

GDCD EN CÓDIGO

LIBRO 1

DISEÑO DE CONTRATOS GUIADO POR LA GOBERNANZA

Gobernar Rust por contrato

Ser dueños de los bits: del compilador al clúster.

La mayoría de los equipos elige un lenguaje y después descubre lo que cuesta. Esta guía empieza al revés: desde el resultado, pasando por las garantías, hasta los contratos que las sostienen. Muestra lo que el compilador de Rust demuestra de verdad, lo que las herramientas deben hacer cumplir a su alrededor y lo que sigue en manos de las operaciones y las personas, desde el primer registro de decisión hasta un servicio en ejecución sobre Kubernetes. Es honesta sobre dónde Rust no encaja.

Juan Cardena

Arquitecto de IA empresarial y sistemas agénticos · blog.jcardena.com

CONTENIDO

00 Por qué este libro 3

PARTE I · LA DECISIÓN

01 Ser dueños de los bits: lo que Rust cambia y lo que no 6

02 El registro de decisión y su renovación 13

03 Cuándo no usar Rust 20

PARTE II · LA PUERTA DEL CÓDIGO

04 El contrato del compilador 26

05 El contrato de excepciones: unsafe, FFI y código en tiempo de compilación 33

06 El contrato de verificación 41

PARTE III · LA PUERTA DE LA CADENA DE SUMINISTRO

07 La fijación: dependencias y cadena de herramientas 48

08 El control de admisión 55

09 La promesa de la interfaz 64

PARTE IV · LA PUERTA DEL ENTORNO DE EJECUCIÓN

10 Publicar el artefacto 72

11 Operar bajo contrato 79

PARTE V · EJECUTAR LA DECISIÓN

12 El modelo operativo 88

APÉNDICES

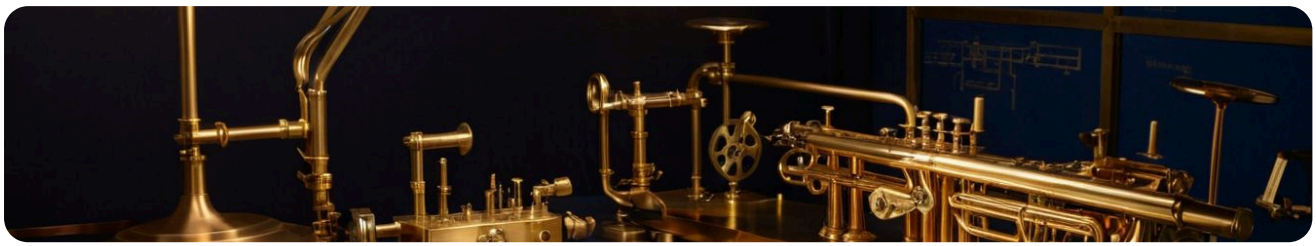
A Plantillas y el servicio de referencia 96

B Los tres niveles en otros lenguajes 113

C Glosario y registro de fuentes 117

PREFACIO

Por qué este libro



La mayoría de los debates sobre lenguajes son discusiones sobre gustos que terminan en una migración que nadie puede deshacer. Este libro trata de la parte que sobrevive al debate: la decisión en sí, quién es su responsable, qué debe aportar y cómo el entorno tecnológico demuestra, año tras año, que sigue en pie. Rust es el caso de estudio. La gobernanza es el tema.

He pasado más de 25 años construyendo plataformas web, software, sistemas de datos y, últimamente, sistemas de IA y agénticos. En ese tiempo he visto a equipos adoptar lenguajes por buenas y por malas razones. Las buenas razones rara vez tienen que ver con la sintaxis. Tienen que ver con el control: dónde vive la memoria, qué contiene un binario, qué introduce cada dependencia, cómo se comporta un servicio en el percentil 99 en un mal día. A eso lo llamo **ser dueños de los bits**. Rust es un lenguaje que permite a un equipo adueñarse de ellos sin un recolector de basura, y pone de por medio un compilador que rechaza clases enteras de errores antes de que el código llegue a ejecutarse.

Ese rechazo es valioso, y también es fácil exagerar su alcance. Un compilador demuestra algunas cosas. No demuestra que sus dependencias sean honestas, que su servicio termine de forma ordenada cuando se retira un nodo (que deje de aceptar solicitudes y complete las que están en curso), ni que la lógica de negocio sea correcta. Por eso el libro se construye sobre una imagen honesta de dónde proviene cada garantía.

Tres niveles, una imagen

La serie a la que pertenece este libro, Diseño de Contratos Guiado por la Gobernanza (GDCD), parte de un resultado de negocio, lo convierte en garantías, ubica esas garantías en capas y las hace cumplir como contratos en cada puerta. En los libros sobre IA la frase es "el modelo propone, el contrato decide". En la puerta del código la misma idea se lee **"el desarrollador propone, el compilador decide"**, pero solo para lo que el compilador realmente puede verificar. Todo lo demás necesita otro mecanismo que lo haga cumplir. La serie también tiene un lema para la capa de software, **"Cada línea se sostiene"**. En este

libro su alcance está acotado: las garantías de una línea se sostienen solo en la medida en que uno de los tres niveles las hace cumplir, y es un objetivo que el entorno verifica con evidencia, no una promesa de que el código no puede fallar.

FIGURA 0.1

Los tres niveles, proyectados sobre la pila GDCD



Cada capítulo pregunta qué nivel sostiene una garantía. Cuanto más se acerca una garantía al Nivel 1, más barato y más fiable es hacerla cumplir. Llevar una garantía a un nivel superior, de las personas a las herramientas o de las herramientas al compilador, es la mayor parte de lo que hace una buena gobernanza de ingeniería.

Los niveles son la columna vertebral del libro. Cada capítulo nombra un contrato, dice qué nivel lo hace cumplir, quién es su responsable, qué pasa cuando falla y qué evidencia muestra que se ejecutó. Los niveles también se aplican a cualquier lenguaje. El Apéndice B los proyecta sobre TypeScript, Go, .NET y Java, de modo que un lector que nunca llegue a poner en producción una línea de Rust puede usar igualmente el método.

A quién va dirigido

El lector que tengo en mente es un CTO, un CIO, un arquitecto empresarial o de plataforma, un líder de ingeniería o un CISO. Usted es responsable de una decisión de lenguaje, o se le pedirá que apruebe una. Puede que usted no escriba Rust. Necesita saber qué aporta, qué cuesta, cómo gobernarlo una vez que está en el entorno y cuándo decir que no.

Cada capítulo comienza exponiendo su idea central con palabras sencillas. Un CTO puede leer el párrafo inicial y la primera sección, y detenerse ahí. A continuación viene el detalle de ingeniería, para quienes construirán los controles.

Lo que este libro no es. No es un tutorial de Rust: el libro oficial y sus complementos lo hacen bien, y remito a ellos. No es un folleto de benchmarks: aquí no hay comparativas de velocidad entre lenguajes, y las únicas cifras de consumo de recursos provienen de un pequeño servicio de referencia del Apéndice A, construido y medido con un método que usted puede repetir. No es un argumento para contratar ni una afirmación de que Rust deba reemplazarlo todo. El capítulo tres trata de cuándo no usarlo.

Cómo está organizado el libro

La Parte I es la decisión: qué significa ser dueños de los bits, cómo redactar el registro de decisión para que pueda renovarse o revertirse, y dónde Rust no encaja. La Parte II es la puerta del código: el contrato del compilador, el contrato de excepción para el código unsafe, la FFI y el código de tiempo de compilación, y el contrato de verificación. La Parte III es la puerta de la cadena de suministro: fijar dependencias y cadenas de herramientas (toolchains), el control de admisión de lo que puede entrar en una compilación, y la promesa que hace una interfaz pública. La Parte IV es la puerta del entorno de ejecución: entregar el artefacto y ejecutarlo bajo contrato. La Parte V lo integra todo como un modelo operativo.

Ruta de lectura	Capítulos	Qué se lleva
Ruta del CTO	Prefacio, 1, 2, 3, 8, 11, 12	Un registro de decisión que puede firmar y renovar, el control de admisión, la evidencia en tiempo de ejecución y el modelo operativo
Ruta técnica del arquitecto	Partes II a IV (capítulos 4 a 11)	Los contratos, sus puntos de aplicación y las plantillas del Apéndice A

Cada uno de los capítulos cuatro a once termina con una ficha de contrato: el contrato, la amenaza que aborda, su responsable, dónde se hace cumplir, cómo falla, quién puede concederle una excepción y hasta cuándo, y la evidencia de que se ejecutó y de que sirvió. Todos los capítulos, del uno al doce, terminan con una breve lista de verificación.

Las herramientas aparecen donde son la forma estándar de hacer algo, y los archivos de configuración completos están en el Apéndice A con la versión de la cadena de herramientas con la que se verificaron. Las herramientas cambian. El contrato es lo que debería seguir en pie cuando cambien.

> *Las herramientas son moda. La confianza es el trabajo.*

PARTE I · LA DECISIÓN

CAPÍTULO UNO

Ser dueños de los bits: lo que Rust cambia y lo que no



Un entorno de ejecución (runtime) hace promesas en su nombre. Un recolector de basura decide cuándo se recupera la memoria. Una máquina virtual decide cuánto de sí misma se entrega junto con su servicio. La mayor parte del tiempo esas promesas son buenas, y aceptarlas es la decisión correcta. Rust es una forma de recuperar algunas de esas promesas y hacerlas usted mismo, con un compilador que verifica una parte de su trabajo. Es una ganancia real, con un precio real, y ambos deben formar parte de la decisión.

Esta es la versión corta para un CTO. Rust da a un equipo control sobre la memoria, la disposición de los datos, la asignación y el tamaño del binario sin un recolector de basura, y su compilador demuestra la seguridad de memoria y la ausencia de carreras de datos para el código escrito en Rust seguro. Esas demostraciones se cumplen dentro de un límite que usted debe gobernar. La evidencia pública más sólida a favor de los lenguajes con seguridad de memoria proviene de bases de código en C y C++, así que es un argumento para dejar C y C++. El argumento frente a entornos de ejecución administrados como Java, .NET o Go es distinto y más acotado: trata de previsibilidad y de consumo de recursos, y tiene costos en tiempo de compilación, habilidades y madurez del ecosistema.

Qué significa ser dueños de los bits

Uso "ser dueños de los bits" para referirme a una **responsabilidad gobernada** sobre aquello de lo que está hecho un servicio y sobre cómo se comporta. No es una sensación de control, ni la afirmación de que el equipo lo escribe todo por sí mismo. Un equipo es dueño de los bits cuando puede responder, con evidencia, cinco preguntas sobre cualquier servicio que opera:

- **Memoria:** ¿dónde se asigna, quién la libera y cuál es el peor caso?
- **Binario:** ¿qué contiene el artefacto y de dónde vino cada parte?
- **Dependencias:** ¿qué introdujo cada una, quién la aprobó y qué puede hacer en tiempo de compilación?
- **Comportamiento bajo carga:** ¿cómo es el percentil 99 y qué lo acota?
- **Responsabilidad:** ¿quién es responsable de cada una de estas respuestas y cuándo se verificaron por última vez?

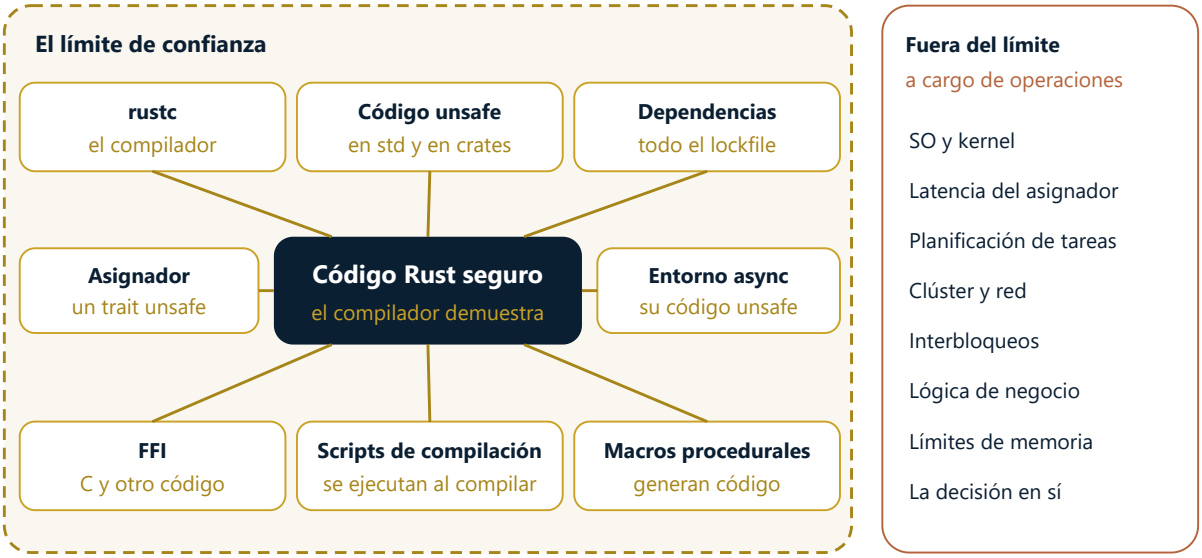
Cualquier lenguaje puede responder esas preguntas con suficiente esfuerzo. Rust abarata algunas de las respuestas, porque el compilador las verifica, y vuelve otras más directas, porque no hay un recolector de basura ni una máquina virtual entre el código y la máquina. La quinta pregunta nunca la responde un lenguaje. La responde un modelo operativo, y por eso este es un libro sobre gobernanza.

El límite de confianza

Las garantías del compilador son precisas, y son condicionales. El Rust seguro no puede causar comportamiento indefinido, como un uso después de liberar o una carrera de datos, pero solo si todo aquello sobre lo que se apoya también es correcto. Esa base es lo que llamo el **límite de confianza**.

FIGURA 1.1

Las garantías del Rust seguro dependen de todo lo que hay dentro del límite



El compilador verifica su código seguro y confía en el resto. Cada caja dentro de la línea discontinua puede romper la garantía si es incorrecta. Cada elemento de la derecha está fuera de lo que el compilador puede ver. El asignador y el entorno de ejecución asíncrono aparecen en ambos lados: se confía en su solidez, y su comportamiento bajo carga es asunto de operaciones.

Hay ocho cosas dentro del límite. El propio **compilador**, que tiene errores como cualquier software. Cada **abstracción unsafe**, en la biblioteca estándar y en los crates, cuyos autores prometen que la interfaz segura que exponen no puede usarse mal. Cada **dependencia** del lockfile, porque el código unsafe de esas dependencias ahora también es el suyo. Cada llamada **FFI** a C o a otro lenguaje, donde Rust da por bueno lo que el código externo declara sobre su comportamiento. Cada **script de compilación** (build.rs), que ejecuta código arbitrario en la máquina de compilación. Cada **macro procedural**, que se ejecuta en tiempo de compilación y escribe código dentro de su crate. El **asignador global**, porque `GlobalAlloc` es un trait unsafe: el código seguro confía en que devuelve memoria válida. El **entorno de ejecución asíncrono** (async runtime), cuyo ejecutor, temporizadores y E/S están construidos sobre código unsafe del que dependen los futures seguros. En estos dos últimos casos, la línea separa la solidez del comportamiento. Que el asignador y el entorno de ejecución sean correctos queda dentro del límite. La velocidad del asignador, si devuelve memoria al sistema operativo y cómo planifica las tareas el entorno de ejecución quedan fuera, y corresponden a operaciones. La Parte II y la Parte III de este libro son, en gran medida, los contratos que gobiernan estas ocho cosas.

La evidencia, con su alcance

El argumento a favor de los lenguajes con seguridad de memoria se apoya en datos públicos, y esos datos tienen un alcance específico. El Security Response Center de Microsoft informó que cerca del 70% de las vulnerabilidades a las que Microsoft asigna un CVE cada año son problemas de seguridad de memoria (MSRC, 2019). El proyecto Chromium informa que alrededor del 70% de sus errores de seguridad de alta gravedad son problemas de seguridad de memoria, que describe como errores con punteros de C y C++ (Chromium Project, 2020). El equipo de Android de Google informó que, a medida que el desarrollo nuevo pasó a lenguajes con seguridad de memoria, la proporción de vulnerabilidades de Android relacionadas con la seguridad de memoria cayó del 76% en 2019 al 24% en 2024 (Google Security Blog, 2024). CISA, la NSA y agencias asociadas han publicado guías que instan a los fabricantes a planificar el paso a lenguajes con seguridad de memoria (CISA et al., 2023), y la Office of the National Cyber Director de EE. UU. defendió lo mismo (ONCD, 2024).

Todo esto trata de bases de código escritas en C y C++. Es evidencia fuerte a favor de dejar C y C++, o para escribir código nuevo en un lenguaje con seguridad de memoria junto a ellos, que es lo que hizo Android. **No respalda reescribir en Rust servicios de Java, .NET, Go o TypeScript.** No es evidencia a favor de abandonar esos lenguajes. Esos lenguajes ya tienen seguridad de memoria en el sentido que miden estos informes. Usar la cifra del 70% para defender la reescritura de un servicio Java en Rust es leer mal la fuente, y un revisor cuidadoso lo notará.

Lo que Rust aporta frente a los entornos de ejecución administrados

Frente a un entorno de ejecución administrado, el argumento trata de previsibilidad y control, no de seguridad de memoria. Hay cuatro ganancias reales:

- **Sin pausas del recolector de basura.** Un valor se libera cuando su propietario sale de ámbito, así que ningún recolector decide cuándo pausar el servicio. Eso elimina una fuente de variación de latencia, no todas: los destructores consumen tiempo, la memoria liberada puede no volver al sistema operativo y los ciclos de valores con recuento de referencias (`Rc`) provocan fugas de memoria. El resto hay que medirlo. Los recolectores modernos como ZGC han hecho las pausas muy cortas, así que esta es una diferencia en el tipo de control, no una afirmación de que los servicios administrados sean lentos.
- **Un único binario pequeño.** Un servicio Rust se compila en un solo ejecutable sin recolector de basura ni máquina virtual que instalar, lo que ayuda a mantener pequeñas las imágenes de contenedor y rápido el arranque. El binario aún puede enlazar la biblioteca de ejecución de C o bibliotecas del sistema, así que solo debe llamarse autocontenido después de inspeccionar sus dependencias dinámicas. .NET NativeAOT, GraalVM native image y Go también producen un único binario nativo, así que esta es una fortaleza que Rust comparte, no una exclusiva suya.
- **Ausencia de carreras de datos en código seguro.** Los traits Send y Sync permiten al compilador rechazar código que compartiría estado mutable entre hilos de forma insegura. Los sistemas de tipos de Go y Java no impiden las carreras de datos: Go incluye un detector dinámico de carreras, y Java depende de la disciplina de sincronización, las pruebas y las herramientas de análisis.
- **Control de la disposición y la asignación.** El equipo decide cómo se disponen los datos en memoria y cuándo se asignan. Eso importa para analizadores sintácticos, proxies, códecs y todo lo que está en una ruta crítica.

En este libro no pongo cifras a estas ganancias, porque los benchmarks entre lenguajes dependen mucho de cómo se escribió y ajustó cada versión, y rara vez coinciden con una carga de trabajo concreta. El Apéndice A mide el consumo de recursos de un servicio de referencia, exponiendo primero el método; el capítulo once presenta el método para medir la latencia.

Lo que cuesta

Los costos son igual de reales, y un registro de decisión que los omita acabará revertido en la primera revisión presupuestaria.

- **Tiempos de compilación.** Las compilaciones de Rust son más lentas que las de Go, y los workspaces grandes necesitan caché, presupuestos de dependencias cuidadosos y más capacidad de CI.
- **Curva de aprendizaje.** La propiedad (ownership) y los tiempos de vida (lifetimes) requieren tiempo para dominarse. La productividad baja antes de recuperarse; mida esa caída en un piloto en lugar de suponer una cifra.
- **Disponibilidad de talento.** Los ingenieros de Rust con experiencia suelen ser más difíciles de encontrar que los de Java, C# o Go. Revise su propio mercado de contratación y planifique formarlos.
- **Un ecosistema más joven en algunas áreas.** Las redes, la serialización y las herramientas de línea de comandos están maduras. Algunas integraciones empresariales, formatos de informes y SDK de proveedores son más limitados o no existen. El capítulo tres explica qué hacer al respecto.

> **Lo que Rust no garantiza.** El Rust seguro no impide los **interbloqueos**, ni las **condiciones de carrera en general**: impide las carreras de datos, no las carreras lógicas en las que dos operaciones correctas ocurren en el orden equivocado. No impide las **fugas de memoria y de recursos**, que la Rust Reference enumera como comportamiento que no se considera unsafe. No impide los **pánicos**, y un pánico en un servicio sigue siendo una caída. El código asíncrono puede **cancelarse** en cualquier punto de await, dejando trabajo a medias. Por defecto, el **desbordamiento de enteros** provoca pánico en compilaciones de depuración, y en compilaciones de release se desborda sin aviso y continúa con aritmética modular (wrapping). Quedarse **sin memoria** aborta el proceso por defecto. Y nada en el lenguaje acota la **latencia** por sí solo: el asignador, el planificador asíncrono, el kernel y el clúster influyen en ella. Cada elemento de esta lista es un contrato para las herramientas o para operaciones.

El punto del desbordamiento sorprende a la gente, así que merece un ejemplo. Un equipo que quiere que el desbordamiento falle de forma visible en producción puede indicarlo en el perfil de release, o hacer explícita la intención en el código:

```
# Cargo.toml
[profile.release]
overflow-checks = true   # panic ante desbordamiento en release, como en debug

// src/stock.rs
pub fn add_units(stock: u32, incoming: u32) -> Option<u32> {
    stock.checked_add(incoming)   // None ante desbordamiento, nunca aritmética modular
silenciosa
}
```

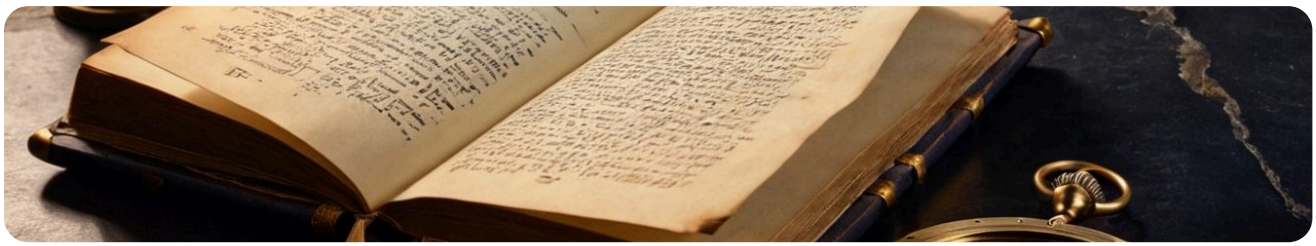
Ese es el patrón de todo el libro. El lenguaje da un valor por defecto. El equipo decide si ese valor coincide con la garantía que prometió, deja la decisión por escrito y hace que una máquina la imponga donde pueda. Donde el desarrollador propone y el compilador decide, la línea se sostiene. Donde el compilador no puede ver, alguien debe hacerse responsable.

LISTA DE VERIFICACIÓN DEL CONTRATO

- La decisión nombra el resultado de negocio al que sirve Rust, en términos de memoria, binario, dependencias, comportamiento bajo carga y responsabilidad.
- La evidencia sobre seguridad de memoria se cita con su alcance: bases de código en C y C++, no entornos de ejecución administrados.
- El límite de confianza está por escrito: versión del compilador, código unsafe, dependencias, FFI, scripts de compilación, macros procedurales, el asignador y el entorno de ejecución asíncrono, cada uno con un responsable.
- Cada elemento de "Lo que Rust no garantiza" está asignado a un contrato de herramientas o de operaciones.
- Los costos (tiempo de compilación, curva de aprendizaje, contratación, carencias del ecosistema) aparecen en el registro de decisión junto a las ganancias.

