entre vistas	En parte en la sección 7 (qué se ejecuta dónde)	Tablas de correspondencia, por ejemplo, de bloque de construcción a nodo de despliegue, o de bloque de construcción a equipo responsable.
Justificación de las decisiones clave	Secciones 4 y 9 (ADR)	Vincula cada decisión con las inquietudes que aborda.

ATENCIÓN

Errores comunes con las plantillas

- Completar todas las secciones porque existen. Una sección vacía con una línea que explique por qué es mejor que una página de relleno.
- Copiar requisitos, decisiones y detalles de infraestructura en la plantilla, donde se desincronizan de los originales. Enlaza en su lugar.
- Una vista de bloques de construcción con decenas de cajas en el nivel 1. El nivel 1 debería caber en una pantalla; el detalle corresponde a los niveles inferiores, y solo donde compensa.
- Escribir la descripción una vez al principio y nunca más. Ponla bajo control de versiones, revísala con los cambios significativos y muestra una fecha de “última revisión”.
- Mezclar plantillas sin un mapa. Si usas arc42 para la estructura y C4 para los diagramas, dilo en la sección 1, para que los lectores conozcan las convenciones.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Qué sección de arc42 suele convertirse en la más extensa y cómo evitas que crezca sin control?

La sección 5, la vista de bloques de construcción. Limita el nivel 1 a un diagrama que quepa en una pantalla y refina a los niveles 2 y 3 solo los bloques de construcción complejos o arriesgados.

▼ ¿Dónde aparecen los objetivos de calidad en arc42 y en qué se diferencian las secciones 1 y 10?

La sección 1 enumera los tres a cinco objetivos de calidad principales por orden de prioridad para orientar; la sección 10 los refina, junto con cualquier otra cualidad, en un árbol de calidad y escenarios concretos y comprobables.

▼ Tu equipo ya guarda los ADR en el repositorio. ¿Qué debería contener la sección 9 de arc42?

Un enlace a la bitácora de decisiones, quizá con una breve lista de las decisiones que todo recién llegado debe leer, en lugar de una copia de los registros.

▼ ¿Usar arc42 hace que una descripción de arquitectura sea conforme con ISO/IEC/IEEE 42010?

No automáticamente. La conformidad depende del contenido: las partes interesadas, las inquietudes, los puntos de vista, las vistas, las correspondencias y la justificación deben ser explícitos, lo que normalmente requiere unos pocos añadidos pequeños a arc42.

▼ Inténtalo: Un equipo de tres personas gestiona un pequeño servicio de cartas de restaurantes. Su responsable les dice que rellenen las doce secciones de arc42. ¿Qué aconsejarías en su lugar y cómo deberían tratarse las secciones que no necesitan?

Usa una versión ligera de arc42. Escribe de tres a cinco escenarios de calidad, el contexto, una breve estrategia de solución, los bloques de construcción de nivel 1 y los conceptos transversales clave, idealmente en una página aproximadamente. Enlaza con los ADR y el código de infraestructura en lugar de copiarlos. Todas las secciones son opcionales, así que indica deliberadamente que una sección no hace falta, en lugar de dejarla en blanco o rellenarla sin más. arc42 dice dónde va la información; nunca pretendió exigir volumen.

FUENTES

- arc42 overview: https://arc42.org/overview
- arc42 documentation: section 1, Introduction and Goals: https://docs.arc42.org/section-1/

- arc42 FAQ: What is the minimal amount of an arc42 documentation?: https://faq.arc42.org/questions/B-4/
- SEI Software Architecture Documentation Template (Views and Beyond): https://wiki.sei.cmu.edu/confluence/display/SAD/Software+Architecture+Documentation+Template
- IEEE 1016-2009: Software Design Descriptions: https://standards.ieee.org/ieee/1016/4502/

SECCIÓN 4.7 DE 4.8 · INTERMEDIO · 22 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Diagramas como código y cómo mantener la documentación al día

ANTES DE LEER ESTO

Esta sección se basa en:

- 0.8 Cómo se hace el software
- 4.1 Por qué documentar
- 4.3 El modelo C4

Términos anteriores que utiliza: Control de versiones, Pull request, Integración continua (CI), Diagrama de secuencia, Documentación viva

LA IDEA EN BREVE

Imagina un libro de recetas familiar en el que cada cambio lo apunta a lápiz quien cocina, sin dejar constancia de quién cambió qué. Al cabo de unos años, nadie se fía de él. Ahora imagina que las recetas están mecanografiadas, que cada cambio se propone, lo revisa un segundo cocinero y se fecha, y que el libro impreso se reimprime automáticamente después de cada cambio aprobado. Los diagramas como código aplican esa disciplina a los diagramas de arquitectura: se escriben como archivos de texto breves, se guardan junto al software, se revisan cuando el software cambia y se vuelven a dibujar automáticamente.

LO QUE NECESITAS SABER

- Diagramas como código significa escribir diagramas en un lenguaje de texto y dejar que una herramienta los dibuje, para que puedan versionarse, compararse línea a línea, revisarse en pull requests y renderizarse en el pipeline de compilación.
- PlantUML (2009) destaca sobre todo en UML, en especial en los diagramas de secuencia; Mermaid (2014) se renderiza directamente en Markdown en GitHub y GitLab; D2 (2022) ofrece disposiciones limpias; Structurizr DSL define un único modelo C4 a partir del cual se generan muchas vistas.
- Las herramientas de dibujo solo conocen formas, así que un mismo elemento puede divergir de un diagrama a otro; las herramientas de modelado guardan los elementos y las relaciones en un modelo, de modo que todas las vistas se mantienen coherentes y el modelo puede consultarse y comprobarse.

- Un pipeline de documentación como código renderiza los diagramas, comprueba los enlaces, genera el sitio de documentación en cada pull request y publica desde main, de modo que los lectores ven una documentación que coincide con el código.
- La documentación se mantiene al día gracias a hábitos, no a herramientas: responsables con nombre, revisión junto con los cambios de código, fechas de última revisión, enlaces entre documentos, código y decisiones, y eliminación de las páginas obsoletas (los ADR se reemplazan, no se eliminan).

Qué significa diagramas como código

Diagramas como código significa escribir un diagrama como un archivo de texto breve en un lenguaje pequeño y dejar que una herramienta lo organice y lo dibuje. En lugar de arrastrar cajas sobre un lienzo, escribes `ordering → db: SQL` y la herramienta produce la imagen. El archivo de texto vive en el repositorio de código y se trata como código: se versiona, se revisa, lo comprueban las compilaciones automáticas y lo publica el pipeline.

Es una parte de la documentación como código (docs as code), la práctica, promovida por la comunidad Write the Docs, de producir documentación con las mismas herramientas y el mismo flujo de trabajo que el software: gestores de incidencias, control de versiones (normalmente Git), lenguajes de marcado de texto plano como Markdown o AsciiDoc, revisión de código y pruebas automáticas. Las secciones anteriores produjeron el material: vistas y diagramas C4, UML donde importa la precisión, ADR y una estructura arc42. Esta sección trata de mantener ese material exacto durante toda la vida del sistema, que es donde fracasa la mayor parte de la documentación de arquitectura. Las razones de ese fracaso se expusieron en por qué y cuánto documentar: wikis desactualizadas, grandes documentos sin responsable y diagramas que ya no coinciden con el sistema.

Por qué poner los diagramas en texto

Control de versiones. Un diagrama en texto tiene historial. Puedes ver quién lo cambió, cuándo y por qué, y comparar dos versiones cualesquiera línea a línea; una imagen binaria o un archivo en el formato de una herramienta de dibujo solo muestra que algo cambió. También puedes crear ramas: el diagrama de un cambio propuesto vive en la misma rama que el código que lo implementa.

Revisión en pull requests. Cuando un desarrollador añade una nueva dependencia entre servicios, el cambio en el diagrama aparece en el mismo pull request que el código, como una diferencia de texto legible. Los revisores pueden preguntar si esa nueva flecha es deseada, lo que supone una revisión de arquitectura en el momento en que resulta más barata.

Coherencia. La herramienta aplica las mismas formas, fuentes y reglas de disposición a todos los diagramas, de modo que los diagramas dibujados por veinte personas siguen pareciendo un solo conjunto. Con un lenguaje de modelado (más adelante), el mismo elemento aparece con el mismo nombre y la misma descripción en todas las vistas.

Generación. El texto puede producirse con programas. Los diagramas pueden generarse a partir del propio código (grafos de dependencias), de la infraestructura como código (diagramas de despliegue), de las especificaciones de API o de un modelo central, y luego comprobarse automáticamente con el sistema.

Los compromisos son reales. La disposición automática es cómoda, pero a veces fea, y pelearse con un motor de disposición para colocar una caja puede llevar más tiempo que dibujarla; la mayoría de las herramientas ofrecen indicaciones como la dirección o la agrupación, no posiciones exactas. Los lenguajes de texto tienen una curva de aprendizaje para quienes no programan, como los responsables de producto. Y un buen boceto en una pizarra durante una discusión de diseño sigue siendo la forma más rápida de pensar; los diagramas como código son para los diagramas que piensas conservar.

Cuatro lenguajes, un ejemplo

Cuatro lenguajes dominan el panorama. Cada ejemplo de abajo describe una parte de la misma plataforma de reparto de comida.