CAPÍTULO DOS

El registro de decisión y su renovación

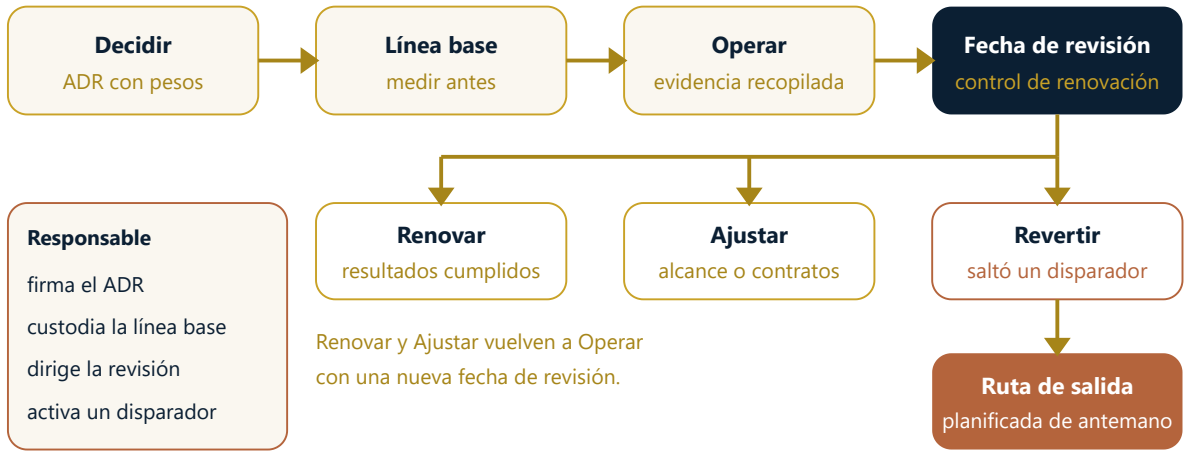


Las decisiones de lenguaje suelen tomarse una sola vez, en una reunión, y luego las defienden durante una década personas que no estaban en la sala. La solución no es una mejor reunión. Es un registro de decisión que establece qué debe aportar la elección, cómo podrá comprobarlo cualquiera, cuándo se volverá a verificar y qué haría que la organización cambiara de rumbo. Una decisión con fecha de revisión y ruta de salida es un contrato. Una decisión sin ellas es un hábito.

La versión corta: escriba un registro de decisión de arquitectura (ADR) antes de que el primer servicio llegue a producción. Puntúe las opciones según criterios cuyos pesos se fijan antes de que nadie puntúe. Exponga por escrito, con equidad, los argumentos de cada alternativa. Calcule el costo total, no solo el costo de ejecución. Luego adjunte un contrato de renovación: resultados esperados, una línea base, una fecha de revisión, un responsable, disparadores de reversión y una ruta de salida. El resto de este capítulo es la plantilla.

FIGURA 2.1

Una decisión que puede renovarse, ajustarse o revertirse



La fecha de revisión es parte de la decisión, no un añadido posterior. La renovación es una elección que el responsable toma a partir de la evidencia. La reversión solo es barata si la ruta de salida se diseñó antes de necesitarla.

La plantilla

Un ADR para la elección de un lenguaje es breve. Su valor está en lo que obliga a las personas a dejar por escrito. La plantilla completa, lista para copiar, está en el Apéndice A.

Campo	Qué escribir
ID, título, estado	"ADR-0000: Rust para servicios con requisitos estrictos de latencia". El estado es propuesto, aceptado, renovado, ajustado o revertido.
Responsable	Un único rol designado, no un comité. La persona que dirigirá la revisión.
Resultado de negocio	El resultado al que sirve la elección, en términos de negocio. Por ejemplo: una ruta de pago cuya latencia en el percentil 99 se mantiene dentro del límite previsto durante los picos de demanda.
Garantías requeridas	Qué debe ser cierto y qué nivel hace cumplir cada cosa: compilador, herramientas u operaciones (capítulo uno).
Alcance	Qué servicios, qué equipos, qué tipos de código nuevo. Qué queda explícitamente fuera del alcance.
Opciones consideradas	Cada alternativa seria, con una defensa equilibrada a su favor, escrita por alguien que la elegiría.
Criterios y pesos	Fijados antes de puntuar. Las puntuaciones y el razonamiento detrás de cada una.
Decisión y costos aceptados	Qué se eligió y qué costo asume conscientemente la organización por ello.
Contrato de renovación	Resultados esperados, línea base, fecha de revisión, disparadores de reversión, ruta de salida (más abajo).
Ubicación de la evidencia	Dónde se guardan las mediciones y las actas de revisión.

Criterios con pesos fijados de antemano

El orden importa más que las cifras. Acuerde los criterios, acuerde los pesos y luego puntúe. Si los pesos se modifican después de conocer las puntuaciones, el registro es una racionalización a posteriori. Aquí no doy pesos, porque dependen del resultado que la organización quiere obtener. Los marcadores de posición muestran la estructura.

Criterio	Peso	Qué cuenta como evidencia
Latencia de cola y uso de recursos previsibles	w1	Medidos en una carga de referencia con un método declarado
Seguridad en los puntos donde se analizan entradas no confiables	w2	Modelo de amenazas, historial de esa clase de componente
Habilidades del equipo y contratación	w3	Inventario actual de habilidades, mercado local de contratación, plan de formación
Encaje del ecosistema con el dominio	w4	Bibliotecas y SDK necesarios, identificados por su nombre, con su madurez y sus responsables
Costo de compilación y de CI	w5	Tiempos de compilación y minutos de CI de un piloto, no estimaciones
Madurez operativa	w6	Herramientas de perfilado, trazado y depuración que los operadores ya conocen
Interoperabilidad con el entorno tecnológico existente	w7	Interfaces con servicios existentes, formatos de datos, identidad
Mantenibilidad a largo plazo	w8	Frecuencia de versiones publicadas, esfuerzo de actualización, quién mantiene la cadena de herramientas (toolchain)

Los pesos w1 a w8 suman 100 y el responsable los firma antes de que comience la puntuación.

Un párrafo equilibrado para cada alternativa

Un registro de decisión inspira confianza cuando cada alternativa la describe alguien que la elegiría. Cada uno de los lenguajes siguientes es la mejor opción en algún terreno, y el registro debe decir dónde.

Go. Compilaciones rápidas, un modelo de concurrencia simple con goroutines y canales, una biblioteca estándar de redes sólida y un único binario estático. Su recolector de basura está diseñado para pausas cortas. Para servicios de red en los que el equipo valora la velocidad de entrega y la legibilidad por encima del control de la memoria, Go suele ser la mejor opción.

C# y .NET. Alta productividad, herramientas maduras, un gran ecosistema de bibliotecas y una integración empresarial profunda. NativeAOT compila las aplicaciones por adelantado en un ejecutable nativo autocontenido con arranque rápido y menor consumo de recursos, con algunas limitaciones en la reflexión y en la generación de código en tiempo de ejecución. Un equipo que ya domina .NET paga un precio real si lo abandona.

Java. Décadas de madurez operativa, perfilado y monitoreo que los operadores conocen bien, y un ecosistema de bibliotecas grande y maduro. ZGC es un recolector de pausas cortas construido para heaps grandes, y GraalVM native image construye ejecutables nativos independientes para un arranque más rápido, bajo un supuesto de mundo cerrado (closed-world) que exige cuidado con la reflexión. Java es una opción predeterminada razonable para muchos servicios internos de gestión (back office) y de datos.

C++. Una amplia base de código existente, bibliotecas y experiencia en muchos dominios, y control total sobre la memoria y la disposición de los datos. Donde existe una base de código C++ grande y bien probada, la pregunta realista suele ser cómo escribir componentes nuevos en un lenguaje con seguridad de memoria junto a ella, no si reescribirla.

Zig. Un lenguaje de sistemas joven con asignación explícita e interoperabilidad simple con C. No ha llegado a la versión 1.0, y el lenguaje y su biblioteca estándar aún cambian entre versiones publicadas. Conviene seguirlo de cerca; todavía no es momento de confiarle el entorno tecnológico de una empresa.

TypeScript y Node.js. El lenguaje natural del navegador y de los servicios que están más cerca de él. Los tipos compartidos entre front end y back end, un gran ecosistema de paquetes y una amplia disponibilidad de talento lo convierten en una opción sólida para interfaces de usuario y servicios perimetrales limitados por E/S. Encaja mal en rutas críticas con uso intensivo de CPU, que es a menudo donde Rust lo complementa (capítulo tres).

Costo total, no solo costo de ejecución

El costo de ejecución es la cifra más fácil de encontrar y la que menos probabilidades tiene de decidir el resultado. El registro debe llevar una línea para cada uno de estos puntos, a partir de un piloto cuando sea posible:

- **Contratación e incorporación:** tiempo para cubrir un puesto, y tiempo hasta que un nuevo ingeniero lleva cambios a producción con seguridad.
- **Formación:** cursos, trabajo en pareja y la caída de productividad durante el período de aprendizaje, medida en el piloto.
- **Minutos de CI y hardware de compilación:** las compilaciones de Rust necesitan más cómputo y un buen uso de caché.
- **Mantenimiento de la cadena de herramientas:** versiones estables de Rust cada seis semanas, y alguien debe ser responsable de las actualizaciones (capítulo siete).
- **Revisión de dependencias:** esfuerzo de revisión para crates nuevos y su código unsafe (capítulo ocho).
- **Costo de ejecución:** cómputo y memoria, medidos en el servicio de referencia del Apéndice A, no citados de un benchmark.

El contrato de renovación

Esta es la parte que la mayoría de los registros de decisión omite, y la parte que convierte el registro en un contrato.

CAMPOS DEL CONTRATO DE RENOVACIÓN

- **Resultados esperados:** qué mejorará de forma medible, por ejemplo la latencia de cola dentro del presupuesto en los picos, o menos hallazgos de seguridad de memoria en un analizador sintáctico que reemplazó código C.
- **Línea base:** las mismas mediciones, tomadas antes del cambio, con el método por escrito.
- **Fecha de revisión:** una fecha de calendario fijada por el responsable (un plazo de seis a doce meses es un supuesto de planificación, no una regla), con la revisión en la agenda del responsable.
- **Responsable:** el rol que dirige la revisión y decide renovar, ajustar o revertir.
- **Disparadores de reversión:** condiciones acordadas de antemano, como puestos sin cubrir durante dos ciclos de contratación, tiempos de compilación por encima de un presupuesto declarado, o resultados no cumplidos en dos revisiones.
- **Ruta de salida:** cómo saldrían los servicios de Rust si se activa un disparador: interfaces neutrales respecto al lenguaje, una pila tecnológica de destino definida y una estimación del esfuerzo.

EJEMPLO: UNA REVISIÓN QUE AJUSTA

Considere un equipo cuyo registro prometía latencia de cola dentro del presupuesto para tres servicios. En la primera revisión, dos servicios cumplen el resultado y uno, un servicio mayormente CRUD, no muestra ganancia y sí una entrega más lenta. El responsable renueva Rust para los dos servicios de ruta crítica y ajusta el alcance: los nuevos servicios CRUD vuelven a la pila administrada habitual de la organización. Eso demuestra que el registro funciona, no que falla.

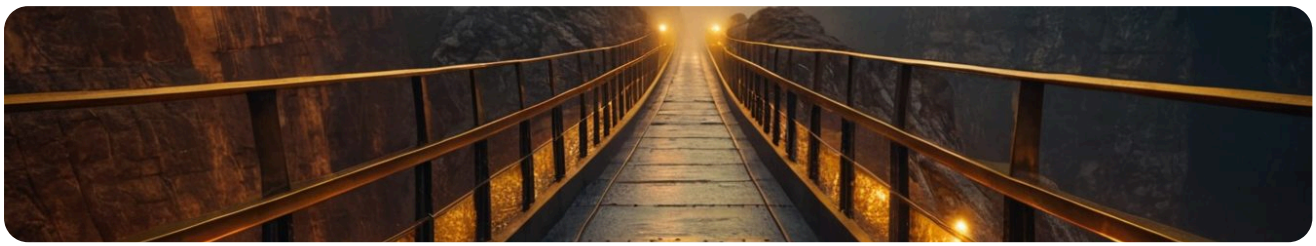
Una ruta de salida es más fácil cuando las interfaces son neutrales respecto al lenguaje desde el principio: contratos HTTP o gRPC versionados, esquemas en un registro compartido y ningún servicio que dependa de los tipos internos de otro. El capítulo nueve trata esa promesa en detalle.

LISTA DE VERIFICACIÓN DEL CONTRATO

- Existe un ADR antes de que el primer servicio Rust llegue a producción, con un único responsable.
- Los criterios y los pesos se firmaron antes de puntuar, y cada alternativa tiene una defensa equilibrada por escrito.
- Las líneas de costo total incluyen contratación, incorporación, CI y hardware de compilación, mantenimiento de la cadena de herramientas y revisión de dependencias.
- Se midió una línea base con un método declarado antes del cambio.
- La fecha de revisión, los disparadores de reversión y la ruta de salida están escritos en el registro.

CAPÍTULO TRES

Cuándo no usar Rust



Lo más creíble que puede decir un defensor de Rust es dónde no debería usarse Rust. Un lenguaje que sirve para todo es un lenguaje que nadie ha evaluado. Decir que no a Rust en los lugares adecuados es lo que protege los lugares donde sí justifica su costo.

La versión corta para un CTO: Rust encaja donde vale la pena pagar por el control sobre la latencia, la memoria y el consumo de recursos, y donde el código vivirá lo suficiente como para amortizar la curva de aprendizaje. Suele ser la elección equivocada para prototipos, para aplicaciones de línea de negocio en un equipo que ya domina .NET o la JVM, y para interfaces de usuario. Muchas organizaciones terminan con un reparto: un núcleo en Rust para las rutas críticas y TypeScript para la interfaz. Cuando una biblioteca necesaria solo existe en otro lenguaje, aíslela detrás de un límite versionado con un responsable y una fecha de vencimiento. Y un equipo que no cuenta con un equipo de plataforma puede empezar con cinco contratos, enumerados al final de este capítulo.

La matriz de encaje

Dos preguntas ubican la mayoría de las cargas de trabajo. ¿Cuánto vale aquí el control: latencia de cola, consumo de memoria, seguridad en los puntos donde se analizan entradas no confiables? ¿Y cuánto costará aquí la adopción: habilidades del equipo, bibliotecas que faltan, presión por entregar este trimestre?

FIGURA 3.1

Matriz de encaje: valor del control frente al costo de adopción



Ubique primero la carga de trabajo y luego el equipo. El mismo servicio puede quedar en cuadrantes distintos para equipos distintos. La matriz es un insumo para el registro de decisión del capítulo dos, no un sustituto.

Tres ubicaciones merecen una mirada más detenida, porque es en ellas donde las organizaciones se equivocan con más frecuencia.

Prototipos. Un prototipo existe para aprender algo rápido y luego desecharse. El compilador de Rust pide decisiones sobre propiedad y manejo de errores que un prototipo todavía no necesita. Construya la prueba de concepto con lo que el equipo escriba más rápido. Si la idea sobrevive y la carga de trabajo cae en la mitad derecha de la matriz, escriba el servicio de producción en Rust con lo que la prueba le enseñó.

Equipos con amplia experiencia en .NET o en la JVM. Un equipo con años de C# o Java, un pipeline de despliegue que funciona, perfiladores en los que confían los operadores y una cartera pendiente de funcionalidades de negocio tiene mucho que perder. Para servicios CRUD, motores de flujo de trabajo y código de integración, el entorno de ejecución administrado suele ser la mejor herramienta. Su recolector de basura es un precio razonable por la productividad, y NativeAOT o GraalVM native image pueden reducir buena parte de la diferencia en tiempo de arranque y consumo de recursos donde eso importa. Rust aún puede ganarse un lugar para un componente crítico, invocado a través de una interfaz limpia.

Interfaces de usuario. Rust puede compilar a WebAssembly, y existen frameworks de interfaz en Rust. Pero para la mayoría de los front ends empresariales, TypeScript tiene los frameworks, las bibliotecas de componentes, las herramientas de diseño y las personas.

Elegir Rust para la interfaz suele sacrificar un ecosistema amplio a cambio de una ganancia pequeña.

Las cargas de trabajo cambian de cuadrante. Una ubicación no es permanente. Un servicio interno de gestión (back office) puede convertirse en una ruta crítica cuando un nuevo canal le envía mucho más tráfico, y un piloto de Rust puede quedarse en el cuadrante costoso porque el equipo nunca recibió la formación prometida. Las ubicaciones se vuelven a verificar en la fecha de revisión del registro de decisión, con las mismas dos preguntas y evidencia nueva.

Una división común: núcleo en Rust, interfaz en TypeScript

Muchos equipos adoptan un patrón que aprovecha ambas fortalezas. El núcleo que hace el trabajo pesado, como el análisis sintáctico, el enrutamiento, la compresión, la búsqueda o la evaluación de políticas, es Rust. La interfaz y los servicios ligeros que están justo detrás son TypeScript. Ambos se comunican mediante un contrato versionado.

- **A través de la red:** una API HTTP o gRPC con un esquema bajo control de versiones. Los tipos para TypeScript se generan a partir del esquema, no se escriben a mano, de modo que los dos lados no pueden divergir sin que se note.
- **En el navegador:** una biblioteca Rust compilada a WebAssembly para una tarea pesada, con una interfaz pequeña y tipada que la UI invoca.
- **En el escritorio:** un proceso Rust con una interfaz de tecnología web encima, que también se comunica mediante un formato de mensajes versionado.

La división funciona porque cada lado se gobierna con los contratos que le corresponden, y el límite entre ambos es en sí mismo un contrato con un responsable. El capítulo nueve trata en detalle esa promesa de interfaz.

La excepción por carencia del ecosistema

Tarde o temprano, un entorno tecnológico basado en Rust necesita algo que solo existe en otro lenguaje: un SDK de proveedor, un formato de documento, una biblioteca de protocolo con años de correcciones detrás. Las respuestas equivocadas son escribir un binding rápido y esperar lo mejor, o abandonar el registro de decisión y dejar que cada equipo elija. La respuesta correcta es una excepción gobernada.

EJEMPLO HIPOTÉTICO: UN FORMATO DE INFORMES SIN BIBLIOTECA EN RUST

Considere un equipo cuyo servicio Rust debe producir documentos en un formato que solo una biblioteca .NET madura maneja correctamente. El equipo no enlaza esa biblioteca mediante FFI dentro del proceso Rust. Ejecuta un pequeño servicio .NET junto al servicio Rust y se comunica con él mediante un protocolo versionado: un esquema de solicitud, un esquema de respuesta y un número de versión en cada mensaje. El registro de la excepción nombra un responsable, el motivo, el límite y una fecha de vencimiento en la que se revisará: se renueva si ninguna opción en Rust ha madurado, se retira si alguna lo ha hecho.

EL PATRÓN DE EXCEPCIÓN

- **Aislar:** poner el código externo detrás de un límite de proceso o de protocolo, no dentro del espacio de direcciones de Rust, salvo que la FFI sea la única opción y pase por el contrato de excepción del capítulo cinco.
- **Versionar:** el límite lleva una versión explícita, para que cualquiera de los lados pueda cambiar sin sorprender al otro.
- **Asignar un responsable:** un responsable con nombre para la excepción, distinto del responsable del servicio Rust.
- **Vencer:** toda excepción tiene una fecha de revisión, y una excepción vencida es visible en el registro.

Esto mantiene honesta la excepción. Está permitida, es visible y no se convierte discretamente en una segunda plataforma.

Gobernanza mínima viable

La mayor parte de este libro supone un equipo de plataforma responsable de la cadena de herramientas (toolchain), las plantillas de CI y el registro de contratos. Muchas organizaciones no tienen uno. Un equipo pequeño que adopta Rust aún puede gobernarlo, con un conjunto inicial de cinco contratos lo bastante pequeños como para implantarse antes de que el primer servicio llegue a producción. El capítulo doce muestra cómo crecer desde este nivel hasta una gobernanza completa.

Contrato inicial	Nivel	Qué requiere
1. Un registro de decisión	Personas	El ADR del capítulo dos, con un responsable y una fecha de revisión
2. Una cadena de herramientas y un lockfile fijados	Herramientas	rust-toolchain.toml y un Cargo.lock incluido y versionado en cada repositorio; las bibliotecas también se prueban periódicamente contra las últimas versiones compatibles
3. Un control de CI que bloquea	Herramientas	Formato, lints como errores, pruebas, todo con el uso del lockfile obligatorio
4. Admisión de dependencias	Herramientas	Avisos y licencias verificados en cada compilación
5. Sin unsafe por defecto	Compilador	unsafe_code prohibido en los crates de aplicación; cualquier excepción registrada con un responsable

En la práctica, los tres primeros contratos ocupan unas pocas líneas cada uno. A continuación hay un breve extracto; las plantillas completas, con la versión de la cadena de herramientas con la que se verificaron, están en el Apéndice A.

```
# rust-toolchain.toml: fija la versión estable exacta que aprobó la organización
[toolchain]
channel = "1.100.0"
components = ["clippy", "rustfmt"]

# Cargo.toml: convierte "sin unsafe" en una regla del compilador, no en una convención
[lints.rust]
unsafe_code = "forbid"

# Control de CI: cada paso debe pasar, y un fallo bloquea la fusión
cargo fmt --check
cargo clippy --all-targets --locked -- -D warnings
cargo test --locked
cargo deny check advisories licenses
```

La versión mostrada es solo un ejemplo; lo importante es que la versión está fijada y se cambia de forma deliberada. Rust 1.100.0 es la versión estable actual en el momento de escribir esta edición (publicada el 2026-11-12, según las notas de versión de Rust). La edición 2024 es estable desde Rust 1.85, lo cual importa cuando un equipo decide qué edición usan sus crates (capítulo siete). Observe lo que el conjunto inicial deja fuera: revisión de dependencias de la cadena de suministro, firma y atestación, un registro de unsafe, verificaciones de API, presupuestos de ejecución. Eso llega más adelante, a medida que crece el entorno tecnológico. El conjunto inicial basta para que la decisión sea visible, la compilación sea reproducible y los fallos evidentes bloqueen el avance.

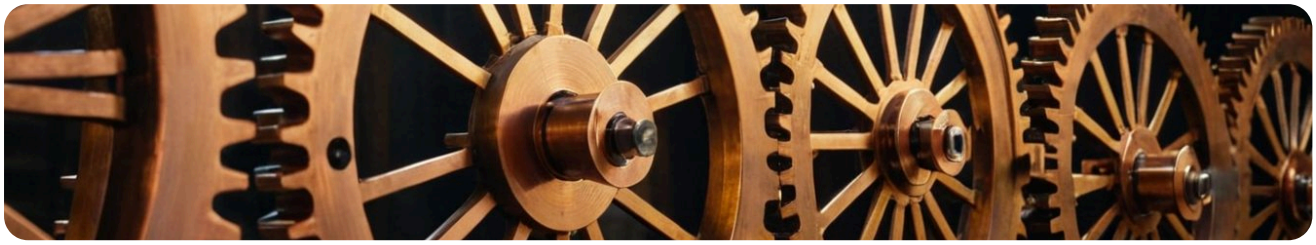
Decir que no también es parte de la gobernanza. Un registro de decisión que dice "Rust para estas cargas de trabajo, no para aquellas" es más sólido que uno que dice "Rust en todas partes", porque puede verificarse, y porque los equipos que están del otro lado de la línea pueden ver que su pila se tomó en serio.

LISTA DE VERIFICACIÓN DEL CONTRATO

- Cada carga de trabajo candidata se ubica en la matriz de encaje antes de elegir un lenguaje, y la ubicación consta en el ADR.
- Los prototipos se construyen en la pila más rápida y se reescriben de forma deliberada, no se promueven por accidente.
- Donde Rust y TypeScript se encuentran, el límite es un esquema versionado y los tipos del cliente se generan.
- Cada excepción por carencia del ecosistema está aislada tras un límite versionado, con un responsable y una fecha de vencimiento.
- Los equipos sin equipo de plataforma aplican los cinco contratos iniciales desde el primer día.

CAPÍTULO CUATRO

El contrato del compilador



La mayor parte de la gobernanza actúa después de escribir el código: una revisión, un escaneo, una prueba en preproducción. Rust adelanta una parte a la propia compilación. El compilador se niega a producir un binario cuando se rompen ciertas reglas, y nadie puede saltarse ese paso por estar ocupado. Este capítulo trata de usar esa negativa de forma deliberada. El contrato es sencillo de enunciar: las reglas importantes del negocio viven en los tipos, cada fallo es un valor que alguien debe manejar, y las caídas siguen una política escrita. El riesgo que elimina es una clase de defectos que, de otro modo, llegarían a producción sin que nadie lo advirtiera: identificadores confundidos, estados imposibles, errores ignorados y uso compartido inseguro entre hilos. El costo también es real: compilaciones más lentas, un período de aprendizaje para el equipo (un supuesto de planificación que conviene medir) y trabajo de diseño por adelantado que algunos equipos preferirían posponer.

Si usted es CTO, lo único que debe retener es el límite. Dentro del Rust seguro, el desarrollador propone y el compilador decide, pero solo sobre la memoria, los tipos y las carreras de datos. No decide nada sobre interbloqueos, lógica de negocio o lo que ocurre cuando una solicitud se cancela a mitad de camino. Eso queda en manos de las herramientas y de las personas, y este capítulo termina diciéndolo con claridad.

FIGURA 4.1

La puerta del código, dividida en sus tres niveles



La columna del compilador es corta deliberadamente. Es la única columna con demostraciones. Las otras dos necesitan responsables, controles y evidencia, que es lo que aporta el resto de la Parte II.

Los tipos como contratos: hacer irrepresentables los estados inválidos

Un tipo es una promesa que el compilador verifica en cada punto de llamada, en cada compilación, para siempre. Eso convierte a los tipos en el contrato más barato que una organización puede redactar. La regla práctica es antigua y sigue siendo buena: hacer irrepresentables los estados inválidos. Si un valor nunca debe confundirse con otro, dele su propio tipo. Si algo solo puede estar en uno de unos pocos estados, expréselo en el tipo, y deje que el compilador rechace todo lo demás.

```
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct CustomerId(u64);

#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub struct InvoiceId(u64);

pub fn refund(customer: CustomerId, invoice: InvoiceId) { /* ... */ }
// refund(invoice, customer) no compila: los tipos son distintos.
```

Este es el patrón newtype. No cuesta nada en tiempo de ejecución, porque el envoltorio desaparece al compilar, y elimina toda una familia de errores en los que se intercambian dos números de la misma forma. La misma idea se extiende a los estados. Un enum recoge cada estado en el que puede estar algo, y cada estado lleva solo los datos que tienen sentido para él.

```
pub enum Payment {
    Pending,
    Settled { reference: String },
    Refunded { reference: String, reason: String },
}

pub fn can_ship(p: &Payment) -> bool {
    match p {
        Payment::Settled { .. } => true,
        Payment::Pending | Payment::Refunded { .. } => false,
    }
}
```

Un pago pendiente no tiene referencia de liquidación, así que ningún código puede leerla por error. Y cuando alguien agrega un estado nuevo, por ejemplo `Disputed`, cada `match` que no lo maneja deja de compilar. Así, el contrato hace parte del trabajo de revisión por usted: el cambio no puede fusionarse hasta que cada punto de decisión haya tomado una decisión. Hay una forma de debilitarlo. Una rama comodín (`_ => false`) absorbe en silencio los estados nuevos. Para los enums de dominio, muchos equipos lo prohíben con el lint de Clippy `wildcard_enum_match_arm`, que está desactivado por defecto y debe activarse expresamente. Los enums públicos de bibliotecas compartidas son un caso distinto, tratado junto con las promesas de interfaz en el capítulo nueve.

Propiedad: quién puede cambiar qué, y cuándo

La propiedad (ownership) es la regla según la cual cada valor tiene un único propietario, y en cualquier momento puede haber muchos lectores o un solo escritor, nunca ambos. Los tiempos de vida (lifetimes) son la forma en que el compilador verifica que una referencia prestada nunca sobreviva a aquello a lo que apunta. Desde el punto de vista de la gobernanza, el significado es sencillo. La pregunta "¿quién puede cambiar estos datos ahora mismo?" tiene una respuesta que el compilador hace cumplir, en lugar de una respuesta escrita en un comentario. De ahí provienen la seguridad de memoria y la ausencia de carreras de datos del Rust seguro. También es la razón por la que espero que los primeros meses sean más lentos, un supuesto de planificación que hay que medir en el piloto: el equipo aprende a declarar la propiedad desde el principio, donde otros lenguajes permiten dejarla implícita hasta que algo se rompe.

Los errores como valores, y una política de pánico

Rust no tiene excepciones que viajen de forma invisible por la pila. Una función que puede fallar devuelve `Result`, y quien la llama tiene que hacer algo con él. `Result` está marcado con `#[must_use]`, así que ignorar uno genera una advertencia del compilador, y el operador `?` pasa un error a quien llama con un único carácter visible.

```rust
pub struct Name(String);
pub struct Record { pub name: Name }

#[derive(Debug)]
pub enum ParseError { Empty, TooLong(usize) }

pub fn parse_name(input: &str) -> Result<Name, ParseError> {
    let s = input.trim();
    if s.is_empty() { return Err(ParseError::Empty); }
    if s.len() > 64 { return Err(ParseError::TooLong(s.len())); }
    Ok(Name(s.to_owned()))
}

pub fn load(input: &str) -> Result<Record, ParseError> {
    let name = parse_name(input)?; // el error pasa a quien llama
    Ok(Record { name })
}
```

Cada ruta de fallo está ahora en la firma, lo que significa que está en la revisión de código. Los pánicos son la otra vía. Un pánico significa "esto debería ser imposible", y detiene el hilo. Por defecto, la pila se desenrolla (unwinding); con `panic = "abort"` en el perfil de release, todo el proceso se detiene de inmediato. El problema empieza cuando se usan pánicos para gestionar entradas incorrectas habituales, a menudo mediante `unwrap()`. Entonces una sola solicitud malformada puede hacer caer un proceso de trabajo (worker), y un atacante que la descubra tiene a su alcance una denegación de servicio.

FIGURA 4.2

Dos vías para el fallo, y quién decide



El fallo esperado es un dato; un pánico es un informe de error. El servicio puede optar por abortar y reiniciarse desde un estado limpio. La biblioteca no puede decidir por quienes la llaman.

Por eso la política de pánico necesita dos reglas, no una. Para un binario de servicio, `panic = "abort"` es aceptable cuando la plataforma reinicia el proceso y el trabajo en curso puede reintentarse con seguridad. Hace que un invariante roto sea evidente y que la ruta de recuperación sea uniforme, y puede reducir el tamaño del binario. Dos advertencias: con abort, los destructores no se ejecutan, así que los registros y las trazas almacenados en búfer deben vaciarse explícitamente por diseño, sin depender de la terminación del proceso; y Cargo ignora este ajuste para pruebas, scripts de compilación y macros procedurales. Para el código de biblioteca, la regla es más estricta: ninguna entrada debe poder provocar un pánico. Una biblioteca no sabe si quien la llama aborta, desenrolla la pila o está atendiendo a miles de otros usuarios en el mismo proceso. Devuelve `Result` y deja que decida quien la llama.

```
# Binario de servicio: Cargo.toml
[profile.release]
panic = "abort"          # solo binarios de servicio
overflow-checks = true   # activación explícita: el desbordamiento provoca panic también en
release

[lints.clippy]
unwrap_used = "warn"
expect_used = "warn"
```

```
# Crate de biblioteca: Cargo.toml (ningún panic alcanzable desde la entrada)
[lints.clippy]
unwrap_used = "deny"
expect_used = "deny"
panic = "deny"
indexing_slicing = "warn"
```

Estos niveles son un punto de partida, no un estándar. Las plantillas completas están en el Apéndice A, que tiene las mismas dos tablas: el binario de servicio y la biblioteca (biblioteca: `unwrap_used`, `expect_used` y `panic` denegados). Las excepciones pasan por `#[expect(lint, reason = "...")]`, estable desde Rust 1.81, de modo que cada excepción lleva su motivo en el código y advierte cuando ya no es necesaria. El capítulo cinco da a esas excepciones un responsable y un vencimiento.

Send, Sync y async, a nivel de gobernanza

Dos traits marcadores sostienen el contrato de concurrencia. Un tipo es `Send` si puede moverse a otro hilo, y `Sync` si puede compartirse entre hilos por referencia. El compilador los deduce y se niega a compilar código que los incumpla. Un valor con recuento de referencias para un solo hilo (`Rc`) no puede transferirse a otro hilo por accidente. Un valor

simple no puede modificarse desde dos hilos sin un bloqueo (lock) o una operación atómica. Esta es la ausencia de carreras de datos de la primera columna de la Figura 4.1, y es una demostración, no una convención.

La programación asíncrona añade una capa que no pertenece al lenguaje. Rust define los futures y la sintaxis `async`, pero el planificador que los ejecuta proviene de una biblioteca: el entorno de ejecución asíncrono (async runtime). La mayoría de los servicios de red usan uno, y la elección determina el modelo de hilos, los temporizadores, la E/S, qué crates encajan entre sí y cómo depuran los equipos en producción. Eso convierte la elección del entorno de ejecución en una decisión de toda la organización, no en una preferencia de cada equipo. El capítulo siete lo incluye en el contrato de dependencias como crate estándar de la organización. Aquí corresponde una regla práctica: mantener la guarda de un bloqueo (MutexGuard) durante un `.await` es una causa habitual de esperas prolongadas, y el lint de Clippy `await_holding_lock` detecta el caso de la biblioteca estándar.

Lo que el compilador no cubre

> **Para recordar los límites:** el Rust seguro no impide los interbloqueos: dos hilos pueden esperar indefinidamente el bloqueo del otro. No impide las carreras lógicas, en las que cada acceso está sincronizado pero el orden de los eventos sigue siendo incorrecto. No hace que el código asíncrono sea seguro ante la cancelación: un future descartado en un `.await` puede dejar trabajo a medias, y la documentación del entorno de ejecución sobre la seguridad ante la cancelación debe leerse operación por operación. No impide los pánicos, las fugas de memoria ni quedarse sin memoria. Y el desbordamiento de enteros provoca pánico en compilaciones de depuración, pero en compilaciones de release se desborda sin aviso y continúa con aritmética modular (wrapping), salvo que se active `overflow-checks` o el código use deliberadamente aritmética `checked_`, `saturating_` o `wrapping_`.

Nada de esto debilita el argumento. Lo acota. El compilador retira de la mesa de revisión una amplia clase de defectos, para que las personas sentadas a esa mesa puedan dedicar su atención a las clases que deja atrás. Otros lenguajes trazan esta línea en lugares distintos, y el Apéndice B la proyecta para TypeScript, Go, .NET y Java.

> *El desarrollador propone, el compilador decide: sobre la memoria, los tipos y las carreras de datos. Cada línea se sostiene solo hasta donde llega esa demostración. Más allá, el contrato necesita responsables.*

Ficha de contrato

Campo	El contrato del compilador
Contrato	Las reglas del dominio viven en los tipos; los fallos son valores; los pánicos siguen una política escrita según el tipo de crate
Amenaza que aborda	Identificadores intercambiados, estados imposibles, errores ignorados, caída ante entradas incorrectas, uso compartido inseguro, desbordamiento silencioso
Responsable	El equipo de plataforma es responsable de la línea base de lints y perfiles; el responsable de cada crate lo es de sus tipos de dominio
Punto de aplicación	El compilador en cada compilación; Clippy en niveles deny en el control de CI; perfil de release del workspace
Comportamiento ante fallos	La compilación falla y el cambio no puede fusionarse
Autoridad de excepción y vencimiento	`#[expect]` local con un motivo, aprobado por el responsable del crate; los cambios de la línea base, por el equipo de plataforma; vencimiento según el ciclo de vida del capítulo cinco
Evidencia de que el control se ejecutó	Registros de CI de la compilación y de Clippy con la cadena de herramientas (toolchain) fijada; el número y la antigüedad de los atributos `expect` y `allow`
Evidencia de que el resultado mejoró	Caídas en producción por pánicos e incidentes por estados inválidos, frente a la línea base del registro de decisión
Frecuencia de revisión	En cada actualización de la cadena de herramientas, y trimestralmente para la línea base de lints

LISTA DE VERIFICACIÓN DEL CONTRATO

- Los identificadores y las cantidades que no deben mezclarse tienen sus propios tipos, y los estados de dominio son enums sin ramas comodín.
- Los crates de biblioteca deniegan `unwrap_used`, `expect_used` y `panic` en rutas de código alcanzables desde la entrada.
- Cada servicio registra su estrategia de pánico y, si aborta, cómo se vacían los registros y las trazas y cómo se reinicia el proceso.
- Los perfiles de release declaran `overflow-checks` explícitamente, y la aritmética de dinero o de cuotas usa operaciones con comprobación de desbordamiento (checked_).
- El entorno de ejecución asíncrono se designa una sola vez para toda la organización, en el contrato de dependencias (capítulo siete).
- El equipo puede enumerar, sin mirar, lo que el compilador no demuestra.

CAPÍTULO CINCO

El contrato de excepciones: unsafe, FFI y código en tiempo de compilación

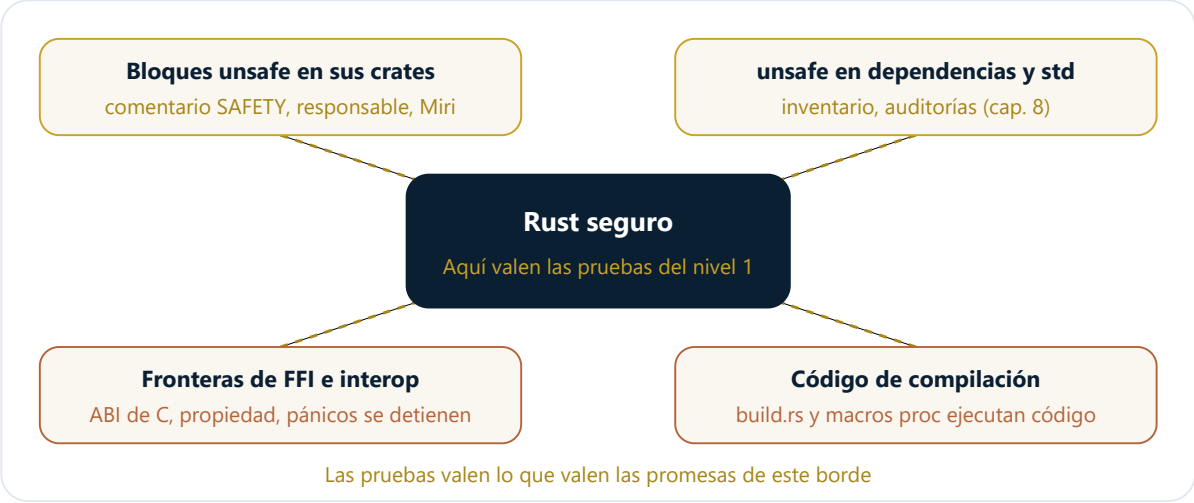


Cada prueba del capítulo cuatro descansa sobre código que el compilador no puede verificar. La biblioteca estándar usa `unsafe` por dentro. También lo hacen muchos crates populares. Las llamadas a C, C# o una JVM cruzan una línea más allá de la cual el compilador no ve, y los scripts de compilación ejecutan código arbitrario en la máquina de compilación antes de que se compile nada. Nada de eso es un escándalo. Es la forma en que un lenguaje de sistemas llega al hardware y al resto del mundo. El contrato de este capítulo trata cada uno de esos puntos como una excepción firmada: una persona designada declara qué debe cumplirse, otra lo verifica, y la excepción vence en una fecha fijada, como cualquier otra. El riesgo que elimina es el código inseguro sin responsable, ese que nadie recuerda haber aprobado. El costo es tiempo de revisión, algunas herramientas que solo funcionan en nightly y la disciplina de mantener un registro al día.

Si usted es CTO, quédese con una frase: las garantías de Rust seguro solo son tan sólidas como las promesas que se hacen en su borde, así que cada promesa en ese borde necesita un responsable y evidencia.

FIGURA 5.1

El límite de confianza y sus cuatro bordes



Cuatro bordes, una regla. Los bordes dorados tienen que ver con la memoria: una persona demuestra lo que el compilador no puede. Los bordes cobrizos tienen que ver con la confianza: código externo se ejecuta con sus privilegios o al otro lado de una línea que ningún tipo cruza.

Qué significa unsafe, y qué no significa

Un bloque `unsafe` no desactiva el verificador de préstamos (borrow checker) ni el sistema de tipos. El libro de Rust enumera las cinco operaciones principales que permite: desreferenciar un puntero crudo (raw pointer), llamar a una función unsafe (incluida una externa), leer o escribir una variable static mutable, implementar un trait unsafe y leer un campo de una unión. Dentro del bloque, el desarrollador asume una tarea que correspondía al compilador: demostrar que esas operaciones no pueden causar comportamiento indefinido. Esa es exactamente la forma de una excepción. El contrato se suspende por una razón declarada, y alguien firma y responde por ella.

El lenguaje ya aporta la mayor parte del papeleo. Por convención, una función unsafe documenta sus precondiciones bajo un encabezado `# Safety`, y cada bloque unsafe lleva un comentario `// SAFETY:` que explica por qué sus precondiciones se cumplen en ese punto.

```
/// Devuelve el primer byte sin comprobar los límites.
///
/// # Safety
/// `buf` no debe estar vacío.
pub unsafe fn first_unchecked(buf: &[u8]) -> u8 {
    // SAFETY: quien llama garantiza que buf.len() > 0,
    // así que el índice 0 está dentro de los límites.
    unsafe { *buf.get_unchecked(0) }
}
```

El bloque `unsafe` interior es obligatorio en espíritu y, desde la edición 2024 (Rust 1.85), su ausencia genera una advertencia por defecto: el lint `unsafe_op_in_unsafe_fn` exige un bloque explícito incluso dentro de una función unsafe, de modo que cada operación de riesgo queda marcada donde ocurre. La edición 2024 también exige `unsafe extern` en las declaraciones externas, lo que somete la firma de cada función externa a la misma regla. Dentro de un bloque así, un elemento puede marcarse como `safe`, y el código que lo llama puede usarlo entonces sin un bloque `unsafe`; llamar a los elementos sin marca sigue siendo unsafe. La marca `safe` es en sí misma una promesa no verificada sobre el código externo, así que necesita la misma revisión que cualquier bloque unsafe.

Convertir la convención en un control

Una convención se convierte en contrato cuando la compilación falla sin ella. Tres ajustes de lint hacen la mayor parte del trabajo, configurados a nivel de crate o de workspace en la tabla `[lints]`.

```
[lints.rust]
unsafe_code = "forbid"            # crates sin motivo para usar unsafe
unsafe_op_in_unsafe_fn = "deny"

[lints.clippy]
undocumented_unsafe_blocks = "deny"
```

La diferencia entre los dos niveles estrictos es importante. `forbid` no puede anularse en niveles inferiores, así que es adecuado para la gran mayoría de los crates, que no tienen motivo para manipular memoria directamente. `deny` puede relajarse con un atributo local, así que es adecuado para los pocos crates que realmente necesitan unsafe: en ellos se mantiene denegado a nivel de crate y se permite solo en un único módulo designado, que es al que apunta la entrada del registro. El lint de Clippy `undocumented_unsafe_blocks` pertenece al grupo opcional de restricciones; genera un error por cada bloque unsafe sin comentario `SAFETY`. La plantilla común de lints del Apéndice A fija `unsafe_code` en `deny`, de modo que un crate que necesita unsafe puede permitirlo en un único módulo designado; los crates que nunca necesitan unsafe lo endurecen a `forbid`, como en el extracto anterior.

El registro de unsafe

Los lints demuestran que el comentario existe. No pueden demostrar que sea cierto. Eso requiere una persona y cierta evidencia, y lo que deja constancia de ambas es el registro de unsafe: una fila por módulo unsafe, guardada junto al código para que cambie en la misma revisión.

Campo	Ejemplo (hipotético)
Ubicación	crates/codec/src/simd.rs, módulo `fast_path`
Responsable	Un responsable técnico designado del equipo de codec
Invariante	La longitud de la entrada es múltiplo de 16; los punteros provienen de un slice todavía válido
Por qué unsafe	Instrucciones vectoriales; según la medición en el benchmark de referencia, la versión segura es más lenta
Revisor	Un segundo ingeniero formado en el Rustonomicon, no el autor
Evidencia	`cargo miri test -p codec` sin errores; objetivo de fuzzing ejecutado de forma programada, sin hallazgos
Vencimiento	Una fecha, y antes si una API segura de la biblioteca estándar cubre el caso

Miri es el intérprete del proyecto Rust para detectar comportamiento indefinido, como accesos fuera de límites, uso después de liberar y valores inválidos. Solo funciona en nightly, verifica solo los caminos que ejecutan las pruebas y no puede ejecutar la mayoría de las llamadas externas. Los sanitizers como AddressSanitizer siguen siendo opciones inestables del compilador en Rust, así que también suelen necesitar una cadena de herramientas (toolchain) nightly. Por eso la evidencia es "las pruebas que ejercitan este módulo pasan bajo Miri", no "está demostrado que este módulo es sólido (sound)". El capítulo seis incluye ambas herramientas en el contrato de verificación.

El unsafe que usted no escribió

`forbid(unsafe_code)` se aplica a su crate, no a sus dependencias. Un servicio sin unsafe propio puede ejecutar mucho unsafe a través de los crates que incorpora, y eso es normal: un unsafe bien revisado en un crate muy usado suele ser más seguro que una alternativa segura hecha en casa. El contrato consiste en saber, no en llegar a cero. La herramienta comunitaria `cargo-geiger` es una forma de verlo: informa del uso de unsafe de cada dependencia de la compilación. Las decisiones que vienen después (qué crates son de confianza, quién los auditó y cómo se admite uno nuevo) corresponden al control de admisión del capítulo ocho.

FFI: la línea que ningún tipo cruza

La interoperabilidad es el punto donde Rust se gana su lugar en un entorno tecnológico existente, y donde termina lo que el compilador puede ver. El ABI de C es el idioma común: casi cualquier otro entorno de ejecución puede llamar a una función de C, y Rust puede exportar una.

Dirección	Camino habitual	Punto del contrato
C o C++ llama a Rust	Funciones `extern "C"`; `cbindgen` genera el encabezado	El encabezado se genera, nunca se escribe a mano
Rust llama a C	`bindgen` genera declaraciones a partir de encabezados de C	Las llamadas son unsafe salvo que una declaración esté marcada como `safe`; cada declaración, marcada o no, necesita una entrada en el registro
C# llama a Rust	ABI de C más P/Invoke	Queda por escrito quién libera la memoria
La JVM llama a Rust	JNI, o la Foreign Function and Memory API de Java	Los hilos se vinculan y se desvinculan de la JVM según una regla definida
Kotlin, Swift, Python	UniFFI (Mozilla) genera bindings a partir de una sola definición	Definición de la interfaz versionada (capítulo nueve)
TypeScript, navegador	WebAssembly con `wasm-bindgen`	La frontera es el sandbox, no el tipo

Tres reglas valen para cada fila. Primera, la memoria asignada en un lado se libera en ese mismo lado; una biblioteca de Rust que entrega un búfer también exporta la función que lo libera. Segunda, un pánico no debe cruzar la frontera. Desde Rust 1.81, un pánico que se propaga fuera de una función `extern "C"` aborta el proceso, lo cual es seguro pero brusco, así que las funciones de frontera capturan el fallo con `std::panic::catch_unwind` y devuelven un código de error en su lugar. Eso solo funciona cuando los pánicos desenrollan la pila (unwinding): `catch_unwind` no puede capturar un pánico en una compilación con `panic = "abort"`, donde el proceso termina en el momento del pánico. Una biblioteca pensada para ser llamada desde otros lenguajes se compila y se prueba con unwinding, o no debe entrar en pánico en ningún camino que llegue a la frontera. Tercera, los bindings generados son artefactos de compilación versionados, no archivos que alguien edita a mano. La interfaz es un contrato con otro equipo, y el capítulo nueve la trata como tal.

Código que se ejecuta mientras usted compila

Dos funcionalidades de Rust ejecutan código en la máquina de compilación. Un script `build.rs` se ejecuta antes de que se compile su paquete, a menudo para localizar una biblioteca de C o generar código. Una macro procedural se ejecuta dentro del compilador mientras se compila su código. Ambas se ejecutan con todos los permisos de quien inicia la compilación: pueden leer archivos, leer variables de entorno y acceder a la red. Eso las convierte en un riesgo de cadena de suministro en el sentido más literal. Una dependencia comprometida no necesita llegar a producción para hacer daño; basta con ejecutarse en el portátil de un desarrollador o en un runner de CI que guarda credenciales.