PlantUML, creado por Arnaud Roques y lanzado por primera vez en 2009, es el más antiguo de los cuatro. Cubre todos los diagramas UML habituales (de secuencia, de componentes, de despliegue, de estados, de actividades, de clases) y muchos otros, y usa Graphviz para organizar la mayoría de los tipos de diagrama. La biblioteca C4-PlantUML añade las formas de C4. Sus diagramas de secuencia son especialmente buenos, lo que lo convierte en una opción natural para los escenarios de ejecución de Lo esencial de UML.

```
@startuml
title Place an order (happy path)
actor Customer
participant "Customer app" as App
participant "Ordering service" as Ord
participant "Payment provider" as Pay
queue "Order events topic" as MB

Customer -> App : Tap "Pay"
App -> Ord : POST /orders (HTTPS)
Ord -> Pay : Authorise payment
Pay --> Ord : Authorised
Ord ->> MB : OrderPlaced (async)
Ord --> App : 201 Created
@enduml
```

PlantUML: un diagrama de secuencia de hacer un pedido; `->>` dibuja un mensaje asíncrono y `-->` una respuesta

Mermaid, creado por Knut Sveidqvist en 2014, está escrito en JavaScript y se renderiza en el navegador. Su mayor ventaja es su alcance: GitHub renderiza los bloques de Mermaid de los archivos Markdown desde febrero de 2022, y GitLab y muchas herramientas de wiki y de toma de notas hacen lo mismo, así que un diagrama en un README no necesita ningún paso de compilación. Admite diagramas de flujo, de secuencia, de clases, de estados y de entidad-relación, entre otros, y tiene soporte experimental para C4. Su control de la disposición es limitado en los diagramas grandes.

```
flowchart LR
    app["Customer app<br/>Kotlin, Swift"]
    gw["API gateway"]
    ord["Ordering service<br/>Java"]
    db[("Order database<br/>PostgreSQL")]
    topic[["Order events<br/>Kafka topic"]]
    disp["Dispatch service<br/>Go"]
    notif["Notification service<br/>Node.js"]

    app -->|HTTPS/JSON| gw
    gw -->|HTTPS/JSON| ord
    ord -->|SQL| db
    ord -->|publishes OrderPlaced| topic
    disp -->|consumes| topic
    notif -->|consumes| topic
```

Mermaid: un diagrama de flujo a nivel de contenedores; `[( )]` dibuja el cilindro de una base de datos y las etiquetas van sobre las flechas; el topic se dibuja como un contenedor, como recomienda la Sección 4.3

Structurizr DSL, creado por Simon Brown junto con el modelo C4, es de otra naturaleza. No describe un diagrama; describe un *modelo* (personas, sistemas de software, contenedores, componentes y las relaciones entre ellos) y después define *vistas* de ese modelo. Cada elemento se define una sola vez, con su nombre, su descripción y su tecnología, y aparece en todas las vistas que lo incluyen. Las relaciones entre contenedores se deducen automáticamente en el nivel del sistema, así que la vista de contexto no necesita flechas aparte. El DSL también puede adjuntar documentación en Markdown y una carpeta de ADR al workspace, y sus herramientas pueden exportar las vistas a PlantUML, Mermaid, D2 y otros formatos.

```
workspace "Food delivery" {
    model {
        customer = person "Customer" "Orders food from local restaurants."
        payments = softwareSystem "Payment provider" "Takes card payments." "External"
        platform = softwareSystem "Food-delivery platform" {
            app = container "Customer app" "Ordering and tracking." "Kotlin, Swift"
            gateway = container "API gateway" "Authenticates and routes requests."
"Managed API gateway"
            ordering = container "Ordering service" "Accepts and tracks orders." "Java"
            db = container "Order database" "Orders and their states." "PostgreSQL"
"Database"
            orderEvents = container "Order events" "Order lifecycle events." "Kafka
topic" "Topic"
            dispatch = container "Dispatch service" "Assigns couriers." "Go"
        }
        customer → app "Places and tracks orders using"
        app → gateway "Makes API calls to" "JSON/HTTPS"
        gateway → ordering "Routes requests to" "JSON/HTTPS"
        ordering → db "Reads from and writes to" "SQL/TCP"
        ordering → payments "Takes card payments using" "JSON/HTTPS"
        ordering → orderEvents "Publishes order events to" "Kafka"
        dispatch → orderEvents "Consumes order events from" "Kafka"
    }
    views {
        systemContext platform "Context" {
            include *
            autoLayout lr
        }
        container platform "Containers" {
            include *
            autoLayout lr
        }
        styles {
            element "Database" {
                shape cylinder
            }
            element "Topic" {
                shape pipe
            }
        }
    }
}
```

Structurizr DSL: un modelo y dos vistas de la plataforma de reparto de comida; la vista de contexto se genera a partir de las mismas definiciones

D2 (Declarative Diagramming), cuyo código abrió Terrastruct en 2022, es el más reciente. Busca un resultado limpio y moderno y una sintaxis legible, admite contenedores anidados (escritos con llaves y referenciados con nombres separados por puntos) y ofrece varios

motores de disposición a elegir: dagre y ELK vienen incluidos y son gratuitos, y TALA, diseñado para diagramas de arquitectura de software, es un motor comercial aparte.

```
direction: right

customer: Customer {shape: person}
payments: Payment provider

platform: Food-delivery platform {
  app: Customer app
  ordering: Ordering service
  db: Order database {shape: cylinder}
  events: Order events topic {shape: queue}
  dispatch: Dispatch service

  app -> ordering: Places order (HTTPS)
  ordering -> db: SQL
  ordering -> events: Publishes OrderPlaced
  dispatch -> events: Consumes OrderPlaced
}

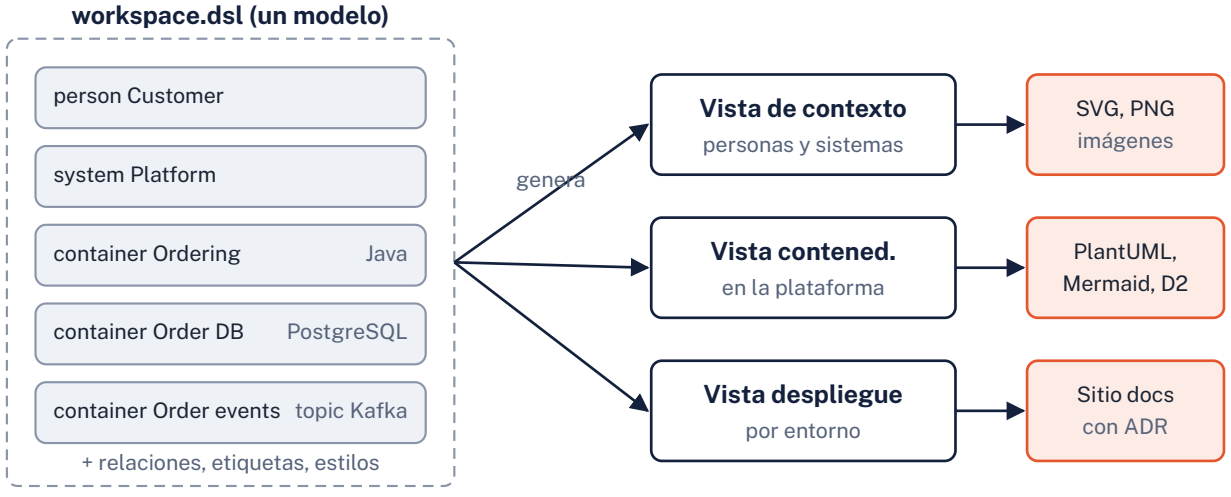
customer -> platform.app: Places orders using
platform.ordering -> payments: Authorise (HTTPS)
```

D2: contenedores anidados, formas y conexiones con etiquetas

Comparación de los cuatro

	PlantUML	Mermaid	Structurizr DSL	D2
Describe	Un diagrama por archivo	Un diagrama por bloque	Un modelo más vistas de él	Un diagrama por archivo
Destaca en	UML, en especial los diagramas de secuencia	Diagramas dentro de Markdown sin paso de compilación	Diagramas C4 coherentes en varios niveles	Diagramas de arquitectura generales y atractivos
Renderizado	Herramienta o servidor Java; Graphviz para la mayoría de las disposiciones	En el navegador; nativo en GitHub y GitLab	Herramientas de Structurizr; exporta a otros formatos	Herramienta de línea de comandos; motor de disposición a elegir
Cuidado con	Aspecto predeterminado anticuado; disposición controlable solo mediante indicaciones	Control limitado de la disposición en diagramas grandes	Solo conceptos de C4; necesita sus propias herramientas	El mejor motor de disposición es comercial

Un modelo, muchas vistas



Renombra Ordering una vez en el modelo y todas las vistas y exportaciones cambian.

Un lenguaje de modelado define cada elemento una sola vez; las vistas y las exportaciones se generan, así que no pueden contradecirse entre sí.

Herramientas de dibujo y herramientas de modelado

La diferencia entre Structurizr y los otros tres es un caso de la distinción entre herramientas de dibujo y de modelado presentada en la Sección 4.3. Una herramienta de dibujo (diagrams.net, Excalidraw, Visio, una pizarra en línea, y también PlantUML, Mermaid y D2) conoce formas, líneas y texto. Una caja con la etiqueta “Servicio de pedidos” en un diagrama no tiene ninguna conexión con la caja que lleva la misma etiqueta en otro. Si renombras el servicio, tienes que buscar y corregir a mano cada diagrama; si lo escribes mal una vez, tienes, hasta donde sabe la herramienta, dos servicios.