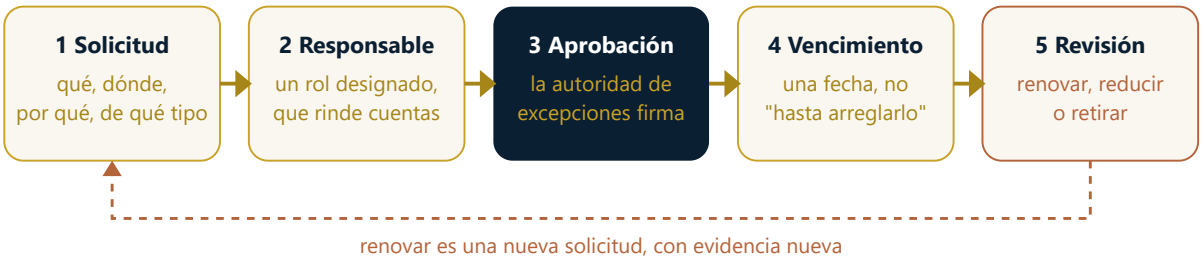
> **La máquina de compilación forma parte de la superficie de ataque.** Compile en runners de CI que no guarden secretos de larga duración. Descargue las dependencias en un paso y luego compile con `--locked --offline` para que Cargo mismo no acceda a la red. Esa opción gobierna solo a Cargo: no aísla los scripts de compilación, las macros procedurales ni los procesos que lanzan, que aún pueden abrir conexiones. Bloquee el acceso a la red durante el paso de compilación mediante el sandbox del runner o del contenedor, y trate `--offline` como una segunda verificación, no como el control. Mantenga un inventario de las dependencias que tienen scripts de compilación o que son macros procedurales, y revíselas primero en cada auditoría de dependencias.

Un solo ciclo de vida para cada excepción

Los módulos unsafe, las llamadas externas, los scripts de compilación, las excepciones de lint y las actualizaciones de emergencia de la cadena de herramientas parecen distintos, pero fallan de la misma manera: alguien los aprueba una vez y nadie vuelve a revisarlos. Por eso la organización aplica un solo ciclo de vida a todos ellos.

FIGURA 5.2

El ciclo de vida de una excepción



Aplica a: unsafe, FFI, scripts de compilación, excepciones de lint, cambios de toolchain

Una excepción que no puede vencer es un cambio de política disfrazado. El regreso al inicio es deliberado: la renovación nunca es automática.

Cada registro ocupa unas pocas líneas de texto en el repositorio, así que acompaña al código. La plantilla completa está en el Apéndice A; un extracto:

```
[[exception]]
kind = "unsafe"     # unsafe | ffi | build-script | lint-waiver | toolchain
location = "crates/codec/src/simd.rs"
owner = "codec team lead"
invariant = "input length is a multiple of 16"
approved_by = "platform security"
evidence = ["cargo miri test -p codec", "scheduled fuzz run, no findings"]
expires = "2027-05-31"
```

La revisión de degradación de los controles es el paso que la mayoría de los programas omite. El código que rodea una excepción cambia, el responsable pasa a otro equipo, una nueva API de la biblioteca estándar hace innecesario el unsafe. Por eso la revisión se activa con cualquiera de cuatro eventos: la fecha de vencimiento, un cambio en el archivo, una actualización de la cadena de herramientas o un cambio de responsable. El control de CI lee los registros y falla cuando uno ha vencido, lo que convierte un olvido en una compilación en rojo en lugar de un riesgo silencioso. La autoridad que aprueba la excepción depende del tipo: el responsable del crate para una excepción de lint, el equipo de plataforma para una actualización de la cadena de herramientas, y el equipo de seguridad para unsafe y scripts de compilación en todo lo que reciba entradas no confiables.

Ficha de contrato

Campo	El contrato de excepciones
Contrato	Cada módulo unsafe, frontera externa, script de compilación y excepción a una regla es una excepción firmada y con vencimiento
Amenaza que aborda	Comportamiento indefinido detrás de interfaces seguras; errores de memoria y de pánico en la FFI; código de compilación malicioso o descuidado
Responsable	El responsable nombrado en cada registro; el equipo de plataforma es responsable del ciclo de vida y de los lints
Punto de aplicación	Lints `unsafe_code`, `unsafe_op_in_unsafe_fn` y `undocumented_unsafe_blocks`; verificación en CI de los registros de excepción; compilaciones sin conexión y sin secretos
Comportamiento ante fallos	Un unsafe sin documentar o un registro vencido hace fallar la compilación
Autoridad de excepción y vencimiento	Según el tipo: responsable del crate, equipo de plataforma o equipo de seguridad; cada registro lleva una fecha
Evidencia de que el control se ejecutó	Resultados de lint en CI; registros de ejecución de Miri y de fuzzing para cada entrada; la salida de la verificación de vencimientos
Evidencia de que el resultado mejoró	Menos módulos unsafe con el tiempo; defectos de seguridad de memoria detectados antes de publicar la versión, y no después
Frecuencia de revisión	Con cada evento desencadenante (vencimiento, cambio de archivo, actualización de la cadena de herramientas, cambio de responsable) y una revisión trimestral del registro

LISTA DE VERIFICACIÓN DEL CONTRATO

- Los crates que no necesitan unsafe usan `forbid(unsafe_code)`; los demás lo deniegan fuera de un único módulo designado.
- Cada bloque unsafe tiene un comentario `SAFETY`, impuesto por un lint, y cada módulo unsafe tiene una fila en el registro con responsable, revisor y evidencia.
- Cada frontera externa declara quién libera la memoria y convierte los pánicos en errores antes de que crucen.
- Las compilaciones se ejecutan sin secretos de larga duración y sin acceso a la red después del paso de descarga.
- Cada registro de excepción tiene fecha de vencimiento, y el control de CI falla ante uno vencido.

CAPÍTULO SEIS

El contrato de verificación



Un pipeline en verde es una afirmación, y la mayoría de las organizaciones nunca comprueba si esa afirmación es cierta. El compilador cubre lo que describió el capítulo cuatro. Todo lo demás que promete la puerta del código, desde la política de lints hasta el comportamiento de un módulo unsafe, solo queda cubierto si alguna verificación se ejecuta, falla cuando debe y deja un registro. Ese es el contrato de verificación: cada promesa tiene una verificación, cada verificación se prueba a su vez con un cambio que debe hacerla fallar, y la organización mantiene separadas dos clases de evidencia: la prueba de que el control se ejecutó y la prueba de que los resultados mejoraron. El riesgo que elimina es la falsa confianza, el control que dejó de bloquear sin que nadie lo notara. El costo es tiempo de CI, algunos trabajos programados que solo funcionan en nightly y el hábito de romper a propósito el propio pipeline.

Si usted es CTO, haga a sus equipos una sola pregunta de este capítulo: ¿cuándo fue la última vez que cada control falló ante un cambio introducido a propósito para hacerlo fallar? Un control sin respuesta todavía no es evidencia.

FIGURA 6.1

La escalera de verificación: qué encuentra cada peldaño

Técnica	Qué encuentra	Versión, cuándo
Compilador	errores de tipo, propiedad, Send y Sync	estable, al compilar
Clippy y rustfmt	violaciones de política y patrones sospechosos	estable, cada PR
Unitarias, integración, doc	el comportamiento que alguien pensó verificar	estable, cada PR
Pruebas de propiedades	invariantes sobre entradas generadas	estable, cada PR
Fuzzing (cargo-fuzz)	fallos ante entradas que nadie imaginó	nightly, programado
Miri	comportamiento indefinido en unsafe ejecutado	nightly, programado
loom	intercalados de hilos erróneos, por modelo	estable, focalizado

Ninguna demuestra la ausencia de errores. Cada una deja evidencia de que se ejecutó.

El costo y la profundidad aumentan al bajar por la escalera. Las filas cobrizas necesitan la cadena de herramientas (toolchain) nightly, así que se ejecutan como trabajos programados sobre una versión nightly fijada, nunca como requisito de fusión sobre la nightly del día.

Lint y formato como política escrita

Clippy es el linter estándar de Rust, y rustfmt el formateador estándar. Trate a ambos como política, no como una cuestión de gustos. El formato no tiene una respuesta correcta, y precisamente por eso debe resolverlo una herramienta, una sola vez, para no volver a discutirlo nunca en una revisión. Los lints son distintos: cada nivel es una decisión sobre el riesgo. El grupo correctness de Clippy está denegado por defecto porque esos lints suelen indicar un error. Los grupos style, complexity, perf y suspicious emiten advertencias. Los grupos pedantic, restriction y nursery están desactivados salvo que usted los active, y el grupo restriction contiene los lints que codifican las reglas de la organización, como `unwrap_used` o `undocumented_unsafe_blocks` de los dos capítulos anteriores.

Desde Rust 1.74, Cargo lee los niveles de lint desde una tabla `[lints]` en `Cargo.toml`. Así la política deja de estar repartida entre atributos dispersos y opciones de línea de comandos y pasa a un único archivo revisado. Un workspace fija la política una vez y cada miembro la adopta.

```
# Cargo.toml del workspace
[workspace.lints.rust]
unsafe_code = "deny"

[workspace.lints.clippy]
all = { level = "warn", priority = -1 }
unwrap_used = "deny"

# Cargo.toml de cada crate miembro
[lints]
workspace = true
```

La prioridad negativa permite fijar primero el grupo completo y que los lints individuales lo anulen. En CI, `-D warnings` convierte las advertencias restantes en fallos, de modo que el archivo de política decide qué bloquea una fusión. Dos notas de gobernanza. Un cambio en este archivo es un cambio de política y lo revisa el equipo de plataforma; no lo fusiona cualquiera a quien un lint le resulte molesto. Además, las nuevas versiones de Clippy añaden lints, así que una actualización de la cadena de herramientas puede poner en rojo de un día para otro una compilación que estaba en verde; esa es una de las razones por las que el capítulo siete fija la cadena de herramientas y trata cada actualización como un cambio planificado.

```
cargo fmt --all --check
cargo clippy --workspace --all-targets --locked -- -D warnings
cargo test --workspace --locked
```

Estas tres líneas son el núcleo del control de fusión. La plantilla completa de CI está en el Apéndice A.

Pruebas, y qué demuestra cada tipo

Rust integra tres tipos de pruebas en Cargo. Las pruebas unitarias están junto al código y pueden probar funciones privadas. Las pruebas de integración están en el directorio `tests/` y solo ven la interfaz pública del crate, así que prueban el contrato del que depende el código que lo llama. Las pruebas de documentación son los ejemplos de código de los comentarios de documentación de un crate de biblioteca: `cargo test` los compila y los ejecuta, de modo que un ejemplo que deja de corresponder a la API real hace fallar la compilación. Así la documentación pasa a formar parte de la promesa de interfaz del capítulo nueve.

Los tres tipos comparten una limitación: verifican los casos que a alguien se le ocurrieron. Las pruebas de propiedades amplían ese alcance. En lugar de una sola entrada, la prueba declara una regla que debe cumplirse para cualquier entrada, y un generador prueba

muchas. `proptest` es un crate muy usado para esto, y reduce una entrada que falla a un ejemplo mínimo antes de notificarla.

```
use proptest::prelude::*;

proptest! {
    #[test]
    fn parse_never_panics(s in ".*") {
        let _ = parse_name(&s);
    }

    #[test]
    fn encode_roundtrips(n in any::<u64>()) {
        prop_assert_eq!(decode(&encode(n)), Some(n));
    }
}
```

La primera propiedad es la regla de pánico para bibliotecas del capítulo cuatro, escrita como prueba: ninguna entrada provoca un pánico. La segunda es la forma más común de todas, la ida y vuelta (codificar y decodificar), y es adecuada para parsers, codecs y formatos de serialización.

Fuzzing, Miri y loom

El fuzzing va más allá que las pruebas de propiedades. Un fuzzer guiado por cobertura observa qué caminos de código alcanza cada entrada y muta las entradas para alcanzar caminos nuevos, durante horas o días. `cargo-fuzz` controla libFuzzer y requiere la cadena de herramientas nightly. Un objetivo ocupa unas pocas líneas:

```
#![no_main]
use libfuzzer_sys::fuzz_target;

fuzz_target!(|data: &[u8]| {
    if let Ok(s) = std::str::from_utf8(data) {
        let _ = my_crate::parse_name(s);
    }
});
```

El fuzzing debe aplicarse a cada parser de entradas no confiables y a cada módulo unsafe del registro. Se ejecuta como trabajo programado, conserva su corpus entre ejecuciones y convierte cada fallo que encuentra en una prueba normal para que el error no pueda reaparecer.

Miri, presentado en el capítulo cinco, ejecuta el conjunto de pruebas en un intérprete que detecta comportamiento indefinido. Solo funciona en nightly, solo ve los caminos ejecutados y no puede ejecutar la mayoría de las llamadas externas, así que su evidencia se limita a las

pruebas que ejecutó. Donde no puede ejecutar el código, use en su lugar fuzzing, sanitizers, pruebas de modelo o una revisión focalizada. loom es un verificador de modelos para código concurrente: una prueba escrita con las versiones de loom de los hilos, los tipos atómicos y los bloqueos se ejecuta muchas veces, cada vez con un intercalado distinto, bajo el modelo de memoria de C11. Es la herramienta adecuada para una estructura lock-free escrita a mano o una primitiva de sincronización propia, y la equivocada para un servicio ordinario; encuentra intercalados erróneos en modelos pequeños y acotados. Una ejecución sin errores solo es evidencia para el modelo que exploró: el código bajo prueba, con las versiones de loom de las primitivas, dentro de los límites que fija la prueba (por ejemplo, un tope de expropiaciones). No dice nada sobre caminos de código, números de hilos o primitivas fuera de ese modelo. Ninguna de las dos herramientas detecta interbloqueos en la lógica de negocio ni condiciones de carrera lógicas entre servicios. Eso queda para la revisión de diseño y para operaciones.

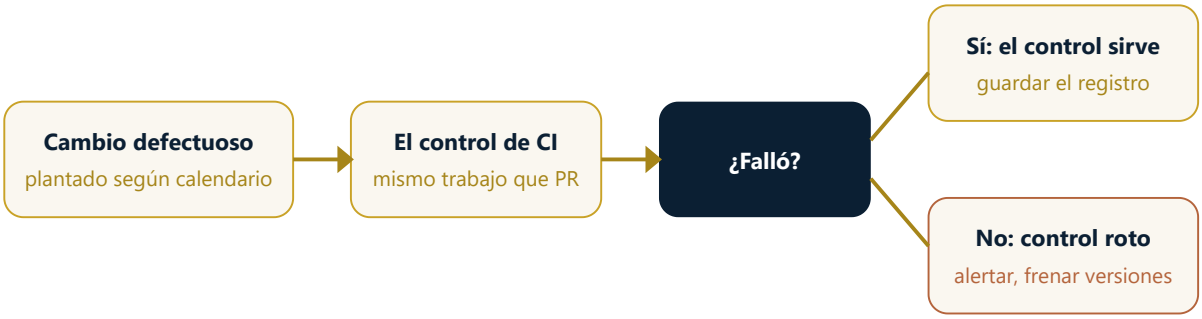
Verificación	Dónde es obligatoria	Evidencia que deja
fmt, Clippy, pruebas	Cada crate, cada fusión	Registro de ejecución de CI con la versión de la cadena de herramientas
Pruebas de propiedades	Parsers, codecs, reglas de dominio	Nombres de prueba asociados a invariantes
Fuzzing	Entradas no confiables, módulos del registro de unsafe	Horas ejecutadas, tamaño del corpus, hallazgos
Miri	Cada entrada del registro de unsafe, donde Miri admita el código; si no, fuzzing, sanitizers, pruebas de modelo o revisión focalizada	Registro de la ejecución programada de cada crate
loom	Primitivas de concurrencia propias	Resultados de las pruebas de modelo

El propio control debe probarse

Los controles dejan de funcionar sin que nadie lo note, y por motivos triviales. Un trabajo se marca como opcional durante una caída y nunca vuelve a ser obligatorio. Un filtro de rutas omite el crate que cambió. Un lint se permite al inicio de un archivo y se olvida. El pipeline sigue en verde, y el verde parece evidencia. La única prueba fiable de un control es un cambio que debe hacerlo fallar.

FIGURA 6.2

Probar el control con un cambio defectuoso conocido



Un control que nunca falla no es evidencia de nada.

La ejecución en rojo es la evidencia. Un trabajo programado abre un cambio que rompe cada regla a propósito y comprueba que el control lo rechaza. Si el control lo deja pasar, el incidente es el control.

En la práctica, un pequeño conjunto de cambios de comprobación (canarios) cubre las reglas que importan: un `unwrap()` en un camino de entrada de un crate de biblioteca, un bloque unsafe sin comentario `SAFETY`, un archivo mal formateado, una prueba que falla, un registro de excepción vencido, una dependencia que el control de admisión debería rechazar (capítulo ocho). Un trabajo programado aplica cada uno en una rama desechable, ejecuta el mismo control que las fusiones reales y deja constancia de que se puso en rojo. La misma idea se aplica a las propias pruebas. Las pruebas de mutación introducen pequeños cambios en el código, como invertir una comparación, y comprueban que alguna prueba falle; `cargo-mutants` es una herramienta comunitaria que hace esto para Rust. Un mutante que sobrevive señala un conjunto de pruebas que se ejecuta pero no verifica.

Dos clases de evidencia

La evidencia de que el control se ejecutó es barata y necesaria: un registro de CI, las horas de un trabajo de fuzzing, una ejecución de Miri por crate, un cambio de comprobación que se puso en rojo. Responde a la primera pregunta de un auditor. No responde a la pregunta del negocio, que es si la organización está mejor gracias a ello. Para eso hace falta evidencia de resultados, medida contra la línea base del registro de decisión del capítulo dos: defectos que llegaron a producción, caídas por pánicos, hallazgos detectados antes de publicar una versión frente a los detectados después, y el tiempo desde un hallazgo hasta su corrección. El porcentaje de cobertura es evidencia de control, no de resultados; un conjunto de pruebas puede cubrir cada línea y no verificar nada. Por esa razón mantengo ambas en columnas separadas en la ficha de contrato. Los equipos que solo informan de la primera columna terminan optimizando la actividad. Los que solo informan de la segunda no pueden explicar por qué cambiaron las cifras.

> *El compilador decide lo que puede demostrar. Para todo lo demás, el contrato solo decide si su control todavía puede decir que no, y la organización lo ha visto decir que no recientemente.*

Ficha de contrato

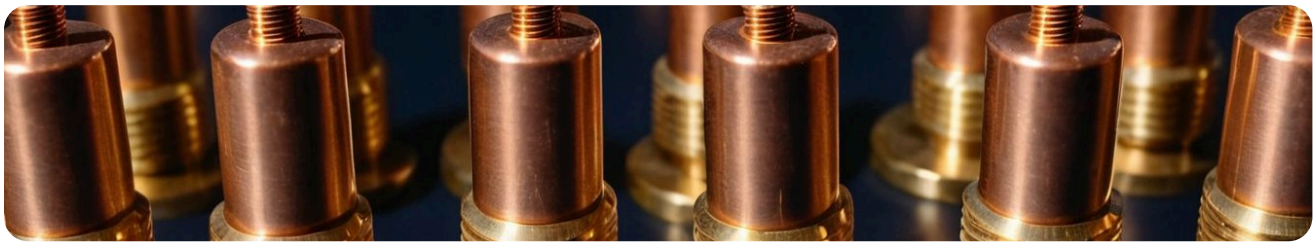
Campo	El contrato de verificación
Contrato	Cada promesa de la puerta del código tiene una verificación; se demuestra que cada control falla ante un cambio defectuoso introducido a propósito
Amenaza que aborda	Controles que dejaron de bloquear sin que nadie lo notara; pruebas que se ejecutan pero no verifican; comportamiento indefinido y caídas ante entradas inesperadas
Responsable	El equipo de plataforma es responsable del control, la política de lints y los cambios de comprobación (canarios); los responsables de cada crate lo son de sus pruebas, propiedades y objetivos de fuzzing
Punto de aplicación	Control de fusión (fmt, Clippy, pruebas en estable); trabajos programados (fuzzing, Miri sobre una nightly fijada); ejecución programada de los canarios
Comportamiento ante fallos	Una verificación de fusión que falla bloquea la fusión; un canario que pasa el control avisa al responsable del control y retiene las versiones publicadas
Autoridad de excepción y vencimiento	Solo el equipo de plataforma, para una verificación concreta y un periodo con fecha, mediante el ciclo de vida del capítulo cinco
Evidencia de que el control se ejecutó	Registros de CI con la versión de la cadena de herramientas; horas de fuzzing y corpus; registros de Miri; las ejecuciones en rojo de los canarios
Evidencia de que el resultado mejoró	Defectos que llegaron a producción, caídas por pánico y tiempo de corrección, frente a la línea base del registro de decisión
Frecuencia de revisión	Canarios con un calendario fijo; política de lints en cada actualización de la cadena de herramientas; evidencia de resultados cada trimestre

LISTA DE VERIFICACIÓN DEL CONTRATO

- Los niveles de lint están en una tabla `[lints]` del workspace, y sus cambios los revisa el equipo de plataforma.
- El control de fusión ejecuta fmt, Clippy con `-D warnings` y todas las pruebas sobre la cadena de herramientas estable fijada.
- Cada parser de entradas no confiables tiene pruebas de propiedades y un objetivo de fuzzing programado; cada entrada del registro de unsafe tiene una ejecución programada de Miri donde Miri admita el código y, si no, fuzzing, sanitizers, pruebas de modelo o revisión focalizada.
- Una tarea de comprobación (canario) programada demuestra que cada regla del control sigue fallando ante un cambio defectuoso conocido, y guarda la ejecución en rojo como evidencia.
- Los informes muestran la evidencia de control y la evidencia de resultados en columnas separadas, con el resultado medido contra una línea base.

CAPÍTULO SIETE

La fijación: dependencias y cadena de herramientas



La mayor parte del código dentro de un servicio Rust no lo escribió el equipo que lo entrega. Llega a través de un manifiesto, un lockfile lo deja fijado y se convierte en código máquina con una versión del compilador que alguien eligió, u olvidó elegir. La fijación es el contrato que establece exactamente qué código y qué compilador se usan, y quién puede cambiarlos. Sin ella, dos compilaciones del mismo commit pueden ser dos productos distintos.

La versión corta para un CTO: incluya y versione el lockfile en el repositorio, fije la cadena de herramientas (toolchain) en un archivo, publique una lista breve de crates estándar de la organización y trate un cambio en cualquiera de los tres como un cambio en el producto. Es barato. Una fijación registra lo que entró; si debía haber entrado es asunto del control de admisión del capítulo ocho.

El compilador no demuestra nada sobre versiones; verifica el código fuente que reciba, sea cual sea. Las herramientas hacen cumplir la fijación: Cargo resuelve las dependencias y fija sus versiones en el lockfile, rustup lee el archivo de la cadena de herramientas y la CI rechaza una compilación cuyo lockfile está desactualizado. Las decisiones corresponden a personas: qué crates son estándar, cuándo se actualiza la cadena de herramientas y quién puede aprobar el uso de nightly.

FIGURA 7.1

Cinco fijaciones, una compilación



Cada fijación es un archivo, así que cada cambio se revisa. Lo que las fijaciones no cubren (en cobrizo) es la razón por la que la reproducibilidad debe probarse, no suponerse.

El lockfile es el hecho

Cargo.toml declara la intención. Un requisito como `serde = "1"` significa "cualquier versión 1.x compatible". Cargo.lock declara el hecho: la versión exacta de cada crate del grafo, directo o transitivo, con un checksum para cada uno que se descarga de un registro.

El consejo de antes era versionar el lockfile en los binarios e ignorarlo en las bibliotecas. En agosto de 2023 el equipo de Cargo cambió esa recomendación: cada proyecto debe hacer lo que más le convenga, tomando como punto de partida versionar el lockfile, y `cargo new` dejó de añadir Cargo.lock a `.gitignore` en las bibliotecas (Rust Blog, 2023). Hay un detalle importante para las bibliotecas. Cargo solo usa el lockfile del workspace de nivel superior que se compila, así que el lockfile que una biblioteca versiona en su repositorio protege su propia CI, no a quienes la consumen.

EL CONTRATO DEL LOCKFILE

- Cada repositorio incluye y versiona Cargo.lock, tanto en binarios como en bibliotecas.
- La CI compila con `--locked`, de modo que un lockfile desactualizado falla en lugar de resolver versiones nuevas en silencio.
- Las actualizaciones de dependencias llegan en sus propios pull requests, para que cada una pase el control de admisión.
- Las bibliotecas también se prueban de forma programada con las últimas versiones compatibles, como aconseja el equipo de Cargo.

Workspaces: un solo lugar para cambiar una versión

Un workspace es un conjunto de crates que comparten un lockfile y un directorio de compilación. Desde Rust 1.64, un workspace puede declarar las dependencias una sola vez en `[workspace.dependencies]` y permitir que sus miembros las hereden, y puede compartir metadatos como la edición y rust-version mediante `[workspace.package]`. Una sola línea de versión, una sola revisión.

```
# Cargo.toml en la raíz del workspace (extracto; plantilla completa en el Apéndice A)
[workspace]
members = ["crates/*"]
resolver = "3"

[workspace.package]
edition = "2024"
rust-version = "1.85"

[workspace.dependencies]
tokio = { version = "1.40", features = ["rt-multi-thread", "macros", "net"] }
serde = { version = "1", features = ["derive"] }

# crates/orders-api/Cargo.toml
[package]
name = "orders-api"
version = "0.1.0"
edition.workspace = true
rust-version.workspace = true

[dependencies]
tokio.workspace = true
serde.workspace = true
```

Fije el resolver de forma explícita en la raíz. Un workspace virtual (una raíz sin paquete propio) no tiene edición de la que inferirlo. El resolver "2" es el predeterminado de la edición 2021. El resolver "3", predeterminado de la edición 2024, requiere Rust 1.84 o posterior y hace que la resolución tenga en cuenta la rust-version de cada crate (Cargo Book, resolver).

Las features se unifican, y eso es un riesgo de gobernanza

Las features de Cargo son interruptores en tiempo de compilación. Cuando varios crates de una misma compilación usan la misma dependencia, Cargo la compila una sola vez con la unión de todas las features que alguien pidió. Por eso el Cargo Book indica que las features deben ser aditivas. Lo que pide un crate cambia lo que reciben todos los demás crates de la misma compilación.

Por tanto, el conjunto de features depende de lo que se compila junto. El resolver 2 deja de unificar entre dependencias de compilación, macros procedurales, targets inactivos y dependencias de desarrollo no compiladas, pero los paquetes compilados en un mismo

comando siguen unificándose. Un binario compilado por separado con `-p` y el mismo binario compilado dentro de `cargo build --workspace` pueden llevar features distintas. Si el binario probado y el binario entregado se compilan de forma distinta, usted probó otro producto.

> **EJEMPLO: LA FEATURE QUE NADIE PIDIÓ**
>
> Imagine un equipo que activa una feature de diagnóstico detallado en un crate de registro (logging) compartido, para una herramienta interna. Las compilaciones de release se ejecutan sobre todo el workspace, así que la feature llega sin que nadie lo note también al servicio público, y ningún diff del servicio lo reflejó. La solución: compilar los artefactos de release paquete a paquete, de la misma forma en CI y en la publicación; `cargo tree -e features -i <crate>` muestra quién activó una feature.

Un presupuesto de dependencias y crates estándar de la organización

Cada dependencia directa trae sus propias dependencias, mantenedores, código unsafe y scripts de compilación. Un presupuesto hace visible ese costo: el equipo de plataforma fija un tope de dependencias directas por servicio, y cada nueva necesita una justificación de una línea en el pull request, incluido por qué no basta un crate estándar existente. `cargo tree --duplicates` muestra los crates compilados en más de una versión.

La segunda mitad del presupuesto es una lista de crates estándar de la organización: un crate elegido para cada función, bajo la responsabilidad del equipo de plataforma. Las funciones típicas son el entorno de ejecución asíncrono, el cliente y el servidor HTTP, la serialización, TLS, el manejo de errores y el registro de trazas (tracing). El entorno de ejecución asíncrono es lo que más importa, porque no es una biblioteca a la que se llama, sino un planificador dentro del cual se ejecuta su código. Dos entornos de ejecución en un servicio implican dos planificadores y confusión sobre a cuál corresponde cada tarea. Tokio es la opción habitual para servicios de red, y crates muy usados como hyper, axum y tonic se basan en él. smol es una alternativa más ligera. Sus mantenedores abandonaron async-std en 2025 y recomiendan smol (RustSec RUSTSEC-2025-0052, 2025). Para absorber ese tipo de cambio existe la lista: un único responsable actúa una sola vez.

El archivo de la cadena de herramientas y la regla de nightly

Un archivo `rust-toolchain.toml` en la raíz del repositorio indica a rustup qué compilador usar. Solo lo anula una elección explícita en la línea de comandos, la variable `RUSTUP_TOOLCHAIN` o una anulación de directorio (rustup book, overrides).

```
# rust-toolchain.toml (versión fijada de ejemplo; actualícela según el calendario del entorno)
[toolchain]
channel = "1.100.0"
components = ["rustfmt", "clippy"]
targets = ["x86_64-unknown-linux-musl", "aarch64-unknown-linux-gnu"]
profile = "minimal"
```

Rust estable publica una versión cada seis semanas. Fije una versión estable exacta y actualícela con una cadencia definida, mediante un pull request ordinario que pase por el control completo. Por tanto, la versión fijada por la organización puede ir hasta un ciclo por detrás de la versión estable más reciente; eso es deliberado, y una corrección de seguridad en el compilador o en la biblioteca estándar es motivo para usar la vía de actualización de emergencia en lugar de esperar. Actualice el lockfile y la cadena de herramientas en pull requests separados, para que un fallo apunte a una sola causa.

Las features inestables necesitan un compilador nightly, y nightly cambia a diario. Mi regla: nightly solo con un responsable designado, una razón por escrito, una fijación con fecha como `nightly-2026-11-01` y una condición de salida, normalmente "cuando la feature llegue a estable". La variable de entorno `RUSTC_BOOTSTRAP` permite que un compilador estable acepte features inestables; la CI debe rechazar las compilaciones que la definan fuera de la excepción aprobada. Es el ciclo de vida de excepciones del capítulo cinco, aplicado al compilador.

MSRV y ediciones

La versión mínima de Rust admitida (MSRV) es una promesa a los consumidores. Declárela en `rust-version` y Cargo rechazará compiladores más antiguos. Desde Rust 1.84 el resolver puede preferir versiones de dependencias cuya propia rust-version sea compatible con la suya; el resolver "3" lo convierte en el comportamiento predeterminado (Cargo Book, resolver). Solo un trabajo de CI con ese compilador exacto demuestra la MSRV: `cargo +1.85.0 check --workspace --locked`. Subirla puede ser un cambio incompatible según la guía SemVer de Cargo, así que el capítulo nueve la incluye en la promesa de interfaz.

Las ediciones son revisiones opcionales del lenguaje elegidas por crate, y los crates de ediciones distintas se enlazan entre sí, así que la migración con `cargo fix --edition` puede hacerse crate a crate. La edición 2024 es estable desde Rust 1.85. Los crates nuevos empiezan en la edición actual; los existentes migran según un plan con fechas.

Los perfiles de release son un contrato sobre el fallo

Algunos ajustes de release cambian el comportamiento, no solo el tamaño y la velocidad.

```
[profile.release]
lto = true            # optimización del programa completo ("fat"); compilaciones más lentas
codegen-units = 1     # mejor optimización; compilación menos paralela
strip = true          # elimina los símbolos del binario entregado
panic = "abort"       # sin desenrollado; el proceso termina ante un panic
overflow-checks = true  # explícito, no por defecto: panic en lugar de aritmética modular
```

`panic = "abort"` significa que un pánico termina el proceso de inmediato: sin unwinding, sin destructores, y `catch_unwind` no puede capturarlo. Las pruebas ignoran este ajuste, porque el arnés de pruebas necesita unwinding (Cargo Book, profiles). Por defecto, en las compilaciones de release un desbordamiento de enteros se produce sin aviso y continúa con aritmética modular (wrapping); overflow-checks convierte ese desbordamiento silencioso en un pánico. Los valores predeterminados también han cambiado: desde Rust 1.77, las compilaciones de release sin información de depuración eliminan la debuginfo por defecto, salvo en los targets Windows MSVC, donde Rust 1.77.1 desactivó el cambio (Rust Blog, 2024). Fije strip de forma explícita para que el contrato no dependa de un valor predeterminado. Para ver qué efecto tienen estos ajustes en el tamaño y la velocidad, consulte el servicio de referencia del Apéndice A.

Los límites de la reproducibilidad

Fijar es necesario, pero no suficiente. El lockfile fija el código fuente de Rust mediante checksums. No fija las bibliotecas de C del sistema, los scripts de compilación que leen el entorno o descargan archivos, ni las rutas absolutas incrustadas en los mensajes de pánico y en la información de depuración, salvo que se reasignen con `--remap-path-prefix`. Una opción de perfil para recortar rutas sigue siendo inestable en el momento de escribir esto. No afirme que tiene compilaciones reproducibles bit a bit hasta haber compilado el mismo commit dos veces, en dos máquinas, en dos directorios distintos, y comparado los hashes. Si difieren, la afirmación honesta es "entradas fijadas".

> *El desarrollador propone una versión. El lockfile la registra. El control del siguiente capítulo decide si puede entrar.*

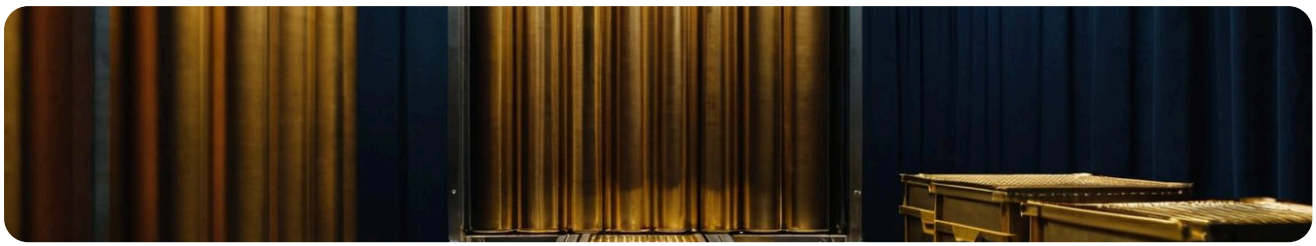
Ficha de contrato	La fijación
Contrato	Cada compilación usa un lockfile incluido y versionado en el repositorio, una cadena de herramientas fijada, un perfil de release explícito y, para las funciones estándar, solo crates estándar de la organización.
Amenaza que aborda	Deriva silenciosa: versiones, compiladores o features que llegan a producción sin revisión.
Responsable	El equipo de plataforma es responsable de la versión fijada de la cadena de herramientas, de la lista de crates estándar y de la plantilla de perfiles. Los equipos de producto son responsables de sus lockfiles.
Punto de aplicación	CI: compilaciones con `--locked`, presencia del archivo de la cadena de herramientas, trabajo de MSRV, rechazo de `RUSTC_BOOTSTRAP`, verificación del presupuesto ante nuevas dependencias directas.
Comportamiento ante fallos	Bloquea por defecto ante un fallo: un lockfile desactualizado, la ausencia del archivo de la cadena de herramientas o una nightly no aprobada bloquean la fusión.
Autoridad de excepción y vencimiento	El responsable de plataforma, para nightly y crates fuera de la lista, con un responsable designado, una razón y un vencimiento de un trimestre como máximo (política de ejemplo para la organización).
Evidencia de que el control se ejecutó	Registros de CI con `--locked` y la versión de la cadena de herramientas de cada compilación; el registro de excepciones de nightly.
Evidencia de que el resultado mejoró	Menos versiones duplicadas de crates con el tiempo; ningún incidente de producción atribuido a un cambio de versión sin revisar.
Frecuencia de revisión	Actualización de la cadena de herramientas en cada ciclo de la cadencia definida; lista de crates estándar cada trimestre; excepciones de nightly cada mes.

LISTA DE VERIFICACIÓN DEL CONTRATO

- Cargo.lock está incluido y versionado en cada repositorio, y la CI compila con `--locked`.
- Un rust-toolchain.toml fija una versión estable exacta, y el uso de nightly tiene responsable, razón, fijación con fecha y condición de salida.
- El workspace fija su resolver de forma explícita y hereda las versiones de las dependencias desde una sola tabla.
- Los artefactos de release se compilan paquete a paquete, de la misma forma en CI y en la publicación, y los conjuntos de features pueden inspeccionarse.
- El campo rust-version está declarado y lo demuestra un trabajo de CI con ese compilador exacto.
- El perfil de release fija de forma explícita el comportamiento ante pánicos, el strip y el desbordamiento; las afirmaciones de reproducibilidad se prueban antes de hacerlas.

CAPÍTULO OCHO

El control de admisión



Un lockfile registra lo que entró en la compilación. Nunca pregunta si debía haber entrado. Cada crate que entra es código que se ejecutará en producción y, a través de scripts de compilación y macros procedurales, parte de él se ejecuta en la máquina de compilación antes de que producción llegue a verlo. El control de admisión es el contrato que responde a la pregunta en la puerta: ¿puede entrar este código, con qué reglas y con la aprobación de quién?

Para un CTO, el control son cinco preguntas que se hacen automáticamente en cada cambio y de nuevo cada día: ¿la licencia es aceptable?, ¿la versión tiene una vulnerabilidad conocida?, ¿el crate está prohibido o duplicado?, ¿viene de un origen aprobado? y ¿lo ha revisado alguien en quien la organización confía? Cualquier respuesta que muestre un incumplimiento bloquea la fusión. Una excepción autorizada tiene un responsable y una fecha de vencimiento. Y el control genera evidencia al final del proceso: una lista de materiales, una firma y un registro de cómo se construyó el artefacto.

El compilador no demuestra casi nada en esta puerta. El Rust seguro dentro de un crate es seguro en memoria, pero el crate puede contener código unsafe, llamar a C o ejecutar código arbitrario en tiempo de compilación (capítulo cinco). Las herramientas hacen cumplir las reglas. Las reglas en sí corresponden a personas: el área jurídica es responsable de la lista de licencias; el equipo de seguridad, de los avisos de seguridad y de las excepciones; y el equipo de plataforma, de los orígenes, las prohibiciones y el buen funcionamiento del propio control.

FIGURA 8.1

El control de admisión, sus salidas y su evidencia



El control se ejecuta dos veces. Una cuando cambia una dependencia, y otra cada día, porque los avisos de seguridad se publican después de que el código se ha admitido. Un bloqueo solo se levanta con una excepción con vencimiento, nunca dejando pasar el cambio sin constancia escrita.

Las cinco preguntas como reglas escritas

Pregunta	Regla	Responsable	Implementación habitual
Licencia	Solo licencias de la lista permitida; lo desconocido significa no	Legal	cargo-deny licenses
Avisos	Ninguna vulnerabilidad conocida; se señalan los crates sin mantenimiento y los que tienen fallos de solidez (unsound)	Seguridad	cargo-deny advisories, cargo-audit
Prohibiciones	Ningún crate prohibido, ningún requisito comodín, duplicados justificados	Plataforma	cargo-deny bans
Orígenes	Solo registros aprobados; orígenes git solo por excepción	Plataforma	cargo-deny sources
Revisión de dependencias	Cada versión revisada internamente o por un auditor en quien la organización confía	Seguridad junto con los equipos de producto	cargo-vet

cargo-deny cubre cuatro de las cinco preguntas desde un solo archivo, `deny.toml`, y termina con error cuando falla una verificación, que es todo lo que necesita un control de CI. Sigue un extracto, probado con cargo-deny 0.20.2, versión que rechaza claves y valores desconocidos al cargar la configuración; las versiones anteriores usaban `name` y `version` donde esta usa `crate`:

```
# deny.toml para cargo-deny 0.20.2 (extracto; plantilla completa en el Apéndice A)
[advisories]
yanked = "deny"
unmaintained = "all"
ignore = [
  { id = "RUSTSEC-0000-0000", reason = "not reachable; waiver W-0000, expires 2027-03-31" },
]

[licenses]
allow = ["MIT", "Apache-2.0", "BSD-3-Clause", "ISC", "Unicode-3.0"]
confidence-threshold = 0.9

[bans]
multiple-versions = "deny"   # el valor por defecto es "warn"; use deny, y skip con un motivo
wildcards = "deny"
deny = [{ crate = "openssl", use-instead = "rustls" }]
skip = [{ crate = "syn:<3", reason = "older major via a proc macro; remove when upstream moves"
}]

[sources]
unknown-registry = "deny"
unknown-git = "deny"
```

Licencias y avisos de seguridad

Una licencia es un compromiso jurídico, así que la lista permitida la redacta el área jurídica y se expresa en identificadores SPDX. Un crate cuya licencia no puede detectarse con suficiente certeza no pasa el control. Las excepciones se conceden crate a crate, con una razón, y nunca amplían la lista.

Los avisos de seguridad provienen de la RustSec Advisory Database, una base de datos comunitaria de avisos de seguridad para crates publicados en crates.io. cargo-audit y la verificación advisories de cargo-deny la leen. Las versiones actuales de cargo-deny tratan cada aviso de vulnerabilidad como un error; la configuración decide el alcance de los avisos sobre crates sin mantenimiento o con fallos de solidez (unsound), y si las versiones retiradas (yanked) hacen fallar el control (documentación de cargo-deny). Ejecute la verificación en cada pull request, y también a diario sobre la rama principal y sobre el lockfile de cada versión publicada que siga desplegada. Un aviso publicado hoy afecta al binario que usted entregó el mes pasado.

Prohibiciones, duplicados y orígenes

Las prohibiciones codifican decisiones de la organización: una sola pila TLS, un solo entorno de ejecución, ningún crate que el equipo de seguridad haya rechazado. El campo `use-instead` convierte una prohibición en una recomendación concreta. Las versiones duplicadas de un mismo crate implican binarios más grandes, compilaciones más largas y más código que revisar. El objetivo es denegar los duplicados; una entrada `skip` con una razón es la forma honesta de convivir con uno mientras el proyecto de origen (upstream) se actualiza. Los requisitos de versión comodín se prohíben por completo, porque hacen que el manifiesto no prometa nada.

Las reglas de orígenes establecen de dónde puede venir el código. La regla que recomiendo es crates.io, o el registro privado o la réplica del registro (mirror) de la organización, y nada más por defecto. Una dependencia git es código que eludió todas las verificaciones del registro, así que necesita una excepción con un responsable, un commit fijado y un plan de salida.

Revisión de dependencias: alguien de confianza ha leído el código

Las verificaciones anteriores se ocupan de los metadatos. La revisión de dependencias se ocupa del código. cargo-vet registra, para cada versión de un crate, que una persona la ha auditado con arreglo a criterios declarados. Hay dos criterios integrados: `safe-to-run`, que significa que el crate puede compilarse, ejecutarse y probarse en una estación de trabajo o en una automatización controlada sin consecuencias sorprendentes, y `safe-to-deploy`, que significa que no introducirá una vulnerabilidad de seguridad grave en software de producción expuesto a entradas no confiables; el segundo implica el primero (cargo-vet book). Las auditorías se guardan en el repositorio, y auditar el diff entre dos versiones permite a un revisor leer solo lo que cambió.

Ninguna organización puede auditarlo todo por su cuenta, así que cargo-vet puede importar los conjuntos de auditorías que publican otras organizaciones. Su registro incluye, entre otras, a Mozilla, Google y la Bytecode Alliance. Importar una auditoría es una decisión de confianza, no un mero detalle técnico: el equipo de plataforma decide de quién son las auditorías que cuentan y deja constancia de esa decisión en la configuración.

```
# supply-chain/config.toml (extracto)
[imports.mozilla]
url = "https://raw.githubusercontent.com/mozilla/supply-chain/main/audits.toml"

[imports.google]
url = "https://raw.githubusercontent.com/google/supply-chain/main/audits.toml"
```

Al adoptar la herramienta, todas las dependencias existentes pasan a una lista de exenciones. Esa lista es la deuda de revisión de dependencias, puesta por escrito. El contrato es que solo puede reducirse: los crates nuevos necesitan una auditoría propia o importada, y el número de exenciones se informa cada mes.

Una sola puerta al exterior: réplica del registro o dependencias en el repositorio

Un control es más fácil de mantener cuando hay una sola puerta. Hay dos patrones que funcionan. Una réplica privada del registro (mirror) hace que las compilaciones de la organización descarguen solo desde un origen que controla el equipo de plataforma. La incorporación de las dependencias al repositorio (vendoring), con `cargo vendor`, copia el código fuente de cada dependencia en el repositorio e imprime la configuración de reemplazo de orígenes (source replacement) que indica a Cargo que lo use. Ambos patrones resisten una caída del registro y la retirada de un crate en su origen. El vendoring, además, hace visibles en el diff los cambios de terceros, a costa del tamaño del repositorio. En ambos casos, el control se ejecuta antes de que el código llegue a la réplica o al directorio de dependencias incorporadas, no después.

Evidencia a la salida: SBOM, firmas, procedencia

Un SBOM (lista de materiales de software) enumera lo que contiene un artefacto. cargo-cyclonedx genera un SBOM CycloneDX a partir del lockfile y de los metadatos de Cargo. Refleja las features y el destino (target) de la compilación entregada solo cuando se le pasan los mismos argumentos de features y de destino que a esa compilación. cargo-auditable sigue otro enfoque: incrusta una lista comprimida de las dependencias del binario en una sección del propio binario, de modo que cargo-audit, Trivy, Grype y Syft pueden analizar un binario entregado sin su código fuente (documentación de cargo-auditable). El proyecto deja claro que esto aporta visibilidad, no protección contra ataques a la cadena de suministro. Use ambos: el SBOM acompaña a la versión publicada, y la lista incrustada no puede separarse del binario.

Una firma indica quién publicó el artefacto. La procedencia indica cómo se construyó. cosign, de Sigstore, firma imágenes de contenedor y adjunta atestaciones como un SBOM, y su modo sin claves vincula la firma a una identidad de carga de trabajo emitida por un proveedor OIDC y la anota en un registro público de transparencia. SLSA describe la procedencia por niveles: en Build L1 la procedencia existe, en L2 está firmada y la genera una plataforma de compilación alojada, y en L3 esa plataforma está reforzada para que los pasos de compilación no puedan acceder al secreto de firma (SLSA v1.0, 2023). Un despliegue que no verifica nada de esto ha recibido la evidencia y la ha ignorado; los capítulos diez y once cierran ese ciclo.

```
# El mismo destino y las mismas features que la compilación entregada
cargo cyclonedx --format json --target x86_64-unknown-linux-gnu --features "$FEATURES"
cosign attest --type cyclonedx --predicate "$SBOM" "$IMAGE@$DIGEST"
cosign verify-attestation --type cyclonedx \
  --certificate-identity "$CI_IDENTITY" \
  --certificate-oidc-issuer "$CI_ISSUER" "$IMAGE@$DIGEST"

# Procedencia de la compilación, GitHub Actions. El trabajo necesita:
#   permissions: { id-token: write, attestations: write, packages: write }
#   (packages: write solo porque push-to-registry es true)
- uses: actions/attest@v4            # sin predicado ni SBOM de entrada: procedencia de
compilación SLSA
  with:
    subject-name: registry.example.com/service
    subject-digest: ${{ steps.build.outputs.digest }}
    push-to-registry: true
# Al desplegar: la procedencia debe venir del repositorio y el workflow esperados
gh attestation verify "oci://$IMAGE@$DIGEST"   --repo org/service   --signer-workflow
org/service/.github/workflows/release.yml
```

El paso de procedencia sigue la documentación de artifact attestations de GitHub (2026); sin entrada de SBOM ni de predicado, `actions/attest` produce una atestación de procedencia de compilación SLSA. `actions/attest-build-provenance@v4` es ahora un envoltorio sobre ella, así que los flujos de trabajo (workflows) existentes que lo usan siguen funcionando, y GitHub recomienda `actions/attest` para los nuevos. Las opciones de verificación son las que enumera `gh attestation verify --help` (gh 2.81): `--repo` u `--owner` acotan la búsqueda, y `--signer-workflow` fija el flujo de trabajo que firmó. Una nota sobre la entrada `skip` anterior: en cargo-deny una especificación de crate admite `name@version` para una versión exacta y `name:requirement` para un requisito de versión, así que `syn:<3` significa "cualquier syn por debajo de 3" (comprobado con 0.20.2). En otros entornos, un generador de procedencia SLSA verificado con `slsa-verifier`, o el propio servicio de atestaciones de la plataforma de CI, cumple el mismo papel.

Comportamiento ante fallos y la vía de excepciones

El control falla en modo cerrado (fail closed): bloquea por defecto ante un fallo. Una verificación que falla bloquea la fusión, y un control que no puede acceder a su base de datos de avisos falla en lugar de aprobar con datos obsoletos. La gravedad cambia la respuesta, no la regla: un nuevo aviso crítico que afecta a una versión publicada y desplegada abre un incidente, mientras que un aviso de falta de mantenimiento abre un ticket con fecha límite.