Una herramienta de modelado (Structurizr, Archi para modelos ArchiMate, Sparx Enterprise Architect para UML y SysML, Capella) mantiene un modelo: un repositorio de elementos y relaciones con propiedades. Los diagramas son vistas del modelo. Si renombras un elemento, cambian todas las vistas. El modelo puede consultarse (“¿qué contenedores dependen del intermediario de mensajes?”), comprobarse (“ningún componente de la app del cliente puede llamar a la base de datos”) y compararse con el sistema en ejecución. El precio es más estructura por adelantado y, con las herramientas comerciales pesadas, licencias, formación y una tendencia a modelar mucho más de lo que nadie lee.

Una regla práctica: usa herramientas de dibujo, de texto o gráficas, para bocetos, explicaciones puntuales y diagramas que se sostienen por sí solos, como un único diagrama de secuencia en un ADR. Usa un modelo cuando los mismos elementos aparezcan en muchos diagramas, cuando contribuyan varios equipos o cuando quieras comprobar la documentación con el código. Muchos equipos combinan ambos: un modelo de Structurizr para la estructura C4 y PlantUML o Mermaid para unos pocos diagramas de secuencia y de estados.

SI ESTO ES NUEVO PARA TI

Paso a paso: un cambio de nombre, tres archivos

La idea más difícil de esta sección es que escribir los diagramas como texto no basta, por sí solo, para mantenerlos coherentes. La Sección 4.3 dividió las herramientas en herramientas de dibujo y herramientas de modelado, y esta sección añade que PlantUML, Mermaid y D2 son herramientas de dibujo, aunque se escriban con el teclado, y que solo una herramienta de modelado como Structurizr sabe que dos cajas son la misma cosa.

- El equipo de la plataforma de reparto de comida mantiene tres diagramas como texto: el diagrama de secuencia en PlantUML de hacer un pedido, el diagrama de flujo de contenedores en Mermaid y el diagrama en D2. Cada archivo escribe por su cuenta la etiqueta 'Ordering service'.
- Supón que el equipo cambia el nombre del servicio a 'Order service'. Alguien tiene que encontrar cada archivo que lo menciona y editar cada etiqueta a mano, porque nada une las tres cajas: son solo texto que casualmente coincide.
- Si se te escapa un archivo, los diagramas no coinciden. Si escribes mal una etiqueta, por ejemplo 'Order servce', para esa herramienta es un servicio distinto. El pipeline sigue renderizando todos los diagramas, porque cada archivo es válido por sí solo.
- En el archivo de Structurizr DSL, el servicio de pedidos se define una sola vez, en el modelo, con su nombre, su descripción y su tecnología. La vista de contenedores, y cualquier otra vista que lo incluya, se genera a partir de esa única definición.
- Cambia el nombre ahí y todas las vistas lo recogen en la siguiente compilación. Como el modelo contiene elementos y relaciones, también se le puede preguntar qué contenedores dependen del topic de eventos de pedidos, o comprobar con una regla.
- **La lección:** el texto da a los diagramas historial, revisión en pull requests y renderizado automático; solo un modelo hace que el mismo elemento signifique lo mismo en todas partes. Usa uno cuando las mismas piezas aparezcan en muchos diagramas.

Generar la documentación en el pipeline

Una vez que la documentación es texto en el repositorio, el pipeline de integración continua del Capítulo 0 puede compilarla y comprobarla como si fuera código. Un pipeline típico renderiza todos los diagramas (y hace fallar la compilación ante un error de sintaxis), comprueba los enlaces, genera un sitio de documentación estático con un generador como MkDocs, Antora o Docusaurus, y lo publica cuando los cambios llegan a la rama principal. Así, los lectores siempre ven la documentación que coincide con el código de main, y nadie tiene que acordarse de exportar una imagen y subirla a una wiki.

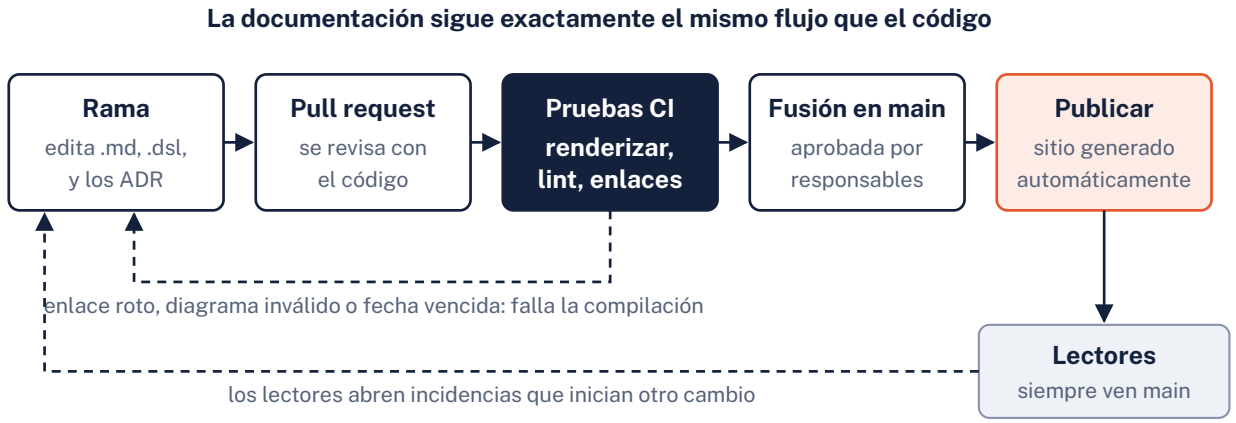
Se pueden añadir más comprobaciones a medida que el equipo madura: un corrector ortográfico y un linter de estilo de redacción (Vale es popular), una comprobación de que cada ADR tiene un estado válido, un aviso cuando la fecha de "última revisión" de una página es más antigua que su intervalo de revisión, y comprobaciones de arquitectura que comparan las dependencias documentadas con las reales del código, que son una forma de función de aptitud (fitness function). Algunos equipos generan partes de la documentación a partir del propio sistema: grafos de dependencias a partir de la compilación, la referencia de la API a partir de las especificaciones OpenAPI, diagramas de despliegue a partir de la infraestructura como código.

```yaml
name: architecture-docs
on:
  pull_request:
    paths: ["docs/**"]
  push:
    branches: [main]

jobs:
  docs:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      # assumes plantuml, d2, lychee and mkdocs are available on the runner
      - name: Render diagrams (fails on syntax errors)
        run: |
          plantuml -tsvg docs/diagrams/*.puml
          for f in docs/diagrams/*.d2; do d2 "$f" "${f%.d2}.svg"; done
      - name: Check links
        run: lychee --offline docs/
      - name: Build site (warnings such as broken links fail the build)
        run: mkdocs build --strict
      - name: Publish
        if: github.ref == 'refs/heads/main'
        run: ./scripts/publish-docs.sh
```

Un flujo de trabajo de GitHub Actions que renderiza los diagramas, comprueba los enlaces, genera el sitio de documentación en cada pull request y publica desde main

El flujo de trabajo de la documentación como código



Los cambios en la documentación siguen el mismo camino que los cambios en el código: rama, revisión, comprobaciones automáticas, fusión y publicación automática.

Mantener la documentación al día

Las herramientas no mantienen la documentación fiel a la realidad; los hábitos sí. Cinco hábitos marcan la diferencia.

Responsabilidad. Cada documento tiene un responsable con nombre, normalmente el equipo dueño del código que describe. En GitHub y GitLab, un archivo `CODEOWNERS` puede convertir a ese equipo en revisor obligatorio de los cambios bajo su carpeta `docs/`. Los documentos que son de “todos” no los mantiene nadie.

Revisión con los cambios de código. Una plantilla de pull request que pregunta “¿Cambia esto una interfaz, una dependencia, un despliegue o una decisión? Si es así, ¿qué documento o ADR has actualizado?” detecta la mayor parte de la deriva en el momento en que se produce. Algunos equipos van más allá y bloquean la fusión de funcionalidades que carecen de documentación, una práctica que menciona la guía de Write the Docs.

Fechas de última revisión. Un pequeño encabezado en cada página registra su responsable, la fecha en que se comprobó por última vez con la realidad y cada cuánto debe revisarse. Los lectores pueden juzgar hasta qué punto fiarse de ella, y el pipeline puede señalar las páginas con la revisión vencida. Una página que dice “última revisión: 2023” es honesta; una página que describe en silencio el sistema de 2023 es una trampa.

Enlazar documentación y código. Las páginas apuntan al código que describen (rutas, módulos), los comentarios del código apuntan a los ADR y los mensajes de commit citan los registros que implementan. Los enlaces en ambas direcciones dejan claro a qué documentos podría afectar un cambio.

Eliminar el material obsoleto. Una documentación errónea es peor que ninguna, porque los lectores actúan según ella. Cuando una página ya no describe el sistema y nadie va a corregirla, elimínala o muévela a un archivo histórico claramente señalado. Los ADR son la excepción: son registros de la historia, así que se reemplazan o se declaran obsoletos, no se eliminan. Esta es la idea de la documentación viva que Cyrille Martraire desarrolló en su libro de 2019 con ese título: una documentación que evoluciona al mismo ritmo que el sistema, idealmente porque gran parte de ella se deriva del propio sistema.

```
---
title: "Ordering service: building block view (level 2)"
owner: "@ordering-team"
last_reviewed: 2026-09-14
review_every: 6 months
code: src/ordering/
decisions: [ADR-0007, ADR-0012]
---
```

Front matter al principio de una página Markdown: responsable, fecha e intervalo de revisión, y enlaces al código y a las decisiones que describe

Lista de comprobación: documentación que se mantiene al día

Úsala al preparar la documentación de arquitectura de un repositorio y vuelve a ella cada pocos meses.