Las excepciones son el punto donde suelen perderse los controles. La autoridad se reparte por pregunta: el área jurídica concede las excepciones de licencias; el equipo de seguridad, las de avisos y de revisión de dependencias; y el responsable de plataforma, las de prohibiciones y orígenes. Cada excepción lleva un responsable, una razón, un control compensatorio y un vencimiento. Como política de ejemplo para la organización, uso un

tope de un trimestre, renovable una vez con una nueva aprobación; fije el suyo y póngalo por escrito. En cargo-deny 0.20.2 una entrada ignore de avisos admite un `id` y una `reason` (una versión retirada puede nombrarse en su lugar con una especificación de crate) y no tiene campo de vencimiento, así que el vencimiento se guarda en el registro de excepciones, no en `deny.toml`. Anote la excepción en ese registro, haga referencia a su identificador en la entrada `ignore` o `skip`, como en el extracto anterior, y haga que un trabajo de CI falle cuando una entrada del registro supere su fecha.

Demostrar que el control bloquea

Un control que nunca ha bloqueado nada es un control que nadie ha probado. Mantenga un proyecto de prueba defectuoso por regla, cada uno preparado para incumplir exactamente una cosa: un crate prohibido, una licencia no permitida, un origen git, una versión con un aviso conocido. Mantenga también un proyecto de prueba limpio, como caso de control, que debe pasar. Un trabajo programado ejecuta el control sobre cada uno. Una simple prueba de "¿falló?" no basta, porque una herramienta ausente, una configuración errónea o una descarga fallida de la base de datos también producen un fallo; el trabajo debe comprobar que cada proyecto defectuoso falla por su propia razón, comparando el código exacto (campo `code`) de un diagnóstico de error en la salida JSON de cargo-deny. Es la regla del capítulo seis: un control debe probarse para demostrar que bloquea.

```
# gate-canary.sh: el caso de control debe pasar; cada proyecto de prueba defectuoso debe fallar
por su propio motivo
set -u
run() { cargo deny --locked --format json --manifest-path "fixtures/$1/Cargo.toml" check $2 2>&1;
}
errors() {  # una línea por error: su código de diagnóstico, o tool-error si falla la
configuración, la descarga o la herramienta
  python3 -c '
import json, sys
for line in sys.stdin:
    try: d = json.loads(line)
    except ValueError: continue
    f = d.get("fields", {})
    if d.get("type") == "log" and f.get("level") == "ERROR": print("tool-error")
    elif d.get("type") == "diagnostic" and f.get("severity") == "error": print(f.get("code"))'
}
run clean "" >/dev/null || { echo "GATE BROKEN: clean control failed" >&2; exit 1; }
check_blocks() {  # proyecto de prueba, comprobación, código esperado
  out=$(run "$1" "$2") && { echo "GATE BROKEN: $1 was admitted" >&2; exit 1; }
  codes=$(printf '%s\n' "$out" | errors)
  printf '%s\n' "$codes" | grep -qx tool-error && {
    echo "GATE BROKEN: $1: tool, config or fetch failure, not a verdict" >&2; exit 1; }
  printf '%s\n' "$codes" | grep -qx "$3" || {
    echo "GATE BROKEN: $1 failed without code $3" >&2; exit 1; }
}
check_blocks banned-crate bans       banned
check_blocks bad-license  licenses   rejected
check_blocks git-source   sources    source-not-allowed
check_blocks vulnerable   advisories vulnerability
echo "gate blocked every fixture for the expected reason"
```

Los códigos son los que emitió cargo-deny 0.20.2, como líneas JSON en la salida de error estándar, para proyectos de prueba construidos de esta forma: `banned`, `rejected`, `source-not-allowed` y `vulnerability`. En cambio, un error de configuración, una base de datos de avisos que no puede descargarse o un manifiesto ausente aparecen como una línea de registro de nivel error, y el script lo notifica como un fallo de la herramienta, no como un bloqueo; el código de salida por sí solo no permite distinguir ambos casos. El script se ejecutó sobre los cuatro proyectos defectuosos y el proyecto limpio de control, y también sobre una configuración rota, una prohibición ausente y una base de datos inaccesible, y notificó cada caso correctamente. Confirme los códigos cuando actualice la versión fijada de cargo-deny. Mantenga los proyectos de prueba dentro del repositorio: cargo-deny 0.20.2 busca `deny.toml` desde el directorio del proyecto de prueba hacia arriba, así que cada uno se evalúa con el archivo de la propia organización. Si la raíz del repositorio es un workspace de Cargo, dé a cada proyecto de prueba una tabla `[workspace]` vacía para que Cargo lo trate como su propia raíz.

El desarrollador propone una dependencia. El control decide, por escrito, cada día, y una persona con autoridad firma cada excepción.

Ficha de contrato	El control de admisión
Contrato	Ningún crate entra en la compilación salvo que su licencia, avisos, estado de prohibición, origen y estado de auditoría cumplan reglas escritas; cada versión publicada lleva un SBOM, una firma y procedencia.
Amenaza que aborda	Código de terceros vulnerable, malicioso, sin licencia o sin revisar que llega a producción; artefactos de origen desconocido.
Responsable	El equipo de plataforma es responsable del control. El área jurídica es responsable de la lista de licencias. El equipo de seguridad es responsable de los avisos y de la revisión de dependencias.
Punto de aplicación	CI de pull requests, una ejecución programada diaria sobre main y sobre los lockfiles desplegados, y la verificación de firmas en el despliegue.
Comportamiento ante fallos	Bloquea por defecto ante un fallo, incluso cuando no puede descargarse la base de datos de avisos. Los avisos críticos que afectan a versiones publicadas y desplegadas abren un incidente.
Autoridad de excepción y vencimiento	El área jurídica para licencias, el equipo de seguridad para avisos y revisión de dependencias, el responsable de plataforma para prohibiciones y orígenes; política de ejemplo para la organización: un trimestre como máximo, renovable una vez.
Evidencia de que el control se ejecutó	Salida del control guardada por compilación; historial de ejecuciones diarias; la tarea de comprobación (canario) que confirma que el caso de control limpio pasa y que cada proyecto defectuoso falla por la razón esperada; SBOM y atestación por versión publicada.
Evidencia de que el resultado mejoró	Exenciones de la revisión de dependencias que se reducen mes a mes; menor tiempo entre la publicación de un aviso y la versión corregida.
Frecuencia de revisión	Registro de excepciones cada mes; listas permitidas, prohibiciones y orígenes de auditorías importadas cada trimestre.

LISTA DE VERIFICACIÓN DEL CONTRATO

- Un solo archivo de configuración codifica las reglas de licencias, avisos, prohibiciones y orígenes, y la CI falla ante cualquier error.
- Las verificaciones de avisos se ejecutan en cada pull request y a diario sobre main y sobre cada versión publicada que esté desplegada.
- La revisión de dependencias queda registrada, los orígenes de auditorías importadas son una decisión de confianza por escrito, y la lista de exenciones solo se reduce.
- Cada versión publicada tiene un SBOM, una lista de dependencias incrustada, una firma y procedencia, y los despliegues los verifican.
- Cada excepción tiene un responsable, una razón, un control compensatorio y un vencimiento, y las excepciones vencidas hacen fallar la compilación.
- Una tarea de comprobación (canario) demuestra, de forma programada, que el caso de control limpio pasa y que cada proyecto de prueba defectuoso falla con el diagnóstico esperado.

CAPÍTULO NUEVE

La promesa de la interfaz



Dentro de una compilación, el compilador verifica cada llamada contra cada firma. Más allá de la frontera de una versión publicada, no puede hacerlo. Cuando otro equipo depende de su crate, o un cliente TypeScript depende de su servicio, lo único que cruza es una promesa: esta versión no romperá su código. El número de versión es la afirmación. La promesa de la interfaz es el contrato que hace que esa afirmación sea cierta, la verifica antes de publicar y da margen de tiempo al código que llama cuando la interfaz debe cambiar.

Para un CTO: cada interfaz pública que expone el entorno tecnológico, sea una biblioteca, una API o un cliente generado, tiene un responsable, una versión que sigue reglas escritas, una verificación automatizada de que la versión dice la verdad, una revisión que lee el diff de la interfaz y no solo el diff del código, y un plazo de deprecación. Los cambios incompatibles están permitidos. Los cambios incompatibles inesperados, no. Aquí es donde "Cada línea se sostiene" se pone a prueba frente al cambio, y la versión honesta del lema es más acotada: cada línea sostiene la promesa que hizo su número de versión, hasta donde las verificaciones alcanzan a ver.

De nuevo, los niveles. El compilador demuestra que el código compila contra la interfaz tal como es hoy. Las herramientas hacen cumplir la comparación con lo que se prometió antes: verificadores de semver, diffs de esquemas, bindings regenerados. Las personas son responsables de decidir qué es público en primer lugar, qué cuenta como cambio incompatible en el comportamiento y cuándo una versión mayor vale lo que cuesta.

Qué es la API pública

La API pública de un crate de Rust es más grande que su lista de `pub fn`. Incluye los tipos públicos y sus campos, las variantes de los enums, las definiciones e implementaciones de traits, los auto traits que un tipo tenga de forma implícita (un tipo que deja de ser `Send` rompe el código de todo aquel que lo haya movido entre hilos), las features de Cargo del crate y su versión mínima de Rust admitida. El comportamiento también forma parte, y

ninguna herramienta ve todo el comportamiento. La gobernanza más barata es una superficie más pequeña: usar `pub(crate)` por defecto y que cada nuevo elemento público sea una decisión explícita.

SemVer tal como lo lee Cargo

Cargo trata un requisito simple como `"1.4"` como un requisito caret: cualquier versión desde 1.4.0 hasta 2.0.0, sin incluirla. Por debajo de 1.0 la regla cambia. Cargo trata solo el número distinto de cero más a la izquierda como la frontera de compatibilidad, de modo que en 0.y.z un cambio incompatible incrementa y, una adición compatible incrementa z, y en 0.0.z todo cambio se trata como incompatible (Cargo Book, SemVer compatibility). Fíjese en el vocabulario: en 0.y.z la versión compatible es un incremento de z, que Cargo acepta como compatible, aunque SemVer en sí llame a z el número de parche. Muchos crates muy usados permanecen en 0.x durante años, lo cual está bien, pero el entorno debe saber que pasar de 0.4 a 0.5 es, por definición, una actualización incompatible. Para las bibliotecas internas, mi regla es publicar la 1.0 en cuanto un segundo equipo dependa del crate. Así el número significa lo que los lectores esperan.

FIGURA 9.1

Qué número se mueve



Cargo en 0.y.z: subir y rompe, subir z es compatible. En 0.0.z, todo incremento rompe.

La primera pregunta decide la mayoría de los casos. "Rompe a quien llama" incluye el comportamiento, que ninguna herramienta verifica por completo. En caso de duda, el cambio es mayor, o se publica con un nombre nuevo y el anterior se depreca.

Aditivo e incompatible, según el manual

El capítulo de compatibilidad SemVer del Cargo Book es la referencia que el entorno debe adoptar en lugar de reescribirla. Unas pocas de sus reglas deciden la mayoría de las revisiones:

Cambio	Guía de Cargo
Añadir nuevos elementos públicos	Menor (las importaciones glob rara vez pueden entrar en conflicto)
Añadir una nueva feature de Cargo	Menor
Añadir una variante a un enum que no es non_exhaustive	Mayor
Añadir un campo público cuando no existe ningún campo privado	Mayor
Añadir un elemento de trait sin implementación por defecto	Mayor
Eliminar una feature de Cargo	Mayor
Añadir non_exhaustive a un enum, variante o struct existente sin campos privados	Mayor
Subir la versión mínima de Rust admitida	Posiblemente incompatible; la guía sugiere tratarlo como menor

La última fila es una decisión de política, así que conviene dejarla por escrito. Mi recomendación: subir la MSRV es una versión menor con nota de versión, nunca un parche, y la MSRV se mantiene igual o por detrás de la cadena de herramientas (toolchain) que el entorno fija según el capítulo siete.

non_exhaustive: espacio para crecer, a un precio

El atributo `#[non_exhaustive]` indica a otros crates que pueden llegar más variantes o campos. El código que lo usa desde fuera del crate debe añadir un brazo comodín al hacer match, y no puede construir el struct con un literal. Añadir una variante más adelante pasa a ser un cambio menor.

```
#[non_exhaustive]
#[derive(Debug, Clone, Copy, PartialEq, Eq)]
pub enum OrderStatus {
    Pending,
    Paid,
    Shipped,
}

// En un crate que lo usa, el brazo comodín es obligatorio:
match status {
    OrderStatus::Pending => hold(order),
    OrderStatus::Paid | OrderStatus::Shipped => release(order),
    _ => escalate(order), // las variantes futuras caen aquí, por contrato
}
```

Observe la contrapartida. El capítulo cuatro elogió el match exhaustivo: se añade una variante y el compilador encuentra cada lugar que debe manejarla. `non_exhaustive` renuncia a esa verificación en la frontera a cambio de libertad para evolucionar. Dentro del crate que lo define, el atributo no tiene efecto. Elija de forma deliberada: en una biblioteca publicada cuyos usuarios se desconocen, marque los enums que van a crecer. En crates de un mismo workspace, donde todo el código que llama cambia en el mismo pull request, déjelos exhaustivos y deje que el compilador haga el trabajo. Decida pronto, porque añadir el atributo más tarde es en sí mismo un cambio incompatible.

Verificar la afirmación en CI

cargo-semver-checks compara el crate actual contra una línea base, por defecto la versión anterior publicada en crates.io, o una revisión de git, e informa de los cambios que rompen SemVer. Devuelve un código de salida cuando hay incumplimientos y otro distinto cuando no pudo ejecutarse, de modo que CI puede distinguir una promesa rota de una verificación rota (documentación de cargo-semver-checks). Todavía no lo detecta todo. Sus mantenedores reconocen lagunas como los cambios en el tipo de un campo o de un parámetro, algunos cambios de genéricos y de tiempos de vida (lifetimes), y las rupturas que solo aparecen con ciertas features. Los registros personalizados no se admiten como línea base, así que los crates privados se comparan contra una revisión de git. Elija la línea base según el propósito. El control de publicación compara contra la última versión publicada, porque eso es lo que tienen los consumidores: `--baseline-version` para un crate en crates.io, o `--baseline-rev` con la etiqueta de la última versión para un crate privado, seleccionada por crate mediante un patrón de etiqueta y excluyendo las versiones preliminares; una primera versión no tiene línea base, así que CI lo indica y la omite (el trabajo de CI completo está en el Apéndice A). Una comparación contra `main` es una retroalimentación útil en un pull request, pero no es el control de publicación, ya que `main` puede contener ya rupturas no publicadas.

```
# Control de release: comparar con la última etiqueta de release estable de este crate (crate
privado)
TAG=$(git describe --tags --abbrev=0 --match 'orders-client-v*' --exclude 'orders-client-v*-*'
HEAD^)
cargo semver-checks --package orders-client --baseline-rev "$TAG"   # p. ej. orders-client-v1.4.2
# o, para un crate publicado en crates.io, la última versión publicada
cargo semver-checks --package orders-client --baseline-version 1.4.2

# Solo como información en el pull request: comparar con la rama de destino
cargo semver-checks --package orders-client --baseline-rev origin/main
```

Una herramienta que detecta la mayoría de las rupturas sigue necesitando a una persona. Haga que el diff de la interfaz forme parte de la revisión: un listado de la API pública antes y después, producido por una herramienta como cargo-public-api (necesita tener instalada la cadena de herramientas nightly para generar el JSON de rustdoc, así que se aplica la regla sobre nightly del capítulo siete). Un pull request etiquetado como incompatible necesita la aprobación del responsable de la interfaz, una versión mayor y una nota de migración en el changelog.

La deprecación tiene plazos

Eliminar algo es un cambio mayor. Anunciarlo no debería serlo. El atributo `#[deprecated]` hace que la compilación de todo el código que lo llama emita una advertencia, con una versión y una nota.

```
#[deprecated(since = "2.3.0", note = "use `submit_order`; removal planned for 3.0")]
pub fn submit(order: &Order) -> Result<Receipt, SubmitError> {
    submit_order(order, SubmitOptions::default())
}
```

EL CONTRATO DE DEPRECACIÓN

- Deprecar en una versión menor, con el reemplazo ya disponible.
- Conservar el elemento deprecado durante un mínimo declarado, por ejemplo dos versiones menores o un trimestre, lo que sea más largo.
- Eliminar solo en una versión mayor, con una nota de migración.
- Registrar cada deprecación, su responsable y su fecha de eliminación en el registro de interfaces, y avisar directamente a los consumidores conocidos.

Deriva del contrato entre lenguajes

Las promesas más difíciles cruzan lenguajes. Un servicio en Rust atiende a un front end en TypeScript, una integración en C# y un proceso batch en Java. Los tipos de Rust cambian, el compilador queda satisfecho y los clientes fallan en tiempo de ejecución, porque nada los compiló juntos. Eso es la deriva del contrato, y la solución es estructural: una única fuente de verdad, todo lo demás generado, y una verificación en CI que falla cuando no coinciden.

FIGURA 9.2

Una fuente, extremos generados, deriva verificada



La deriva es una verificación ausente, no un documento ausente. Tanto si la fuente es el esquema como si son los tipos de Rust, todos los demás extremos se generan a partir de ella, y CI demuestra que el código generado está al día.

Hay dos diseños sólidos. Esquema primero: la interfaz se escribe en OpenAPI o Protocol Buffers, y el servidor en Rust y cada cliente se generan a partir de ella. Código primero: los tipos de Rust son la fuente, se genera un esquema a partir de ellos y se versiona en el repositorio, y el diff de ese archivo se revisa como código. En ambos casos, el esquema tiene su propia verificación de cambios incompatibles; para Protocol Buffers, `buf breaking` compara un esquema contra una versión anterior. Para las fronteras dentro del proceso rige la misma regla con distintos generadores: cbindgen escribe cabeceras C, wasm-bindgen escribe el código de enlace en JavaScript, UniFFI escribe bindings para Kotlin, Swift y Python. Los archivos generados se regeneran en CI y la compilación falla si difieren de lo versionado. El capítulo cinco cubre el lado de la seguridad de esas fronteras; este capítulo cubre su estabilidad.

La documentación y los doc tests son el contrato legible

Quienes usan una API leen la documentación, no las firmas de tipos. Rust hace verificable esa parte: los ejemplos de código en los comentarios de documentación compilan y se ejecutan como tests con `cargo test` en los crates de biblioteca, así que un ejemplo que se

desvía de la API hace fallar la compilación.

```
/// Analiza un identificador de pedido como "ORD-1042".
///
/// # Errors
/// Devuelve `ParseError::Format` si falta el prefijo.
///
/// ```
/// use orders_client::OrderId;
/// let id: OrderId = "ORD-1042".parse()?;
/// assert_eq!(id.number(), 1042);
/// # Ok::<(), orders_client::ParseError>(())
/// ```
```

Configure `missing_docs` como warn o deny en los crates publicados, y use los lints de Clippy para las secciones Errors, Panics y Safety que falten, de modo que el contrato nombre lo que puede fallar. La documentación que declara el comportamiento es además el único lugar donde el comportamiento se promete por escrito, lo cual importa en los casos que ninguna herramienta verifica.

> *El desarrollador propone un cambio. El compilador decide si compila. La promesa de la interfaz decide si se publica como parche, menor o mayor, y a quién hay que avisar.*

Ficha de contrato	La promesa de la interfaz
Contrato	Cada interfaz pública tiene un responsable y una versión que sigue las reglas SemVer de Cargo; los cambios incompatibles son mayores, anunciados y documentados.
Amenaza que aborda	Rupturas por sorpresa en los consumidores; deriva entre lenguajes que compila pero falla en tiempo de ejecución.
Responsable	El equipo de producto responsable de cada interfaz; el equipo de plataforma es responsable de las verificaciones y del registro.
Punto de aplicación	CI del pull request: verificación semver, regenerar y comparar, verificación de cambios incompatibles en el esquema, doc tests; aprobación del responsable de la interfaz para cambios incompatibles.
Comportamiento ante fallos	Una ruptura detectada sin versión mayor bloquea la fusión; una verificación que no pudo ejecutarse también bloquea.
Autoridad de excepción y vencimiento	El responsable de la interfaz junto con el responsable de plataforma, solo para una versión concreta, con aviso por escrito a los consumidores.
Evidencia de que el control se ejecutó	Resultados de las verificaciones de semver y de deriva por pull request; el registro de deprecaciones; changelogs con notas de migración.
Evidencia de que el resultado mejoró	Menos incidentes en consumidores atribuibles a cambios de interfaz; deprecaciones eliminadas en la fecha prevista.
Frecuencia de revisión	Registro de interfaces y reloj de deprecación cada trimestre; versión del verificador y lagunas conocidas en cada actualización de la cadena de herramientas.

LISTA DE VERIFICACIÓN DEL CONTRATO

- Cada interfaz pública tiene un responsable con nombre y figura en el registro de interfaces.
- CI ejecuta una verificación semver contra la última versión publicada, y sus lagunas conocidas quedan cubiertas por la revisión humana del diff de la API.
- Los enums y structs pensados para crecer se marcan como non_exhaustive antes de la 1.0, como decisión deliberada.
- Las deprecaciones llevan una versión, un reemplazo y una fecha de eliminación, y la eliminación ocurre solo en una versión mayor.
- Las interfaces entre lenguajes se generan a partir de una única fuente, y CI falla cuando el código generado está obsoleto o el esquema se rompe.
- Los ejemplos de la documentación se ejecutan como tests, y los crates publicados documentan sus errores, panics y condiciones de seguridad.

CAPÍTULO DIEZ

Publicar el artefacto



Todo lo que los capítulos anteriores demostraron, fijaron y admitieron termina en un solo archivo. Un binario o una imagen es lo que realmente se ejecuta, y el compilador no tiene nada que decir sobre cómo se construye, adónde se copia ni qué versión acaba teniendo una máquina. Eso convierte al artefacto en una nueva puerta con su propio contrato: construido una sola vez, a partir de entradas fijadas, firmado, promovido por digest y siempre capaz de volver a la versión anterior.

Si deja de leer aquí, quédese con una regla. **Nunca recompile para promover.** Los bytes que se probaron son los bytes que se publican. Todas las demás reglas de este capítulo protegen esa, tanto para herramientas de escritorio, herramientas de línea de comandos y daemons como para imágenes de contenedor.

Dónde se ubican los tres niveles en esta puerta

El compilador demuestra cosas sobre el código, no sobre la entrega. Un binario con seguridad de memoria compilado en un portátil y copiado a mano a un servidor sigue teniendo seguridad de memoria, y sigue sin estar gobernado.

- **El compilador demuestra** lo que ya demostraba antes: seguridad de memoria y ausencia de carreras de datos en Rust seguro, dentro del límite de confianza del capítulo uno.
- **Las herramientas hacen cumplir** cómo se construye el artefacto: la cadena de herramientas (toolchain) fijada en el capítulo siete, las compilaciones con `--locked`, las imágenes base fijadas, y las firmas, los SBOM y la procedencia del capítulo ocho.
- **Operaciones y las personas son responsables** de su puesta en producción: los canales de publicación, el ritmo del despliegue, la decisión de revertir, y la CPU y el sistema operativo con los que el binario se encuentra en el entorno real.

Herramientas de escritorio, herramientas de línea de comandos y daemons

Un único binario nativo, sin recolector de basura ni máquina virtual que instalar, es una de las razones por las que los equipos eligen Rust, y los equipos de Go conocen bien esa misma ventaja. Aun así, puede enlazar el entorno de ejecución de C u otras bibliotecas del sistema, así que solo conviene llamarlo autocontenido después de inspeccionar sus dependencias dinámicas en cada destino (por ejemplo con `ldd` en Linux). También es una responsabilidad. Nadie lo instala por usted, así que el proceso de publicación tiene que hacer lo que de otro modo haría un gestor de paquetes o una plataforma.

Firma de código. Cada sistema operativo tiene su propia cadena de confianza. Windows usa firmas Authenticode, macOS espera una firma Developer ID y la notarización de Apple antes de que Gatekeeper permita a los usuarios abrir el binario sin advertencias, y los entornos Linux suelen firmar los paquetes para su distribución o firmar los archivos de la versión con Sigstore. Trate las claves de firma como activos de nivel tres: guardadas en un módulo de hardware o en un servicio de firma administrado, usadas solo por el pipeline de publicación y nunca en la máquina de un desarrollador.