- [ ] **La documentación vive en el repositorio que describe**
  Páginas Markdown o AsciiDoc, fuentes de diagramas y ADR bajo `docs/`, no en una wiki aparte que nadie actualiza en el mismo cambio.
- [ ] **Las fuentes de los diagramas son texto y las renderiza el pipeline**
  Nada de imágenes exportadas a mano que se quedan atrás en silencio respecto a sus fuentes; la compilación falla ante un diagrama no válido.
- [ ] **Cada elemento se define una sola vez donde los diagramas se solapan**
  Usa un modelo (como Structurizr DSL) cuando los mismos contenedores aparecen en varios diagramas.
- [ ] **Cada página tiene un responsable y una fecha de última revisión**
  Lo garantizan `CODEOWNERS` y una comprobación del pipeline que avisa cuando ha pasado el intervalo de revisión.
- [ ] **Los pull requests preguntan por la documentación**
  La plantilla pregunta si ha cambiado una interfaz, una dependencia, un despliegue o una decisión, y qué documento se ha actualizado.
- [ ] **Los enlaces se comprueban automáticamente**
  Los enlaces internos rotos y las referencias a archivos eliminados hacen fallar la compilación.
- [ ] **La documentación apunta al código y el código apunta a las decisiones**
  Rutas en la documentación, números de ADR en los comentarios del código y en los mensajes de commit.
- [ ] **El sitio publicado se genera desde main**
  Los lectores siempre ven la versión que coincide con el código desplegado, sin ningún paso de publicación manual.
- [ ] **Las páginas obsoletas se eliminan o se archivan**
  Las páginas que nadie va a corregir se retiran; los ADR se reemplazan o se declaran obsoletos, nunca se eliminan.
- [ ] **Al menos una comprobación compara la documentación con la realidad**
  Por ejemplo, una prueba que falla si aparece en el código una dependencia que el diagrama de contenedores no muestra.

ATENCIÓN

Errores comunes

- Elegir una herramienta antes de decidir qué documentar. La herramienta importa mucho menos que los hábitos de responsabilidad y revisión.
- Ponerlo todo en una única fuente de diagrama gigantesca que nadie puede leer ni como texto ni como imagen.
- Hacer commit de imágenes renderizadas sin sus fuentes, o de fuentes sin una compilación que las renderice, de modo que unas y otras se separan.
- Tomar los diagramas como código como excusa para saltarse leyendas, títulos y etiquetas: las reglas de notación de C4 se siguen aplicando a los diagramas generados.
- Hacer opcionales los cambios de documentación en la revisión, de modo que siempre son lo primero que se abandona bajo la presión de los plazos.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Qué puede ver un revisor en un pull request cuando un diagrama es código que no puede ver cuando es una imagen exportada?

Una diferencia legible, línea a línea, de lo que ha cambiado, como una nueva relación entre dos servicios, en el mismo pull request que el código que la introdujo.

▼ ¿Por qué renombrar un servicio da más trabajo con una herramienta de dibujo que con una herramienta de modelado?

En una herramienta de dibujo, cada diagrama guarda su propia copia de la etiqueta, así que hay que encontrar y editar cada diagrama; en una herramienta de modelado, el elemento se define una sola vez y todas las vistas recogen el nuevo nombre.

▼ ¿Por qué hay que eliminar la documentación obsoleta en lugar de dejarla donde está, y cuál es la excepción?

Los lectores actúan según lo que leen, así que una documentación errónea provoca decisiones erróneas. Los ADR son la excepción: registran la historia y, en lugar de eliminarse, se marcan como reemplazados u obsoletos.

▼ **Nombra dos comprobaciones que un pipeline de documentación puede ejecutar además de renderizar diagramas.**

Comprobación de enlaces, linting de redacción u ortografía, una compilación estricta del sitio, un aviso para las fechas de revisión vencidas y la comparación de las dependencias documentadas con las del código.

▼ **Inténtalo: Tu equipo tiene ocho diagramas en una herramienta de dibujo. Tras renombrar un servicio, tres siguen mostrando el nombre antiguo y una flecha de despacho difiere entre dos. ¿Qué cambiarías, en herramientas y en hábitos, y por qué?**

Las herramientas de dibujo solo conocen formas, de modo que cada imagen deriva por separado. Pasa a un modelo, como Structurizr DSL, guardado en el repositorio: define cada elemento una vez y genera las vistas, de modo que un cambio de nombre llegue a todos los diagramas. Renderízalos en el pipeline en cada pull request. Las herramientas por sí solas no bastan, así que nombra a un responsable, revisa la documentación con los cambios de código, muestra una fecha de última revisión y elimina las páginas obsoletas. PlantUML o Mermaid servirían para diagramas de secuencia puntuales.

FUENTES

- Docs as Code, Write the Docs: https://www.writethedocs.org/guide/docs-as-code/
- Structurizr DSL language reference: https://docs.structurizr.com/dsl/language
- Mermaid documentation: https://mermaid.js.org/
- PlantUML: https://plantuml.com/
- Include diagrams in your Markdown files with Mermaid, GitHub Blog (2022): https://github.blog/2022-02-14-include-diagrams-markdown-files-mermaid/

SECCIÓN 4.8 DE 4.8 · INTERMEDIO · 24 MIN DE LECTURA · ÚLTIMA REVISIÓN: NOVIEMBRE DE 2026

Comunicar la arquitectura a distintos públicos

ANTES DE LEER ESTO

Esta sección se basa en:

- 1.5 Partes interesadas e inquietudes
- 1.6 Decisiones y compromisos
- 4.3 El modelo C4

Términos anteriores que utiliza: Compromiso, Inquietud, Modelo C4, Matriz de decisión ponderada, Último momento responsable

LA IDEA EN BREVE

Un guía turístico explica la misma catedral de forma distinta a escolares, arquitectos y peregrinos: datos distintos, palabras distintas, recorridos distintos por el edificio. Los arquitectos de software tienen que hacer lo mismo. Una directora financiera quiere saber cuánto cuesta un diseño y qué podría salir mal; un desarrollador quiere saber cómo se conectan las piezas; un auditor quiere pruebas. La buena comunicación empieza por preguntarse quién escucha y qué tiene que decidir, y después le muestra justo eso, con claridad y honestidad, incluidos los inconvenientes.

LO QUE NECESITAS SABER

- Comunicar la arquitectura es una tarea arquitectónica central; empieza toda explicación nombrando al público y lo que deberá decidir o hacer después.
- Los directivos necesitan la recomendación, el costo, el riesgo y la decisión que se les pide; los desarrolladores necesitan la estructura, las interfaces y las razones; operaciones, seguridad y los auditores necesitan, cada uno, vistas de los fallos, de los límites de confianza y de las evidencias.
- Explica los compromisos como opciones, criterios, recomendación y consecuencias, empezando por la respuesta y diciendo con claridad a qué se renuncia.
- Un resumen de arquitectura en una página y una historia numerada que sigue una solicitud hacen comprensible cualquier sistema; etiqueta cada caja y cada línea al hacer un boceto, y deja constancia de las conclusiones después de la sesión en la pizarra.
- Los documentos de diseño y los RFC proponen un cambio por escrito para recibir comentarios antes de construirlo; después, los ADR registran las decisiones de forma permanente.

> - Los diagramas claros tienen un título, una leyenda, una notación coherente, pocos colores y con significado, una única dirección de flujo y flechas etiquetadas; los desacuerdos se resuelven acordando criterios, reuniendo evidencias, buscando un consenso aproximado y dejando constancia de las discrepancias.

Comunicar es el trabajo

Una arquitectura que solo entiende su arquitecto ha fracasado, por buena que sea. Las decisiones determinan lo que decenas de personas construyen, operan, financian y auditan, y solo funcionan si esas personas las entienden lo bastante bien como para actuar en consecuencia y para darse cuenta de cuándo la realidad se aparta de ellas. Por eso el plan de estudios de iSAQB incluye la comunicación de la arquitectura entre las tareas centrales de un arquitecto, y por eso el rol del arquitecto descrito en el Capítulo 1 depende tanto de las habilidades interpersonales.

Las secciones anteriores de este capítulo aportaron el material: vistas, diagramas C4 y UML, ADR, una estructura arc42 y un pipeline que los mantiene al día. Esta sección trata de usar ese material con las personas. La habilidad central es la misma en cualquier situación: decide quién es el público, decide qué tiene que hacer después de escucharte y dale exactamente lo que eso requiere, en la forma en que pueda asimilarlo. La imagen del *ascensor del arquitecto* (architect elevator) de Gregor Hohpe (*The Software Architect Elevator*, 2020) recoge esa amplitud: un arquitecto sube y baja entre la sala del consejo, donde se fija la estrategia en términos de negocio, y la sala de máquinas, donde se construyen los sistemas, y tiene que hacerse entender en cada planta.

Adaptarse al público

Cada público tiene sus propias inquietudes, exactamente en el sentido de partes interesadas e inquietudes, y su propio vocabulario y capacidad de atención. La tabla es un punto de partida; la verdadera habilidad consiste en preguntarse, antes de preparar nada, “¿qué decidirá o hará de forma distinta esta persona después de esta conversación?”

Lo que necesitan los distintos públicos

Público	Lo que le importa	Qué mostrar	Qué evitar
Directivos y patrocinadores	Costo, riesgo, tiempo de salida al mercado, opciones estratégicas, lo que deben decidir	Una página: la recomendación, el impacto en el negocio, el costo, los principales riesgos y la decisión que necesitas de ellos; como mucho, un diagrama de contexto del sistema	Nombres de tecnologías sin significado para el negocio; diagramas de componentes; un repaso de las opciones antes de la respuesta
Responsables de producto	Lo que se vuelve posible o más difícil, las fechas de entrega, el comportamiento visible para el usuario	Capacidades y restricciones en términos de producto; cómo afecta la arquitectura a la hoja de ruta; una historia en ejecución de un recorrido de usuario	La estructura interna por sí misma
Desarrolladores	Cómo construir dentro de la arquitectura: límites, interfaces, convenciones, por qué las cosas son como son	Diagramas C4 de contenedores y de componentes, diagramas de secuencia, ADR, ejemplos de código, la bitácora de decisiones	Principios vagos sin ejemplos; diagramas que no coinciden con el código
Operaciones y SRE	Despliegue, modos de fallo, monitorización, capacidad, recuperación	Diagramas de despliegue, escenarios de fallo en ejecución, SLO, runbooks, mapas de dependencias	Flujos que solo muestran el camino feliz
Seguridad	Límites de confianza, flujos de datos, identidades, superficie de ataque	Diagramas de flujo de datos con los límites de confianza marcados, flujos de autenticación y autorización, modelo de amenazas	Diagramas que ocultan adónde van los datos personales o de pago
Auditores y reguladores	Evidencias de que los controles existen y se cumplen; trazabilidad	Documentos versionados con responsables y fechas de revisión, ADR, correspondencia entre requisitos y controles	Bocetos informales sin fecha, responsable ni versión

Dos hábitos ayudan con cualquier público. **Empieza por la respuesta.** Los lectores ocupados deben conocer la conclusión en la primera frase, después las razones que la respaldan y después el detalle; es la estructura que Barbara Minto enseñó a los consultores como el Principio de la Pirámide, y es la misma idea que la pirámide invertida de Fowler para los ADR.

Usa sus palabras. Para una directora financiera, “consistencia eventual” es ruido; “el restaurante ve un pedido nuevo hasta dos segundos después que el cliente, y no se pierde nada” es un hecho que puede sopesar.

Explicar los compromisos

La mayor parte de la comunicación arquitectónica consiste, en el fondo, en explicar un compromiso. Quienes no son especialistas desconfían de las recomendaciones que parecen no tener ningún inconveniente, y hacen bien. Una estructura clara, usada de forma constante, hace que los compromisos sean fáciles de seguir: las **opciones** (incluida la de no hacer nada), los **criterios** que importan a este público, cómo sale parada cada opción, la **recomendación** y sus **consecuencias**, incluido aquello a lo que renuncias. Es la anatomía de una decisión del Capítulo 1 y el contenido de un ADR, adaptados a quien escucha.

EJEMPLO PRÁCTICO

Ejemplo práctico: explicar un compromiso al equipo directivo

La plataforma de reparto de comida ha sufrido dos caídas un viernes por la noche causadas por un servicio de notificaciones lento. El arquitecto tiene cinco minutos en la reunión de dirección para conseguir la aprobación del ADR-0012 (eventos de pedido asíncronos).

Primero la respuesta	"Recomiendo que cambiemos la forma en que los pedidos circulan entre nuestros sistemas para que un fallo en las notificaciones o en el despacho de repartidores ya no pueda impedir que los clientes hagan pedidos. Cuesta unas seis semanas-equipo y unos pocos cientos de libras al mes."
Por qué ahora	"El viernes 21 de agosto, una caída de las notificaciones bloqueó todos los pedidos durante 23 minutos en el pico del viernes. A unos 40 pedidos por segundo, eso supone aproximadamente 55 000 pedidos que no se pudieron hacer, seguidos de reembolsos y quejas."
Las opciones en sus términos	"Podríamos (A) mantener el diseño actual y añadir límites de seguridad, lo que es barato pero sigue permitiendo que un servicio lento ralentice todos los pedidos; (B) hacer el cambio recomendado; o (C) dejar que otros sistemas lean directamente nuestros datos de pedidos, lo que es rápido pero hace más arriesgado cada cambio futuro."
A qué renunciamos	"Los restaurantes verán los pedidos nuevos uno o dos segundos después de la confirmación del cliente, y asumimos un servicio gestionado más que operar. Creemos que ambas cosas son aceptables; operaciones está de acuerdo."
La decisión que se necesita	"Necesito la aprobación de la dirección para dedicar seis semanas-equipo este trimestre, lo que retrasa dos semanas la funcionalidad de fidelización."

No se nombra ninguna tecnología: "intermediario de mensajes" y "bandeja de salida transaccional" se quedan en el ADR. La cifra de pedidos (40 pedidos por segundo durante 23 minutos) se comprueba con datos reales antes de la reunión.

El costo de la recomendación y las alternativas se exponen con claridad, para que los directivos puedan discrepar por los méritos y no por una cuestión de confianza.

El resumen de arquitectura en una página

Todo sistema se beneficia de un resumen de arquitectura en una página: una sola página que una persona recién llegada, un directivo o un revisor de otro equipo puede leer en cinco minutos. No sustituye a la documentación completa, sino que es su puerta de entrada. Una estructura útil es: qué hace el sistema y para quién (dos frases); los tres objetivos de calidad

principales; un diagrama de contexto del sistema; un diagrama de contenedores o una lista de los bloques de construcción principales; las cinco o así decisiones que lo determinan todo (una línea cada una, con enlace a los ADR); los principales riesgos; y dónde encontrar más. La página arc42-lite de arc42 y plantillas es uno de esos resúmenes.

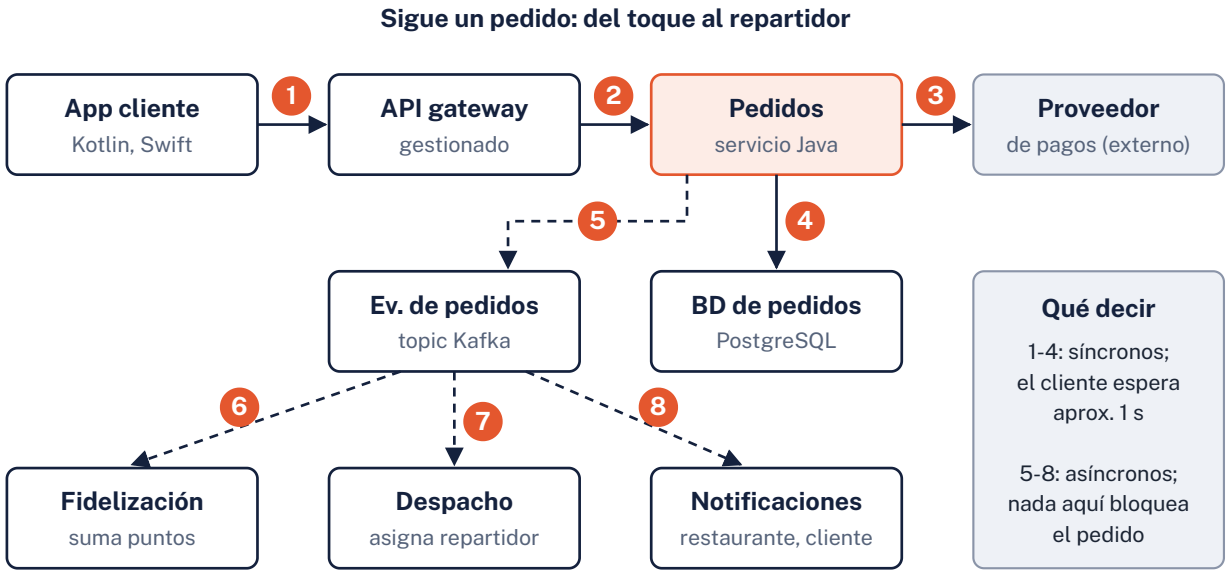
La disciplina de que la arquitectura quepa en una página es valiosa en sí misma. Si las ideas principales no pueden expresarse en una página, probablemente tampoco estén claras para el equipo.

Contar historias: seguir una solicitud

Las personas recuerdan mejor las historias que las estructuras. Una técnica eficaz para explicar cualquier sistema es seguir una solicitud: elige un único evento concreto e importante, como un cliente que hace un pedido, y recorre paso a paso lo que le ocurre a lo largo de la arquitectura. Un diagrama estático de cajas muestra lo que existe; la historia muestra por qué existe y cómo cooperan las partes. Además, revela de forma natural los momentos interesantes: dónde espera el cliente, dónde los datos cruzan un límite de confianza, qué ocurre si el proveedor de pagos va lento.

Elige la historia según el público. Para los desarrolladores, el camino feliz y después un fallo (“ahora el proveedor de pagos agota el tiempo de espera”). Para operaciones, el fallo y la recuperación. Para los directivos, la experiencia del cliente con solo los momentos relevantes para el negocio. Numera los pasos en el diagrama para poder señalarlos, y mantén el mismo diagrama en pantalla mientras hablas en lugar de saltar de una diapositiva a otra.