Destinos de compilación cruzada. Rust nombra cada plataforma con una tripleta de destino, como `x86_64-pc-windows-msvc`, `aarch64-apple-darwin` o `x86_64-unknown-linux-musl`, y `rustup target add` instala la biblioteca estándar para cada una. El proyecto Rust publica niveles de destino: los destinos Tier 1 tienen garantía de funcionar, los destinos Tier 2 tienen garantía de compilar. Escriba la lista de destinos admitidos en el contrato y compruebe el nivel de cada destino antes de prometérselo a los usuarios. Los enlazadores y las cadenas de herramientas de C para otros destinos son la fricción habitual; herramientas como `cross`, que compila dentro de contenedores, o `cargo-zigbuild` la reducen.

```
# Destinos de release admitidos (extracto; matriz de CI completa en el Apéndice A)
rustup target add x86_64-unknown-linux-gnu aarch64-unknown-linux-gnu
rustup target add x86_64-pc-windows-msvc aarch64-apple-darwin
cargo build --release --locked --target aarch64-unknown-linux-gnu
```

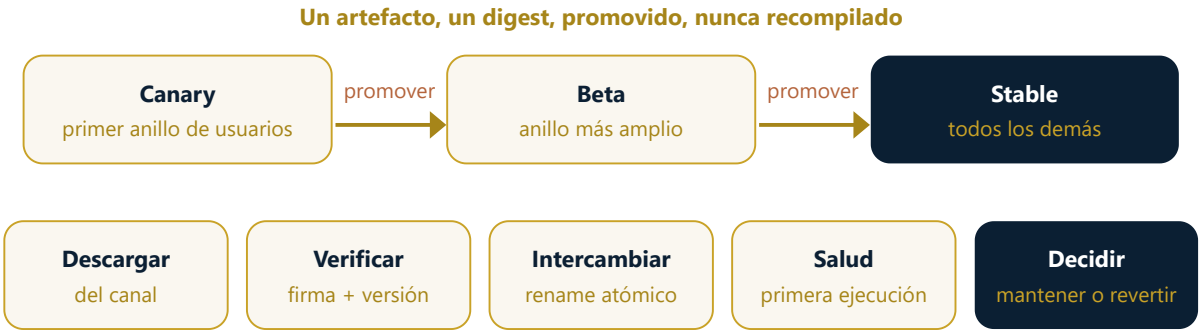
Autoactualización con reversión. Una herramienta que se actualiza a sí misma es un sistema de despliegue en miniatura, y necesita el mismo contrato que uno. La actualización debe verificar una firma y una versión antes de tocar nada. La sustitución del binario debe ser atómica, lo que en la mayoría de los sistemas significa escribir el archivo nuevo junto al anterior y renombrarlo para ocupar su lugar. El binario anterior debe permanecer en disco hasta que el nuevo haya arrancado y superado una comprobación de salud. Si la

comprobación falla, la herramienta vuelve atrás sin preguntar al usuario. Añada una versión mínima admitida que el servidor pueda hacer cumplir, y una forma de pausar las actualizaciones para todos.

Canales de publicación versionados. Un canal como canary, beta o stable es un puntero a una versión, no una compilación aparte. La promoción mueve el puntero. El artefacto, y su digest, siguen siendo los mismos.

FIGURA 10.1

Los canales promueven un artefacto; la autoactualización siempre puede volver atrás



El binario anterior permanece en disco hasta que el nuevo supera su comprobación de salud.

Un canal es un puntero, no una compilación. La promoción mueve el puntero al mismo digest. El lado del cliente del contrato son los cinco pasos de abajo, y el último paso siempre puede elegir la versión anterior.

Imágenes de contenedor

Para los servicios, el artefacto es una imagen. El contrato es el mismo: entradas fijadas, una superficie pequeña, un usuario conocido, una referencia exacta.

Compilaciones multietapa. La etapa de compilación contiene la cadena de herramientas, el código fuente y la caché de compilación. La etapa de ejecución contiene el binario y casi nada más. Los compiladores y los shells se quedan atrás, fuera de la imagen a la que llegaría un atacante.

Caché de la capa de dependencias. Las compilaciones de Rust dedican la mayor parte de su tiempo a compilar dependencias, y un Dockerfile sin optimizar las recompila todas cada vez que cambia cualquier archivo fuente. `cargo-chef` divide la compilación en dos: una etapa planner escribe una receta del grafo de dependencias, y una etapa cook compila solo las dependencias a partir de esa receta. La capa cook se reutiliza hasta que cambian las

dependencias. Su documentación advierte que todas las etapas deben usar la misma versión de Rust, o la caché no se comportará como se espera, lo cual es una razón más por la que importa la cadena de herramientas fijada en el capítulo siete.

```
# Extracto; plantilla completa en el Apéndice A. Sustituya cada <digest> por un sha256 fijado.
FROM rust:1.100.0-slim-bookworm@sha256:<digest> AS chef
RUN cargo install cargo-chef --locked --version 0.1.78
WORKDIR /app

FROM chef AS planner
COPY . .
RUN cargo chef prepare --recipe-path recipe.json

FROM chef AS builder
COPY --from=planner /app/recipe.json recipe.json
RUN cargo chef cook --release --locked --recipe-path recipe.json
COPY . .
RUN cargo build --release --locked --bin service

FROM gcr.io/distroless/cc-debian12:nonroot@sha256:<digest>
COPY --from=builder /app/target/release/service /service
USER 65532:65532
ENTRYPOINT ["/service"]
```

Bases mínimas. Un binario de Rust enlazado contra glibc se ejecuta sobre una base distroless que incluye el entorno de ejecución de C y poco más, como la imagen distroless `cc`. Un binario compilado para un destino musl, como `x86_64-unknown-linux-musl`, se enlaza estáticamente por defecto y puede ejecutarse sobre `scratch`, que está vacía, o sobre la imagen distroless `static`. Recuerde lo que significa vacía: `scratch` no tiene certificados de CA, ni datos de zonas horarias, ni shell. Un servicio que llama a endpoints TLS debe traer sus propios certificados, y nadie hará `exec` en el contenedor para depurarlo. La telemetría del capítulo once tiene que cubrir ese hueco.

> **musl tiene contrapartidas; no es una ganancia gratuita.** Una compilación estática con musl puede permitir una imagen de ejecución mínima, como `scratch` o distroless `static`; mida el tamaño. A cambio, el asignador de memoria por defecto de musl puede rendir peor que el de glibc bajo cargas multihilo con muchas asignaciones, y por eso muchos equipos configuran otro asignador global para las compilaciones musl. La resolución de nombres y algunas dependencias de C también se comportan de forma distinta. Decida midiendo su propio servicio con ambas variantes, con el método de medición del consumo de recursos del Apéndice A y el método de latencia del capítulo once, y anote el resultado en el registro de decisión.

Digests fijados para las imágenes base. Una etiqueta como `cc-debian12` cambia con el tiempo. Un digest no. Fije cada línea `FROM` por digest, y deje que un bot abra un pull request cuando cambie la imagen de origen, para que las actualizaciones lleguen a través de la revisión y del control de admisión en lugar de colarse en silencio con la siguiente compilación.

Usuario no root. Ejecute con un usuario numérico y sin privilegios. El número importa: Kubernetes solo puede verificar `runAsNonRoot` cuando la imagen declara un usuario numérico o el Pod establece `runAsUser`, y por eso el extracto usa `65532`, el usuario nonroot de distroless, en lugar del nombre.

Un destino de CPU portable. `-C target-cpu=native` indica al compilador que use todas las instrucciones que tenga la máquina de compilación. En una estación de trabajo está bien. En un pipeline ata el artefacto a la CPU que tuviera el runner de CI, y el binario puede fallar con una instrucción ilegal en un nodo más antiguo. Elija una línea base con nombre, como un nivel de microarquitectura x86-64, y hágala parte del contrato con los grupos de nodos.

```
# .cargo/config.toml (extracto): una línea base acordada, no la máquina de compilación
[target.x86_64-unknown-linux-gnu]
rustflags = ["-C", "target-cpu=x86-64-v2"]
```

FIGURA 10.2

Del código fuente fijado a una imagen referenciada por digest



La capa de dependencias solo se reconstruye si cambia la receta. Cada base se fija por digest.

Seis pasos, un resultado. El pipeline termina en una referencia de imagen con un digest, una firma y un SBOM. La base de ejecución es una elección entre dos caminos, tomada midiendo y registrada.

El servicio de referencia, y por qué aquí no hay cifras

Los lectores quieren saber cuánto ocupa y qué velocidad tiene. No responderé con cifras en un capítulo, porque una cifra de consumo de recursos sin su método es marketing, y una cifra obtenida en un entorno no es extrapolable a otro. El Apéndice A contiene un pequeño servicio de referencia escrito para este libro, la plantilla completa del Dockerfile (revisada, no compilada para esta edición) y la configuración de compilación anterior, y una medición rápida del consumo de recursos (tamaño del binario y memoria) con su método: el hardware, la versión de la cadena de herramientas y cada paso. No informa de latencia; el capítulo once da el método para medirla. Vuelva a ejecutar ambas cosas en su propio hardware. El número que debe guiar su decisión es el que usted mida.

EJEMPLO: UNA IMAGEN, DOS ENTORNOS

Piense en un equipo que construye una imagen en CI, la prueba en staging y luego la recompila con una etiqueta de versión para producción. La recompilación toma una nueva imagen base publicada durante la noche. Producción ejecuta ahora bytes que nadie probó. Con el contrato, el pipeline compila una sola vez, registra el digest y promueve ese digest; el manifiesto de producción no puede nombrar una etiqueta, así que la recompilación no puede ocurrir por accidente.

> *El desarrollador propone. El compilador decide lo que puede. El pipeline decide qué bytes salen del edificio, y lo decide una sola vez.*

Ficha de contrato

Campo	Publicar el artefacto
Contrato	Cada artefacto se compila una sola vez a partir de una cadena de herramientas fijada, un lockfile incluido y versionado en el repositorio y bases fijadas por digest; se firma, lleva un SBOM y se promueve por digest. Los binarios que se autoactualizan siempre pueden volver a la versión anterior.
Amenaza que aborda	Bytes no probados en producción, imágenes base que cambian sin control, versiones manipuladas o sin firmar, contenedores root, binarios que fallan en CPU más antiguas, actualizaciones que no se pueden deshacer.
Responsable	El equipo de plataforma es responsable de las plantillas de pipeline, el servicio de firma y la lista de imágenes base; cada equipo de producto es responsable de sus destinos, sus canales y su prueba de reversión.
Punto de aplicación	Plantilla de publicación en CI; admisión en el registro y en el clúster que solo acepta imágenes firmadas, referenciadas por digest y ejecutadas como no root; lógica de actualización del cliente que verifica firmas.
Comportamiento ante fallos	La publicación se bloquea si alguna entrada no está fijada, no está firmada o se compiló con `target-cpu=native`; una comprobación de salud fallida tras la actualización revierte automáticamente.
Autoridad de excepción y vencimiento	El responsable de seguridad de plataforma; una excepción nombra el artefacto y el control ausente y vence, como máximo, al cabo de un ciclo de publicación.
Evidencia de que el control se ejecutó	Registros de firma y atestación por digest, el registro del pipeline que muestra `--locked`, los registros de admisión, un resultado de simulacro de reversión por canal.
Evidencia de que el resultado mejoró	Proporción de cargas de trabajo de producción que ejecutan un digest que pasó por staging; reversiones completadas sin recompilar; ningún fallo por instrucción ilegal tras cambios de nodos.
Frecuencia de revisión	Trimestral, y cada vez que cambie una familia de imágenes base, una clave de firma o un destino admitido.

LISTA DE VERIFICACIÓN DEL CONTRATO

- Los manifiestos de producción y los canales de publicación referencian los artefactos por digest, y la promoción nunca recompila.
- Cada imagen base está fijada por digest y solo se actualiza mediante un pull request revisado.
- Las imágenes se ejecutan con un usuario numérico no root sobre una base mínima, y la elección entre glibc y musl se registra con su medición.
- Las compilaciones de release usan una línea base de CPU acordada, nunca `target-cpu=native`.
- Cada binario publicado lo firma el pipeline, y la autoactualización verifica, sustituye el binario de forma atómica y puede revertir.
- La lista de destinos admitidos está escrita, con el nivel de cada destino.

CAPÍTULO ONCE

Operar bajo contrato



Una imagen pequeña y firmada es una promesa sobre lo que se publicó. Todavía no dice nada sobre cómo se comporta el servicio a las tres de la mañana, bajo carga, durante el vaciado de un nodo (node drain) o a mitad del despliegue de una versión defectuosa. Ese comportamiento es responsabilidad de operaciones, y es donde los niveles anteriores o bien rinden frutos o bien dejan de importar sin que nadie lo note. El contrato en esta puerta es sencillo de enunciar: el clúster sabe lo que el servicio necesita, el servicio dice la verdad sobre su salud, se detiene de forma ordenada cuando se le pide, puede verse desde fuera y puede volver atrás.

Para un CTO: Rust puede hacer que un servicio sea ligero y que su latencia sea estable, pero ningún lenguaje hace eso visible. Si la plataforma no puede verlo, el entorno tecnológico no puede contar con ello.

Los requests son datos de entrada, los limits se hacen cumplir, ninguno es una medición

La lectura errónea más común en Kubernetes es tratar los números de recursos de un Pod como una descripción del servicio. No lo son. La documentación de Kubernetes es precisa: el scheduler usa el **request** para decidir en qué nodo se coloca un Pod. Un contenedor puede usar más que su request cuando el nodo tiene margen, y el uso por encima del request es lo primero que mira el kubelet cuando un nodo se queda corto de memoria y tiene que desalojar. Los **limits** se hacen cumplir: los límites de CPU mediante throttling, los de memoria mediante el kernel con terminaciones por falta de memoria (OOM kills), aplicadas de forma reactiva cuando el kernel detecta presión de memoria (documentación de Kubernetes, Resource Management for Pods and Containers, 2026).

Ninguno de los dos números es uso medido. El scheduler coloca el Pod según sus requests. En tiempo de ejecución, el request de CPU se convierte en un peso, de modo que bajo contención el kubelet da al contenedor al menos su request de CPU; un request de memoria

es sobre todo una entrada de planificación, no una reserva firme. Un limit indica dónde interviene el kernel. Solo la telemetría dice lo que el servicio hace en realidad. Un request pequeño escrito porque "Rust es pequeño" es una afirmación, y el scheduler la creerá.

FIGURA 11.1

Tres números distintos, tres responsables distintos



El uso puede superar el request. Memoria sobre el limit: riesgo de OOM kill; la CPU se limita.

Fije los requests a partir del uso medido, no de una creencia. El scheduler confía en el request. El kernel impone el limit. El comportamiento real del servicio vive en la fila del medio, y solo la telemetría de más adelante en este capítulo puede completarla.

La elección de requests y limits también fija la clase de calidad de servicio (QoS) de un Pod. Bajo presión de memoria, el kubelet no ordena los Pods por clase de QoS. Los ordena según si el uso supera los requests, luego por la prioridad del Pod y luego por el uso relativo a los requests (documentación de Kubernetes, Node-pressure Eviction, 2026). La QoS es una buena estimación de ese orden para la memoria, no la regla en sí. La tabla trata de la presión de memoria: la presión de disco ordena los Pods por uso de disco, y la presión de inodos o de PID solo por prioridad.

Clase de QoS	Cuándo la obtiene un Pod	Qué significa bajo presión de memoria
Guaranteed	Cada contenedor tiene requests de CPU y memoria iguales a sus limits.	Se desaloja al final, junto con los Pods Burstable que están por debajo de sus requests, según prioridad. El uso de otro Pod por sí solo no lo desaloja, pero sigue siendo desalojable, empezando por la prioridad más baja, cuando los daemons del sistema exceden su reserva. El valor por defecto correcto para servicios sensibles a la latencia.
Burstable	Al menos un contenedor tiene un request o un limit, pero el Pod no es Guaranteed.	Queda entre los primeros cuando usa más que su request; por debajo de su request, se ordena junto a los Pods Guaranteed según prioridad.
BestEffort	Ningún request ni limit.	Cualquier uso supera un request de cero, así que queda entre los primeros. No es aceptable para nada que tenga un contrato.

Los límites de CPU merecen una valoración aparte. El throttling no hace caer un servicio; lo ralentiza, y la ralentización aparece en la cola de latencia. Algunos equipos dejan la CPU sin límite en los servicios sensibles a la latencia; otros necesitan límites de CPU estrictos por equidad. Ambas opciones son defendibles si la decisión se basa en las métricas de throttling.

Tres probes, tres preguntas distintas

Probe	Pregunta que responde	Qué hace Kubernetes ante el fallo	Regla
Startup	¿Ha terminado de arrancar el servicio?	Aplaza las probes de liveness y readiness hasta que tiene éxito; reinicia el contenedor si nunca lo tiene.	Úsela en lugar de largos retrasos de liveness.
Readiness	¿Debe este Pod recibir tráfico ahora?	Quita el Pod de los endpoints del Service; no lo reinicia.	Puede reflejar dependencias críticas y la terminación ordenada.
Liveness	¿Está el proceso atascado sin posibilidad de recuperarse?	Reinicia el contenedor.	Nunca verifica una dependencia compartida, como una base de datos.

La regla de liveness es la que los equipos rompen, como muestra el ejemplo al final de este capítulo. Liveness solo debe preguntar si este proceso todavía puede avanzar. Rust no evita los interbloqueos, así que una comprobación de liveness que usa el mismo executor que atiende las peticiones reales vale más que una que devuelve una constante. Una forma

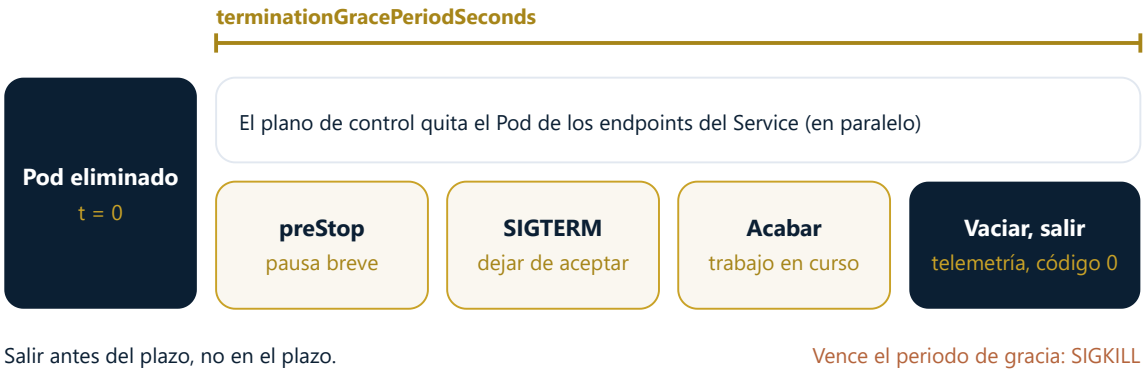
sencilla: el handler lanza una tarea trivial en el entorno de ejecución asíncrono y espera su resultado, de modo que un entorno de ejecución que ya no planifica trabajo hace fallar la probe.

Detenerse de forma ordenada con SIGTERM

Cuando se elimina un Pod, el kubelet ejecuta el hook `preStop` si lo hay y envía SIGTERM al proceso principal. Cuando termina el periodo de gracia, los procesos restantes reciben SIGKILL. El periodo de gracia es de 30 segundos por defecto y se configura con `terminationGracePeriodSeconds` (documentación de Kubernetes, Pod Lifecycle, 2026). Al mismo tiempo, y sin un orden garantizado, el plano de control quita el Pod de los endpoints del Service, de modo que durante una breve ventana todavía pueden llegar peticiones nuevas después de la señal.

FIGURA 11.2

Apagado ordenado dentro del periodo de gracia



Ocurren dos cosas a la vez. Los endpoints se actualizan mientras el contenedor se apaga. La breve pausa de preStop cubre la ventana en la que todavía llega tráfico; después el servicio completa las solicitudes en curso, envía la telemetría pendiente y sale con holgura dentro del periodo de gracia.

En un servicio Rust sobre Tokio, el patrón es un future de apagado que se resuelve con SIGTERM y que se entrega al servidor para que deje de aceptar conexiones y termine las que tiene abiertas. El soporte de señales de Tokio necesita la feature `signal`. El extracto siguiente usa axum; otros servidores tienen el mismo hook.

```
use tokio::signal::unix::{signal, SignalKind};

async fn shutdown_signal() {
    let mut term = signal(SignalKind::terminate()).expect("install SIGTERM handler");
    tokio::select! {
        _ = term.recv() => {}
        _ = tokio::signal::ctrl_c() => {}
    }
    tracing::info!("shutdown signal received, draining");
}

// Deja de aceptar conexiones, completa las solicitudes en curso y luego retorna.
axum::serve(listener, app)
    .with_graceful_shutdown(shutdown_signal())
    .await?;
// Fuerce aquí el envío de los datos pendientes de los exportadores de telemetría, antes de que
main retorne.
```

Kubernetes no cambia la readiness cuando llega SIGTERM. Si la probe de readiness debe pasar a falso mientras el servicio termina de forma ordenada, la aplicación tiene que cambiarla en su ruta de apagado, como hace el servicio de referencia del Apéndice A.

Hay dos detalles que suelen sorprender a los equipos. Primero, una imagen distroless no tiene shell, así que un hook `preStop` no puede ejecutar `sleep` como comando. Use la acción integrada `sleep` para los hooks de ciclo de vida, que estuvo en beta y habilitada por defecto desde Kubernetes 1.30 y es estable desde la 1.34 (documentación de Kubernetes, feature gates, 2026), o espere brevemente dentro del proceso tras SIGTERM antes de cerrar el listener. Segundo, las tareas en segundo plano lanzadas con `tokio::spawn` no se detienen porque el servidor se haya detenido. Deles una señal de cancelación, y acote toda la terminación ordenada con un plazo más corto que el periodo de gracia menos la pausa de preStop.

```
# Extracto del Deployment; plantilla completa en el Apéndice A. Los valores de recursos provienen
de mediciones.
# Burstable a propósito: sin límite de CPU (véase la limitación de CPU más arriba). Para
Guaranteed,
# fije los límites de cpu y memoria iguales a sus requests.
spec:
  terminationGracePeriodSeconds: 30
  containers:
  - name: service
    image: registry.example.com/service@sha256:<digest>
    resources:
      requests: { cpu: "<measured>", memory: "<measured>" }
      limits:   { memory: "<measured>" }
    lifecycle:
      preStop: { sleep: { seconds: 5 } }   # acción integrada; no requiere shell
    startupProbe:   { httpGet: { path: /healthz, port: 8080 } }
    livenessProbe:  { httpGet: { path: /healthz, port: 8080 } }
    readinessProbe: { httpGet: { path: /readyz, port: 8080 } }
    securityContext: { runAsNonRoot: true, readOnlyRootFilesystem: true,
allowPrivilegeEscalation: false }
```

Visibilidad: registros, trazas y métricas

Una imagen mínima sin shell solo es operable si se explica a sí misma. Tres señales lo hacen posible.

- **Registros estructurados.** Un evento JSON por línea, con campos en lugar de frases: ID de petición, ID de traza, ruta, resultado. El crate `tracing` es la capa de instrumentación habitual en Rust, y `tracing-subscriber` puede emitir JSON.
- **Trazas.** Los spans de `tracing` se corresponden de forma natural con trazas distribuidas, y los crates de OpenTelemetry para Rust las exportan por OTLP al backend que use el entorno. A noviembre de 2026, la página de OpenTelemetry para Rust (https://opentelemetry.io/docs/languages/rust/) indica métricas y registros como estables y trazas como beta. Revísela antes de estandarizar, y fije juntos los crates de OpenTelemetry; avanzan como un conjunto.
- **Métricas.** Para cada endpoint: tasa, errores y duración, con la duración registrada como histograma. Añada las señales del entorno de ejecución que explican la cola de latencia: el throttling de CPU del clúster, la memoria frente al limit, y la profundidad de cola o el número de tareas del entorno de ejecución asíncrono.

```
// Extracto de main.rs: registros JSON con el contexto del span
tracing_subscriber::fmt()
    .json()
    .with_current_span(true)
    .init();
```

p99, medido con honestidad

Una latencia de cola predecible es una de las principales razones por las que los equipos eligen Rust frente a un entorno de ejecución con recolector de basura, y también es la afirmación que con más frecuencia se hace sin evidencia. Los recolectores modernos, como ZGC en la JVM, mantienen las pausas cortas, y un servicio Rust puede presentar igualmente latencias elevadas en los percentiles superiores por contención de bloqueos, una llamada bloqueante en un hilo de trabajo asíncrono, el comportamiento del asignador o el throttling de CPU. Una medición rigurosa tiene reglas.