Seguir un pedido a través de la plataforma



Flechas continuas: llamadas síncronas. Flechas discontinuas: eventos asíncronos. Números: orden de la historia.

Un diagrama de historia numerado: los pasos 1 a 4 ocurren mientras el cliente espera; los pasos 5 a 8 ocurren de forma asíncrona y no pueden bloquear el pedido.

Pizarra y bocetos bien hechos

Gran parte de la comunicación arquitectónica ocurre en directo, ante una pizarra o en un lienzo compartido en línea, y unos pocos hábitos hacen que los bocetos sean mucho más útiles. **Di lo que vas a dibujar antes de dibujarlo** (“este es el nivel de contenedores: cosas que desplegamos por separado”), para que el público conozca el nivel de abstracción. **Empieza por el contexto**: los usuarios y los sistemas externos en los bordes, y después el interior. **Etiqueta cada caja y cada línea** a medida que dibujas, incluidos el sentido y lo que fluye; una flecha sin etiqueta la leerán de tres maneras distintas tres personas. **Dibuja una pequeña leyenda** en una esquina cuando uses más de un tipo de línea o de caja. **Deja espacio**, porque las mejores ideas llegan en mitad de la discusión y necesitan un sitio donde ir.

Los bocetos son herramientas para pensar y están hechos para tirarse, pero las conclusiones no. Fotografía la pizarra y, en el plazo de un día, convierte todo lo que vaya a importar en un diagrama en condiciones, un ADR o un cambio en la documentación. Simon Brown, creador del modelo C4, observó que los diagramas de pizarra de muchos equipos son cajas y líneas ambiguas precisamente porque carecen de un conjunto de abstracciones acordado; usar los niveles de C4 en la pizarra es una de las formas más sencillas de solucionarlo.

Documentos de diseño y RFC

Muchas organizaciones de ingeniería usan un documento de diseño escrito, a menudo llamado RFC (request for comments, solicitud de comentarios), para proponer un cambio significativo antes de construirlo. El nombre evoca la serie RFC de Internet, que empezó con el RFC 1 de Steve Crocker en abril de 1969. El autor expone el problema, el contexto, los objetivos y los no objetivos, el diseño propuesto, las alternativas consideradas y las inquietudes transversales como la seguridad, la privacidad y la observabilidad; el documento se hace circular entre las personas afectadas, que comentan por escrito; y se aprueba o se revisa. La muy leída descripción de Malte Ubl de los documentos de diseño en Google (2020) enumera exactamente esas secciones y sugiere de diez a veinte páginas para los proyectos grandes y de una a tres páginas para los pequeños "minidocumentos de diseño". Gergely Orosz describió cómo Uber usó un proceso de RFC desde sus primeros años para difundir el conocimiento y detectar los problemas a tiempo (2018).

Amazon lleva la escritura más lejos: su carta a los accionistas de 2017 explica que la empresa no usa presentaciones con diapositivas en las reuniones, sino "memorandos de seis páginas con estructura narrativa", que los asistentes leen en silencio al principio. El beneficio común a todas estas prácticas es que escribir obliga a pensar con claridad, los comentarios por escrito permiten contribuir a los compañeros más callados y a los que trabajan en remoto, y el documento se convierte en un registro. Los riesgos también son comunes: documentos largos que nadie termina, procesos de revisión que se convierten en barreras burocráticas y documentos de diseño que nunca se actualizan después de la implementación.

Los documentos de diseño y los ADR se complementan. Un documento de diseño explora un problema y propone un diseño completo, que a menudo abarca varias decisiones; una vez aprobado, las decisiones significativas que contiene se recogen como ADR, que son más breves, permanentes e indexados. Una regla razonable es: el documento de diseño, para pensar y persuadir; el ADR, para recordar.

Presentar en una revisión de arquitectura

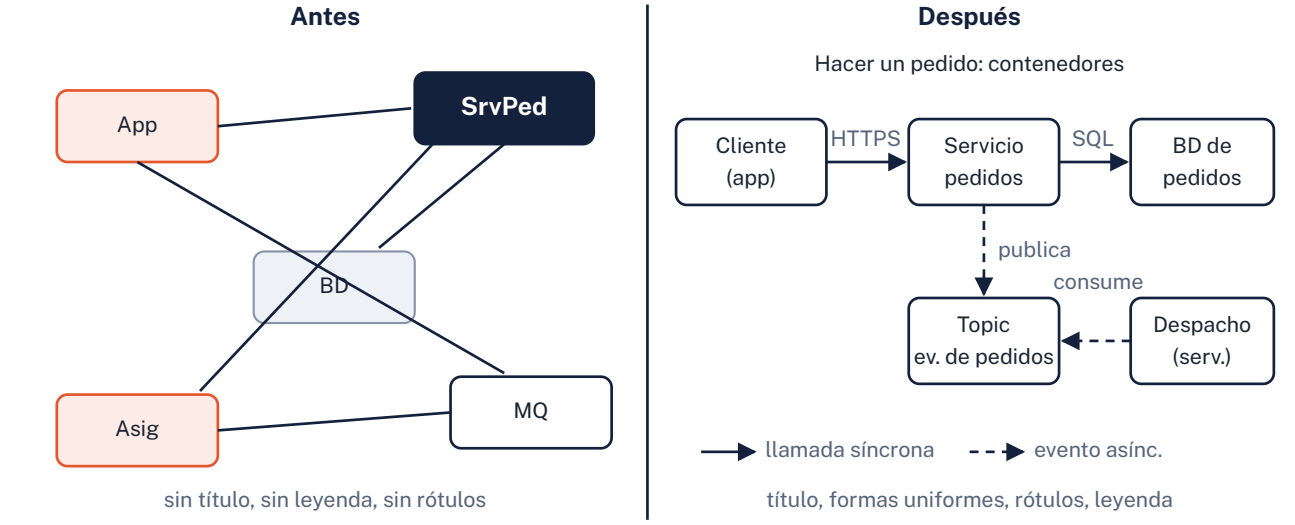
Una revisión de arquitectura es una reunión en la que otras personas con experiencia examinan un diseño para encontrar riesgos antes de que salgan caros. Los métodos estructurados como ATAM y las revisiones más ligeras se tratan en Capítulo 11: revisiones ligeras; aquí lo que importa es cómo presentar bien. Envía el material al menos con dos días de antelación: el resumen de una página, los diagramas clave, los ADR pertinentes y los escenarios de calidad. Di al principio qué quieres de la revisión: una aprobación, una lista de riesgos o una elección entre dos opciones. Presenta los impulsores antes que el diseño, porque un revisor no puede juzgar un diseño sin saber qué cualidades debe lograr. Recorre uno o dos escenarios con la

técnica de seguir una solicitud, incluido un fallo. Muestra los compromisos que asumiste y los riesgos que conoces; un revisor que encuentra un riesgo que ya habías enumerado confía más en ti, no menos. Anota cada problema planteado, con un responsable, y envía la lista después.

Reglas de diseño visual para los diagramas

Los diagramas comunican antes de que nadie lea una palabra, así que su diseño visual importa. Las reglas siguientes proceden de las pautas de notación de C4 de el modelo C4 y del diseño de información en general. El diagrama de antes y después las aplica a los mismos cinco elementos.

El mismo diagrama, antes y después



A la izquierda, el color y las abreviaturas no significan nada y las líneas se cruzan sin etiqueta; a la derecha, un título, formas coherentes, un flujo de izquierda a derecha, flechas etiquetadas y una leyenda.

Reglas para diagramas de arquitectura claros

1. **Pon un título a cada diagrama**
   Indica el tipo y el alcance, como “Contenedores: plataforma de reparto de comida”, para que el lector conozca el nivel de abstracción.

2. **Usa una sola notación de forma coherente**
   La misma forma para el mismo tipo de cosa en todos los diagramas; no mezcles símbolos de C4 y de UML en un mismo diagrama sin decirlo.

3. **Incluye una leyenda**
   Explica cada forma, color, estilo de línea y punta de flecha que tenga significado. Si no puedes explicarlo, quítalo.

4 **Limita los colores y dales significado**
Dos o tres colores, cada uno con un significado (externo, nuevo, en riesgo). No te fíes nunca solo del color: los diagramas se imprimen en blanco y negro y muchos lectores tienen alguna deficiencia en la visión de los colores.

5 **Flujo de izquierda a derecha o de arriba abajo**
Sigue la dirección de lectura de tu público, con las solicitudes fluyendo en un solo sentido, para que la vista siga la historia.

6 **Etiqueta cada elemento y cada flecha**
Nombres sin abreviaturas sin explicar; flechas en un solo sentido, con lo que fluye y, entre contenedores, el protocolo.

7 **Evita que las líneas se crucen siempre que puedas**
Primero reorganiza las cajas; si siguen quedando cruces, divide el diagrama. Un diagrama con muchos cruces suele intentar mostrar demasiado.

8 **Que quepa en una pantalla**
Como mucho, una o dos docenas de elementos aproximadamente. Las imágenes más grandes corresponden a un nivel inferior o a varios diagramas.

SI ESTO ES NUEVO PARA TI

Paso a paso: decidir cuando no todos están de acuerdo

La idea que requiere más atención aquí es el consenso aproximado (rough consensus): se puede tomar una decisión una vez que cada objeción ha sido escuchada con equidad y ha recibido una respuesta razonada, aunque algunos sigan prefiriendo otra opción. Imagina cómo podría desarrollarse una propuesta.

- El equipo de pedidos abre el ADR-0012, publicar los eventos de pedido a través de un intermediario de mensajes, como registro propuesto en un pull request. Se pide a los equipos de restaurantes, de repartidores y de notificaciones, y a SRE, que comenten.
- Llegan dos objeciones. Un ingeniero de notificaciones prefiere mantener las llamadas síncronas con timeouts y reintentos, la opción que necesita menos infraestructura nueva. Una ingeniera del equipo de restaurantes teme que las tabletas muestren tarde los pedidos nuevos.
- Cada objeción se responde por escrito. El registro explica que, con llamadas síncronas, un servicio lento sigue retrasando todos los pedidos, y admite que la tableta de un restaurante puede ir uno o dos segundos por detrás de la confirmación del cliente: un costo que deja registrado.
- El ingeniero de notificaciones sigue sin estar convencido. Su objeción ha sido escuchada con equidad y ha recibido respuesta, pero no se atiende: el diseño no cambia para ajustarse a ella. Para el consenso aproximado, eso basta.
- Antes de decidir, quienes deciden preguntan: '¿Hay alguien que no pueda aceptar esto, y por qué?'. No surge ningún nuevo problema bloqueante. Si una objeción se hubiera quedado sin respuesta, aún no habría consenso aproximado, por muchas personas que estuvieran a favor.
- El registro se acepta con la discrepancia del ingeniero y sus razones por escrito en él. El ingeniero se compromete a que funcione; si más adelante se demuestra que tenía razón, el registro muestra qué hay que revisar.
- **La lección:** busca responder a cada objeción, no que todo el mundo esté de acuerdo. Registra la discrepancia junto con la decisión y establece de antemano quién decide si no se puede alcanzar el consenso.

Gestionar los desacuerdos y construir consenso

Los desacuerdos sobre la arquitectura son normales y sanos: suelen significar que las personas ven riesgos distintos. El objetivo no es ganar, sino llegar a una decisión que las personas que tendrán que vivir con ella entiendan y apoyen. Varias prácticas ayudan.

Separa los intereses de las posiciones. “Tenemos que usar Kafka” es una posición; “me preocupa que perdamos eventos durante los despliegues” es un interés, y los intereses a menudo pueden satisfacerse de varias formas. **Acuerda los criterios antes de comparar opciones**, como en una matriz de decisión ponderada; la mayoría de las discusiones acaloradas son en realidad desacuerdos sobre lo que importa, y se enfrían en cuanto eso se hace explícito. **Resuelve la incertidumbre con evidencias**: un spike de dos días o una prueba de carga zanjan muchos debates que las reuniones no pueden zanjar. **Ponlo por escrito**: un ADR propuesto o un documento de diseño permite que todos comenten a su ritmo y da voz a los compañeros más callados.

Busca el consentimiento en lugar de la unanimidad. El RFC 7282 del IETF (Pete Resnick, 2014) describe el consenso aproximado: se llega a una decisión cuando se han abordado todas las objeciones, lo que significa que se han considerado y respondido, aunque no necesariamente atendido. Preguntar “¿hay alguien que no pueda aceptar esto, y por qué?” saca a la luz los problemas bloqueantes más deprisa que preguntar si todos están de acuerdo. Cuando el consenso no es posible, alguien tiene que decidir, y debe estar claro de antemano quién. La carta a los accionistas de 2016 de Jeff Bezos popularizó el *disagree and commit* (discrepar y comprometerse): las personas expresan abiertamente su desacuerdo y después se comprometen por completo con la decisión una vez tomada. Registra la discrepancia y sus razones en el ADR; si quienes discrepaban acaban teniendo razón, el registro muestra qué hay que revisar y por qué.

Dos modos de fallo merecen una advertencia. El arquitecto que gana todas las discusiones por antigüedad deja de enterarse de los problemas. El equipo que no puede decidir sin unanimidad deja que las decisiones importantes se tomen por omisión, que es lo contrario del último momento responsable.

Lista de comprobación: una presentación de arquitectura

Úsala antes de cualquier presentación de arquitectura, desde una actualización de cinco minutos para la dirección hasta una revisión formal.

- [ ] **Público y resultado identificados**
  Puedes decir en una frase quién estará en la sala y qué debería decidir o hacer después.
- [ ] **Primero la respuesta**
  La recomendación o el mensaje principal está en la primera diapositiva o en el primer párrafo.
- [ ] **Impulsores antes que diseño**
  Los objetivos de negocio, los objetivos de calidad y las restricciones se muestran antes que la solución.
- [ ] **El nivel de abstracción se ajusta al público**
  Nivel de contexto para los directivos, niveles de contenedores y de componentes para los ingenieros; no más de dos niveles en una misma sesión.
- [ ] **Una historia contada de principio a fin**
  Un recorrido siguiendo una solicitud, que incluye un fallo para los públicos técnicos.
- [ ] **Compromisos y riesgos mostrados con honestidad**
  Las opciones, lo que cuesta cada una y aquello a lo que renuncias; los riesgos conocidos, enumerados antes de que nadie más los encuentre.
- [ ] **Cada diagrama tiene título, leyenda y etiquetas**
  Notación coherente, pocos colores y con significado, legible desde el fondo de la sala.
- [ ] **Material enviado con antelación**
  En las revisiones y decisiones, el resumen y los ADR se envían al menos con dos días de antelación, para que la reunión sirva para debatir.
- [ ] **Lo que pides está explícito**
  Aprobación, riesgos, una elección entre opciones o simplemente conocimiento; di cuál, y para cuándo.
- [ ] **Seguimiento registrado**
  Preguntas, objeciones y acciones registradas con responsables; decisiones convertidas en ADR; documentación actualizada.

COMPRUEBA LO QUE HAS ENTENDIDO

▼ ¿Qué pregunta deberías responder antes de preparar cualquier explicación de arquitectura?

Quién es el público y qué debería decidir o hacer de forma distinta después; el contenido, el nivel de abstracción y el vocabulario se derivan de eso.

▼ ¿Por qué contar una historia que sigue una solicitud en lugar de mostrar solo un diagrama de cajas?

Un diagrama de cajas muestra lo que existe; la historia muestra cómo cooperan las partes y por qué, y revela de forma natural las esperas, los límites de confianza y los puntos de fallo que una imagen estática oculta.

▼ ¿En qué se diferencian en su propósito un documento de diseño y un ADR?

Un documento de diseño explora un problema y propone un diseño completo para recibir comentarios antes de construirlo; un ADR es un registro breve y permanente de una decisión. Los documentos de diseño aprobados suelen dar lugar a varios ADR.

▼ ¿Qué significa consenso aproximado en el RFC 7282?

Que cada objeción se ha considerado y respondido, aunque no necesariamente atendido; no es unanimidad ni una votación por mayoría.

▼ Inténtalo: Un arquitecto le dice al equipo directivo: "Necesitamos una bandeja de salida transaccional (transactional outbox) y un broker para desacoplar nuestra ruta síncrona de pedidos". Reescríbelo para ellos en unas tres frases, con la recomendación, el costo y lo que deben decidir.

"Recomiendo cambiar la forma en que los pedidos pasan entre nuestros sistemas, para que un defecto en las notificaciones ya no pueda impedir que los clientes hagan pedidos, como ocurrió dos veces hace poco. Llevaría unas seis semanas de trabajo del equipo y un pequeño costo mensual de operación; la contrapartida es que algunas actualizaciones llegarían unos segundos más tarde. Solicito su aprobación hoy." La respuesta va primero, en términos de negocio, con los costos y el inconveniente expuestos con claridad y sin jerga.

FUENTES

- C4 model: notation: https://c4model.com/diagrams/notation
- Design Docs at Google, Malte Ubl (2020): https://www.industrialempathy.com/posts/design-docs-at-google/

- Scaling Engineering Teams via RFCs: Writing Things Down, Gergely Orosz: https://blog.pragmaticengineer.com/scaling-engineering-teams-via-writing-things-down-rfcs/
- RFC 7282: On Consensus and Humming in the IETF: https://www.rfc-editor.org/rfc/rfc7282.html
- Amazon 2017 letter to shareholders: https://www.aboutamazon.com/news/company-news/2017-letter-to-shareholders

Lecturas recomendadas

- *Documenting Software Architectures: Views and Beyond, 2nd edition*, Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Paulo Merson, Robert Nord and Judith Stafford (2010), Addison-Wesley. El tratamiento más completo de la documentación de arquitectura: vistas, estilos, las siete reglas para una documentación sólida y una plantilla de documentación.
  https://www.sei.cmu.edu/library/documenting-software-architectures-views-and-beyond-second-edition/
- *Software Systems Architecture: Working with Stakeholders Using Viewpoints and Perspectives, 2nd edition*, Nick Rozanski and Eóin Woods (2012), Addison-Wesley. Un catálogo práctico de siete puntos de vista y de perspectivas como la seguridad y el rendimiento, orientado a grandes sistemas de información.
  https://www.viewpoints-and-perspectives.info/
- *Architectural Blueprints: The "4+1" View Model of Software Architecture*, Philippe Kruchten (1995), IEEE Software 12(6). El breve artículo que popularizó la descripción de arquitectura con múltiples vistas y la validación basada en escenarios.
  https://arxiv.org/abs/2006.04975
- *ISO/IEC/IEEE 42010:2022 Software, systems and enterprise: Architecture description*, ISO/IEC JTC 1/SC 7 and IEEE (2022), ISO. La norma internacional que define las partes interesadas, las inquietudes, los puntos de vista, las vistas y los tipos de modelo.
  https://www.iso.org/standard/74393.html
- *The C4 Model: Visualizing Software Architecture*, Simon Brown (2026), O'Reilly Media. El relato del propio creador sobre las abstracciones, los diagramas y la notación de C4; el sitio web cubre lo esencial de forma gratuita.
  https://c4model.com/
- *UML Distilled: A Brief Guide to the Standard Object Modeling Language, 3rd edition*, Martin Fowler (2003), Addison-Wesley. Una guía breve y práctica de las partes de UML que vale la pena conocer, y el origen de la distinción entre boceto, plano y lenguaje de programación.
- *Living Documentation: Continuous Knowledge Sharing by Design*, Cyrille Martraire (2019), Addison-Wesley. Cómo derivar la documentación del propio sistema y limitar el material escrito a mano a lo que solo las personas pueden aportar.

Glosario

arc42

Una plantilla gratuita de doce secciones para documentar y comunicar la arquitectura de software, creada por Gernot Starke y Peter Hruschka.

Bitácora de decisiones (decision log)
La colección ordenada de los ADR de un sistema, normalmente con un índice que enumera el número, el título, el estado y la fecha de cada registro.

Bloque de construcción (building block)
En arc42, cualquier parte de la estructura estática de un sistema, como un subsistema, un servicio, un componente o un módulo, que se muestra en la vista de bloques de construcción.

Componente (C4)
En el modelo C4, un grupo de funcionalidades relacionadas detrás de una interfaz clara, que se ejecuta dentro de un contenedor y se despliega con él.

Componente de vista
En ISO/IEC/IEEE 42010:2022, una parte de una vista, como un diagrama, una tabla o un modelo, regida por un tipo de modelo. La edición de 2011 lo llamaba modelo de arquitectura.

Concepto transversal
Una regla o solución que se aplica a muchas partes de un sistema, como la seguridad, el manejo de errores, el registro o la persistencia.

Consenso aproximado (rough consensus)
Decisión alcanzada cuando todas las objeciones se han considerado y respondido, aunque no necesariamente atendido; descrito en el RFC 7282 del IETF.

Contenedor (C4)
En el modelo C4, una unidad que se puede ejecutar o desplegar por separado y que ejecuta código o almacena datos, como una aplicación web, una app móvil, un servicio o una base de datos. No es un contenedor Docker.

Correspondencia
Una relación registrada entre elementos de una descripción de arquitectura, por ejemplo, entre un contenedor en una vista y el servidor en el que se ejecuta en otra.

Declaración Y (Y-statement)
Registro de decisión en una sola frase, de Olaf Zimmermann: en el contexto de, ante, decidimos, y descartamos, para lograr, aceptando que.

Descripción del diseño de software (SDD)
Documento que describe en detalle un diseño de software, cuyo contenido especifica la norma IEEE Std 1016-2009; habitual en proyectos regidos por contrato.

Diagrama de actividades
Un diagrama UML que muestra un flujo de trabajo como acciones, decisiones, bifurcaciones y uniones paralelas, a menudo dividido en calles (swimlanes) según quién hace qué.

Diagrama de contenedores
Un diagrama C4 que muestra las aplicaciones, los servicios y los almacenes de datos dentro de un sistema, con sus tecnologías y cómo se comunican.

Diagrama de contexto del sistema
Diagrama C4 que muestra un sistema como una sola caja, las personas que lo usan y los demás sistemas con los que interactúa.

Diagrama de despliegue
Un diagrama (en C4 o UML) que muestra cómo se asigna el software a la infraestructura en un entorno, incluidos los nodos, las réplicas y las rutas de red.

Diagrama de máquina de estados
Diagrama UML que muestra los estados en que puede estar un objeto y las transiciones entre ellos, cada una desencadenada por un evento.

Diagrama de secuencia
Diagrama UML que muestra los mensajes intercambiados entre participantes a lo largo del tiempo para un escenario, con el tiempo avanzando hacia abajo en la página.

Diagrama dinámico
Un diagrama C4 complementario que muestra cómo colaboran los elementos en tiempo de ejecución en un escenario, normalmente como interacciones numeradas.

Diagramas como código
Escribir diagramas como texto en un pequeño lenguaje, como PlantUML, Mermaid, D2 o Structurizr DSL, que una herramienta distribuye y dibuja.

Documentación como código (docs as code)
Producir documentación con las mismas herramientas y el mismo flujo de trabajo que el software: archivos de texto plano en control de versiones, revisión de código, comprobaciones automatizadas y publicación.

Documentación justa y necesaria
Documentación dimensionada según lo que sus lectores realmente necesitan, de modo que escribir y mantener más costaría más de lo que aporta.

Documentación viva
Documentación que evoluciona con el sistema que describe, a menudo porque se genera a partir del propio sistema o se verifica contra él.

Documento de diseño (RFC)
Una propuesta escrita para un cambio significativo, que abarca el problema, los objetivos, el diseño y las alternativas, y que se distribuye para recibir comentarios antes de construir el cambio.

Entidad de interés
En ISO/IEC/IEEE 42010:2022, aquello cuya arquitectura se describe: un sistema de software, un sistema de sistemas, una línea de productos o una empresa.

Guarda (guard)
Una condición entre corchetes en una transición o un flujo que debe ser verdadera para que se tome, como [code matches].

Herramienta de dibujo
Una herramienta de diagramas que solo conoce formas, líneas y texto, de modo que el mismo elemento dibujado en dos diagramas no tiene ningún vínculo entre ellos.

Herramienta de modelado
Herramienta que almacena elementos y relaciones en un modelo y genera diagramas como vistas de él, manteniendo la coherencia de todas las vistas.

Impulsor de decisión
Una fuerza o un criterio, como un objetivo de calidad o una restricción, que una decisión arquitectónica debe satisfacer y con respecto al cual se comparan las opciones.

Interfaz proporcionada
En un diagrama de componentes UML, una interfaz que un componente ofrece a otros, representada como un pequeño círculo (una bola) sobre un palo.

Interfaz requerida
En un diagrama de componentes UML, una interfaz que un componente necesita que otro componente proporcione, representada como un semicírculo (un enchufe).

Lenguaje Unificado de Modelado (UML)
Lenguaje gráfico de modelado de propósito general estandarizado por el Object Management Group. La versión actual es la 2.5.1 (2017).

Línea de vida
En un diagrama de secuencia, la línea vertical discontinua bajo un participante, a lo largo de la cual se muestran sus mensajes y activaciones en orden temporal.

MADR
Markdown Architectural Decision Records: una plantilla de ADR muy utilizada con impulsores de la decisión, opciones consideradas, el resultado, las consecuencias y la confirmación.

Marco de descripción de arquitectura
En ISO/IEC/IEEE 42010:2022, un conjunto establecido de convenciones para elaborar descripciones de arquitectura, como puntos de vista, plantillas y reglas.

Modelo C4
El enfoque de Simon Brown para los diagramas de arquitectura: una jerarquía de diagramas de contexto del sistema, de contenedores, de componentes y de código, más algunos diagramas complementarios.

Modelo de dominio
Un pequeño diagrama o descripción de los conceptos de negocio clave de un sistema, sus atributos y cómo se relacionan, compartido por los desarrolladores y los expertos del dominio.

Modelo de vistas 4+1
Enfoque de Philippe Kruchten de 1995 que describe un sistema mediante las vistas lógica, de procesos, de desarrollo y física, unidas por un pequeño conjunto de escenarios.

Multiplicidad
En un diagrama de clases UML, cuántos objetos pueden participar en cada extremo de una asociación, como 1, 0..1, 0..* o 1..*.

Perspectiva
En el enfoque de Rozanski y Woods, una inquietud de calidad, como la seguridad o el rendimiento, que se aplica a todas las vistas en lugar de representarse como una vista propia.

Proceso de consejo (advice process)
Una forma descentralizada de tomar decisiones en la que cualquiera puede decidir, siempre que antes pida consejo a los afectados y a los expertos pertinentes.

Resumen de arquitectura en una página
Una sola página con el propósito del sistema, sus objetivos de calidad, su contexto, sus bloques de construcción principales, sus decisiones clave y sus riesgos, como puerta de entrada a una documentación más completa.

Seguir una solicitud
Explicar un sistema rastreando un único evento concreto, como un pedido, paso a paso a través de cada parte por la que pasa.

Sistema de software (C4)
En el modelo C4, el nivel de abstracción más alto: algo que aporta valor a sus usuarios y que normalmente pertenece a un equipo u organización.

Structurizr
Las herramientas de Simon Brown para el modelo C4, en las que un modelo basado en texto genera muchos diagramas coherentes.

Tipo de modelo (model kind)
En ISO/IEC/IEEE 42010, las convenciones para un tipo de modelo, como un diagrama de secuencia UML o un diagrama de contenedores C4.

Vista de asignación

Una vista que relaciona el software con elementos externos a él: la infraestructura (despliegue), los archivos y las compilaciones (implementación) o los equipos (asignación de trabajo).

Vista de componentes y conectores

Una vista de lo que existe en tiempo de ejecución, como servicios, clientes, bases de datos y colas, y de los conectores (llamadas, mensajes, datos compartidos) entre ellos.

Vista de módulos

Vista de las unidades de implementación (paquetes, bibliotecas, capas) y sus dependencias, tal como existen en tiempo de desarrollo.