REGLAS PARA UNA AFIRMACIÓN DE LATENCIA

- La misma carga de trabajo, el mismo hardware y la misma configuración del clúster para cada candidato que se compare.
- Calentar primero, y descartar el periodo de calentamiento de los resultados.
- Informar percentiles (p50, p99, p99.9) a partir de un histograma, nunca un promedio, y nunca promediar percentiles entre instancias; en su lugar, combinar los histogramas.
- Generar la carga a una tasa fija. Una herramienta de bucle cerrado que espera cada respuesta antes de enviar la siguiente petición oculta las esperas prolongadas; es el problema de la omisión coordinada (coordinated omission), y hace que la cola de latencia parezca mejor de lo que perciben los usuarios.
- Publicar el método junto con el número: herramienta, tasa, duración, hardware, versión de la cadena de herramientas.

El Apéndice A informa solo del consumo de recursos del servicio de referencia, no de su latencia, porque su cliente secuencial simple es el patrón de bucle cerrado que estas reglas excluyen. Mida la latencia de su propio servicio con estas reglas antes de que un número aparezca en un registro de decisión.

Volver atrás

`kubectl rollout undo` devuelve un Deployment a su revisión anterior, es decir, a la especificación de Pod anterior. Si esa especificación nombraba una etiqueta, los bytes detrás de ella pueden haber cambiado. Con digests, la revisión anterior es exactamente el artefacto anterior. Además, la reversión solo es real cuando los cambios de base de datos son compatibles hacia atrás, de modo que la versión anterior siga funcionando con el nuevo esquema, y cuando se ha ensayado.

EJEMPLO: LA TORMENTA DE REINICIOS

Piense en un equipo cuya probe de liveness consulta la base de datos compartida. Una reconstrucción lenta de un índice lleva el tiempo de consulta más allá del timeout de la probe. Todas las réplicas fallan la liveness y se reinician a la vez, los reinicios añaden carga de conexiones y el servicio deja de responder por completo. Con el contrato, liveness verifica solo el proceso, readiness saca de rotación las réplicas lentas, y el servicio se degrada en lugar de fallar.

El compilador demostró que el servicio tiene seguridad de memoria. Solo la telemetría del clúster puede demostrar que está sano. Cada línea se sostiene solo hasta donde alguien puede ver que se sostiene.

Ficha de contrato

Campo	Operar bajo contrato
Contrato	Cada servicio declara requests a partir del uso medido, se ejecuta como Guaranteed o como Burstable justificado, expone probes de startup, readiness y liveness con significados distintos, termina de forma ordenada al recibir SIGTERM dentro del periodo de gracia, emite registros estructurados, trazas y métricas en histograma, y se despliega por digest con una reversión ensayada.
Amenaza que aborda	Nodos con sobreasignación de recursos, desalojos inesperados, tormentas de reinicios, peticiones perdidas durante los despliegues, latencia de cola invisible, afirmaciones de rendimiento sin respaldo, reversiones a bytes desconocidos.
Responsable	El equipo de plataforma es responsable de la plantilla de Deployment, la política de admisión y el pipeline de telemetría; cada equipo de producto es responsable de sus probes, su código de apagado, sus valores de recursos y su presupuesto de latencia.
Punto de aplicación	Política de admisión del clúster (digests, no root, requests presentes, probes presentes); verificación de manifiestos en CI; una prueba de apagado en el pipeline que envía SIGTERM bajo carga.
Comportamiento ante fallos	Los manifiestos sin requests, probes o digests se rechazan en la admisión; una versión cuyo p99 supera su presupuesto en staging no se promueve.
Autoridad de excepción y vencimiento	El responsable de plataforma para el nivel del servicio; una excepción nombra el control ausente y el compensatorio y vence dentro de un trimestre (política de ejemplo de un entorno).
Evidencia de que el control se ejecutó	Registros de admisión, resultados de las pruebas de apagado, el método de prueba de carga y el histograma publicados por versión, registros de simulacros de reversión.
Evidencia de que el resultado mejoró	Los errores durante despliegues y vaciados de nodos tienden a bajar; el p99 se mantiene dentro del presupuesto entre versiones; los recursos solicitados siguen al uso medido; el tiempo de reversión es conocido.
Frecuencia de revisión	Trimestral, después de cada incidente que implique reinicios, desalojos o reversiones, y cuando cambia la versión del clúster.

LISTA DE VERIFICACIÓN DEL CONTRATO

- Los requests de recursos se fijan a partir del uso medido y se revisan contra la telemetría, no se escriben a partir de expectativas.
- Liveness nunca verifica una dependencia compartida; las probes de startup y readiness existen y significan lo que dicen.
- Al recibir SIGTERM, el servicio deja de aceptar solicitudes, completa las que están en curso, detiene las tareas en segundo plano y sale dentro del periodo de gracia, y una prueba lo demuestra.
- Los registros son estructurados, las trazas se exportan por OpenTelemetry y la duración se registra como histograma.
- Cada afirmación de latencia lleva su método: carga a tasa fija, calentamiento, percentiles, hardware y cadena de herramientas.
- Los Deployments referencian digests, y la reversión se ensaya cada trimestre.

CAPÍTULO DOCE

El modelo operativo



Un equipo que escribe Rust es una elección técnica. Veinte equipos que escriben Rust son un modelo operativo, lo haya diseñado alguien o no. Los contratos de este libro solo se sostienen si alguien es responsable de cada uno, si alguien puede conceder una excepción y retirarla, y si alguien comprueba, en una fecha, que los controles siguen ejecutándose y que los resultados siguen justificando la elección.

La versión corta para un CTO: un equipo de plataforma es responsable de la maquinaria compartida, los equipos de producto son responsables de su código y de sus excepciones, cada contrato es una ficha con un responsable y evidencia, un pequeño cuadro de mando muestra los resultados, y una vez al año la decisión se vuelve a tomar de forma deliberada.

Quién es responsable de qué

El tercer nivel, operaciones y personas, abarca toda esta puerta. El compilador no puede ocupar un puesto en un comité de revisión, así que la primera pregunta de diseño es la responsabilidad. El reparto que funciona en la mayoría de los entornos tecnológicos es un equipo de plataforma y muchos equipos de producto.

FIGURA 12.1

Las plantillas bajan, las excepciones y la evidencia suben



La responsabilidad es el contrato que está debajo de todos los demás contratos. El equipo de plataforma hace que el camino correcto sea el camino fácil. Los equipos de producto son responsables de lo que construyen y piden lo que necesitan. El foro, que puede ser un punto recurrente del orden del día en lugar de un nuevo comité, decide las excepciones y revisa la degradación de los controles.

- **El equipo de plataforma es responsable** de la cadena de herramientas (toolchain) fijada y su calendario de actualización, de las plantillas de CI que ejecutan los controles de las Partes II y III, de la política de admisión, de las plantillas de imagen y de Deployment de la Parte IV, de los crates estándar del entorno (incluido el entorno de ejecución asíncrono) y de las plantillas de contrato.
- **Los equipos de producto son responsables** de sus crates, sus interfaces públicas, sus entradas en el registro de unsafe, sus probes y presupuestos de latencia, y sus excepciones.
- **El foro de gobernanza** concede las excepciones que superan la autoridad de un equipo, dirige la revisión de degradación de los controles y prepara la reevaluación anual de la decisión. Manténgalo pequeño: el responsable de plataforma, un responsable de seguridad y responsables de equipos de producto de forma rotativa.

Dos niveles de gobernanza

Un modelo de gobernanza dimensionado para cincuenta equipos resultará aplastante para tres. El capítulo tres presentó un nivel mínimo viable; aquí aparece junto al nivel completo.

Área	Nivel mínimo viable	Nivel completo
Cadena de herramientas	rust-toolchain.toml fijado; Cargo.lock versionado en cada repositorio	Calendario de actualización de la cadena de herramientas para todo el entorno, a cargo del equipo de plataforma, con una vía de actualización de emergencia
Control del código	fmt y Clippy en CI, con las advertencias tratadas como errores	Perfil de lints compartido, controles probados, Miri y fuzzing donde exista código unsafe o de parsing
Dependencias	Escaneo de avisos en cada compilación; lista de licencias permitidas	Control de admisión completo: prohibiciones, orígenes, revisión de dependencias, réplica privada del registro (mirror)
Excepciones	Un único archivo de registro para unsafe, FFI y excepciones, cada una con responsable y vencimiento	Ciclo de vida de las excepciones con autoridad de excepción según el nivel de riesgo y revisión de degradación de los controles
Artefactos	Compilar una vez, desplegar por digest	Firma, SBOM, procedencia, admisión que rechaza todo lo demás
Entorno de ejecución	Probes, terminación ordenada con SIGTERM, registros estructurados	Plantilla de Deployment, pipeline de OpenTelemetry, presupuestos de latencia con método publicado
Decisión	ADR con fecha de revisión	ADR con contrato de renovación, cuadro de mando y reevaluación anual de la decisión

Pase un área de la columna izquierda a la derecha cuando el cuadro de mando diga que el riesgo lo justifica, no porque exista la columna derecha.

La ficha de contrato, completa

Cada uno de los capítulos cuatro a once terminó con una ficha, la unidad de gobernanza de este libro. Cada campo tiene una función.

Campo	Qué debe decir
Contrato	Una o dos frases, verificables. "Cada versión está firmada" es un contrato. "Nos tomamos la seguridad en serio" no lo es.
Amenaza que aborda	El fallo concreto que el contrato previene o detecta. Si nadie puede nombrar uno, retire el contrato.
Responsable	Un rol, no una persona, con la persona que lo ocupa registrada en otro lugar.
Punto de aplicación	Dónde lo verifica una máquina: compilador, trabajo de CI, política de admisión, entorno de ejecución. La "revisión de código" por sí sola es un control de nivel tres, y así debe indicarse.
Comportamiento ante fallos	Qué ocurre cuando la verificación falla: bloquear, advertir, revertir. Un control que solo advierte es un informe.
Autoridad de excepción y vencimiento	Quién puede decir "sí, por ahora", y cuánto puede durar como máximo ese "por ahora".
Evidencia de que el control se ejecutó	El registro que prueba que la verificación se ejecutó, legible sin preguntar a nadie.
Evidencia de que el resultado mejoró	La medida que muestra que la amenaza es realmente menor. Es el campo que la mayoría de las fichas cubre peor.
Frecuencia de revisión	Una fecha, más los eventos que obligan a adelantar la revisión.

Guarde las fichas como archivos junto al control, de modo que ambos cambien en el mismo pull request. La revisión funciona en tres pasos. Una ficha nueva la propone su responsable y la revisan el equipo de plataforma, en cuanto al punto de aplicación, y seguridad, en cuanto a la amenaza. Entra como borrador y solo pasa a estar activa cuando ambos campos de evidencia apuntan a algo que existe. En cada fecha de revisión, el foro lee la evidencia, no la prosa, y decide mantener, endurecer, relajar o retirar.

```
# contracts/admission-gate.card.yaml (extracto; plantilla en el Apéndice A)
contract: "Ninguna dependencia con un aviso abierto y sin excepción entra en una compilación de
release."
owner: platform-security-lead
enforcement_point: ci/release-gate
failure_behavior: block
waiver: { authority: platform-security-lead, max_days: 30 }   # política de ejemplo de un entorno
evidence_ran: "registro del trabajo del control en cada release"
evidence_outcome: "días desde la publicación del aviso hasta su corrección en producción"
review: { every: quarter, also_on: [gate-bypass, incident] }
status: active
```

Un cuadro de mando que muestra resultados, no rituales

Las métricas rituales solo cuentan actividad, y siempre arrojan buenos resultados. Las métricas de resultados muestran si el entorno es más seguro o menos costoso de modificar, y pueden arrojar resultados desfavorables.

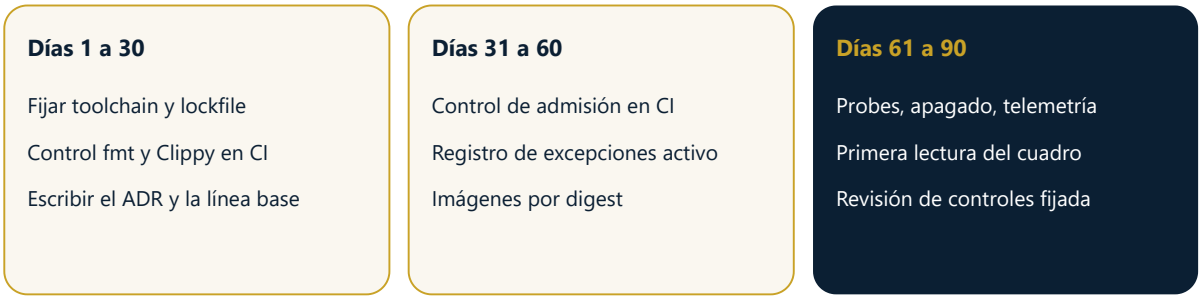
Medida	Qué muestra	El ritual que reemplaza
Tiempo desde que se publica un aviso hasta que se corrige en producción	La respuesta de la cadena de suministro, de principio a fin	Número de escaneos ejecutados
Excepciones vencidas, y excepciones renovadas más de una vez	Si las excepciones siguen siendo temporales	Número de excepciones aprobadas
Bloques unsafe con un responsable, un invariante declarado y tests bajo Miri donde Miri admita el código, o en su defecto fuzzing, sanitizers o tests de modelo	Qué parte del límite de confianza está realmente bajo control	Líneas de código unsafe
Incidentes de producción por nivel: compilador, herramientas, operaciones	De dónde vienen realmente los fallos	Número total de incidentes
Tiempo para llevar todo el entorno a una nueva versión de la cadena de herramientas	Si la fijación es un control o una trampa	Versión de la cadena de herramientas notificada
Versiones publicadas dentro del presupuesto de latencia, según el método publicado	Si la afirmación sobre el entorno de ejecución se sostiene	Una única diapositiva de benchmark
Semanas que tarda un ingeniero nuevo en fusionar un cambio revisado	Si la curva de aprendizaje está gestionada	Horas de formación registradas

La fila de incidentes por nivel es la que más importa. Si la mayoría de los incidentes vienen de la lógica y de las operaciones, esa es la forma esperada una vez que el compilador ha eliminado una clase de errores, y muestra dónde invertir a continuación.

Un despliegue en 30-60-90 días

FIGURA 12.2

Primero lo mínimo viable, luego la evidencia, luego el nivel completo donde compense



Primero el nivel mínimo viable; el completo solo donde el cuadro de mando diga que compensa.

Noventa días hasta una línea base gobernada, como supuesto de planificación. Dimensione las fases según su entorno. Cada fase añade controles que producen la evidencia que usa la fase siguiente. Para el día noventa, el entorno tiene su primer cuadro de mando honesto y una fecha para su primera revisión de degradación de los controles.

Empiece con un servicio que tenga un responsable real y un riesgo moderado, y haga que sus controles pasen antes de que sean obligatorios para los demás. En el segundo mes, active los contratos de dependencias y de excepciones en modo de advertencia durante un periodo fijo, y después en modo de bloqueo. En el tercer mes, añada el contrato del entorno de ejecución y haga la primera lectura del cuadro de mando. Estará incompleta. Publíquela de todos modos, indicando las lagunas.

Habilidades y contratación

Seré claro sobre esto. Según mi experiencia, la oferta de ingenieros con experiencia en Rust es más pequeño que el de Java, .NET, Go o TypeScript; contrástelo con su propio mercado de contratación, tal como el capítulo dos le pidió que lo incluyera como costo en la decisión. La respuesta que funciona es sobre todo interna. Forme a los ingenieros en los que ya confía, porque el conocimiento del dominio y el criterio son más difíciles de contratar que la sintaxis. Empareje a los recién llegados con revisores experimentados en Rust para sus primeros cambios. Mantenga una guía breve del entorno que establezca criterios claros y

que nombre los crates estándar, el patrón de errores y el entorno de ejecución asíncrono. Cuente con que la curva de aprendizaje aparezca en la fila de incorporación del cuadro de mando, y gestiónela como un costo.

Al contratar, busque pensamiento sistémico. Según mi experiencia, un ingeniero que razona bien sobre la propiedad (ownership) en C++ o sobre la concurrencia en Go aprende las reglas de Rust más rápido de lo que sugiere una descripción de puesto. No deje que Rust se convierta en un símbolo de estatus dentro de la organización. Un equipo que mantiene un servicio en Go o .NET porque se ajusta mejor está siguiendo el registro de decisión, no resistiéndose a él.

La revisión de degradación de los controles y la reevaluación de la decisión

Todo control se degrada. Las excepciones se renuevan por costumbre, un atajo para una versión urgente se queda para siempre, y las fichas siguen "activas" después de que se rompan sus enlaces de evidencia. La **revisión de degradación de los controles**, una vez por trimestre, existe para detectarlo. Para cada ficha hace cuatro preguntas: ¿se ejecutó el control?, ¿es legible la evidencia?, ¿cambiaron los resultados?, ¿hay excepciones que han sobrevivido a su razón de ser? Las fichas que no superan la revisión vuelven a borrador, con un responsable y una fecha.

Una vez al año, el foro prepara la **reevaluación de la decisión**. El capítulo dos terminó con un contrato de renovación: resultados esperados, una línea base, una fecha de revisión, criterios de reversión y una ruta de salida. La reevaluación contrasta el cuadro de mando con ese contrato y responde con honestidad a una pregunta: ¿la evidencia sigue respaldando Rust para el alcance para el que se eligió? La respuesta puede ser "continuar", "ampliar", "reducir" o "salir". Cada una es un resultado legítimo. Una decisión cuyo resultado no puede ser distinto no es una decisión; es un hábito.

> **EJEMPLO: UNA REDUCCIÓN QUE FUE UN ÉXITO**
>
> Piense en un entorno que eligió Rust para su plano de datos y más tarde lo permitió para herramientas internas. En la reevaluación, el plano de datos está dentro de su presupuesto de latencia, mientras que las herramientas muestran una incorporación lenta y ninguna ganancia en los resultados. El foro reduce el alcance: las herramientas pasan al principal lenguaje administrado del entorno en su próxima reescritura, y el ADR registra por qué. Así es como funciona el contrato.

Cerrar el libro

Este libro empezó con el deseo de ser dueños de los bits. Los tres niveles son la forma en que ese deseo se convierte en algo con lo que el entorno tecnológico puede contar. El compilador demuestra lo que puede demostrar, dentro del límite de confianza, cada vez. Las herramientas hacen cumplir lo que el compilador no puede ver, mientras alguien comprueba que los controles siguen bloqueando. Operaciones y las personas son responsables del resto.

Así que la frase de la que partí necesita su forma completa. El desarrollador propone. El compilador decide lo que puede. Las personas deciden el resto, con contratos que tienen responsables, vencimientos y evidencia, y una fecha en la que volverán a decidir.

> *El desarrollador propone, el compilador decide lo que puede y las personas deciden el resto. Confianza, ganada en cada capa.*

LISTA DE VERIFICACIÓN DEL CONTRATO

- Cada contrato tiene una ficha en el repositorio que contiene su control, con un rol como responsable y ambos campos de evidencia completos.
- El equipo de plataforma es responsable de la cadena de herramientas, las plantillas de CI y las plantillas de contrato; los equipos de producto son responsables de sus crates y de sus excepciones.
- El entorno ha elegido su nivel de gobernanza por área, y solo pasa un área al nivel completo con evidencia del cuadro de mando.
- El cuadro de mando informa de resultados, incluidos los incidentes por nivel, y se publica aunque esté incompleto.
- Una revisión trimestral de degradación de los controles y una reevaluación anual de la decisión, frente al contrato de renovación del capítulo dos, están en el calendario con responsables nombrados.

APÉNDICE A

Plantillas y el servicio de referencia

Los capítulos dos a doce muestran extractos breves. Los archivos completos están aquí, listos para copiar, cada uno con la versión de la cadena de herramientas con la que se verificó; la única excepción deliberada, el trabajo de release de la plantilla de CI, está marcada como esqueleto. Las plantillas envejecen: cuando actualice las versiones fijadas, vuelva a ejecutar cada una y actualice su línea de verificación, de la misma forma en que los capítulos piden renovar una decisión.

La segunda mitad del apéndice describe el servicio de referencia: un pequeño servicio HTTP escrito para este libro, construido con estas plantillas y medido con un método que se puede repetir. Es la única fuente de las cifras de consumo de recursos del libro. Sus cifras describen ese servicio en ese hardware, no a Rust en general ni a su carga de trabajo.

La fijación: rust-toolchain.toml

Ubicado en la raíz del repositorio, este archivo hace que `rustup` seleccione el mismo compilador en cada portátil y en cada runner de CI. Actualícelo según el calendario que la organización fije para todo el entorno tecnológico, nunca sin aviso en una rama de funcionalidad.

```
# Verificado con Rust 1.100.0 estable, noviembre de 2026
[toolchain]
channel = "1.100.0"
components = ["rustfmt", "clippy"]
profile = "minimal"
# Añada los destinos de release que admite el entorno tecnológico, por ejemplo:
# targets = ["x86_64-unknown-linux-musl", "aarch64-unknown-linux-gnu"]
```

Manifiesto del paquete y perfil de release

El manifiesto declara la edición y la versión mínima de Rust admitida. El perfil release sacrifica tiempo de compilación a cambio de un binario más pequeño y más rápido. `panic = "abort"` termina el proceso ante un pánico en lugar de desenrollar la pila, de modo que el orquestador lo reinicia; elíjalo de forma deliberada. `overflow-checks` se declara de forma explícita en lugar de dejarse al valor por defecto, como pide la lista de

verificación del capítulo cuatro: este perfil prefiere un pánico a un desbordamiento silencioso. Un equipo que conserva el valor por defecto del perfil release escribe `overflow-checks = false` y usa aritmética comprobada (checked) para importes y cuotas.

El servicio de referencia promete 1.85, la misma MSRV que el ejemplo de workspace del capítulo siete. Es el mínimo que imponen sus dependencias (las versiones fijadas en el lockfile declaran como máximo una rust-version de 1.85) y la primera versión de Rust con la edición 2024. La promesa se demuestra con `cargo +1.85.0 check --locked --all-targets`, que se completa sin errores; el trabajo `msrv` de la plantilla de CI ejecuta la misma verificación.

```
# Verificado con Rust 1.100.0 estable, noviembre de 2026
[package]
name = "refservice"
version = "0.1.0"
edition = "2024"          # estable desde Rust 1.85
rust-version = "1.85"     # la MSRV que promete este crate
publish = false
license = "MIT OR Apache-2.0"

[profile.release]
lto = true                # optimización del programa completo; compilaciones más lentas
codegen-units = 1         # mejor optimización; compilación menos paralela
strip = true              # elimina los símbolos del binario entregado
panic = "abort"           # sin desenrollado; el proceso termina ante un panic
overflow-checks = true    # explícito, no por defecto: panic en lugar de aritmética modular
```

Las tablas [lints]

Los lints del manifiesto viajan con el código, de modo que la política es la misma en el editor y en CI. En un workspace, colóquelos bajo `[workspace.lints]` y añada `[lints] workspace = true` a cada miembro. Hay dos tablas, porque un binario de servicio y una biblioteca tienen reglas de pánico distintas (capítulo cuatro). Ambas fijan `unsafe_code` en `deny`, de modo que un crate que necesita unsafe puede permitirlo en un único módulo designado; un crate que nunca necesita unsafe lo restringe a `forbid` (capítulo cinco). Es lo que hace el servicio de referencia.

```
# Verificado con Rust 1.100.0 estable, noviembre de 2026
# Binario de servicio
[lints.rust]
unsafe_code = "deny"                # "forbid" si el crate nunca necesita unsafe
unsafe_op_in_unsafe_fn = "deny"     # para los crates del registro de unsafe
missing_debug_implementations = "warn"

[lints.clippy]
all = { level = "warn", priority = -1 }
undocumented_unsafe_blocks = "deny"
unwrap_used = "warn"
expect_used = "warn"
```

```
# Verificado con Rust 1.100.0 estable, noviembre de 2026
# Biblioteca: ningún panic alcanzable desde la entrada
[lints.rust]
unsafe_code = "deny"                # "forbid" si el crate nunca necesita unsafe
unsafe_op_in_unsafe_fn = "deny"
missing_debug_implementations = "warn"
missing_docs = "warn"

[lints.clippy]
all = { level = "warn", priority = -1 }
undocumented_unsafe_blocks = "deny"
unwrap_used = "deny"
expect_used = "deny"
panic = "deny"
indexing_slicing = "warn"
```

El control de admisión: deny.toml

cargo-deny lee este archivo. Cada entrada de `ignore` necesita un registro de excepción con vencimiento. La lista de licencias y la prohibición de `openssl` son ejemplos; el área jurídica y el equipo de plataforma son responsables de las definitivas. Este es el archivo completo que usa el servicio de referencia; el capítulo ocho muestra un extracto con una excepción de ejemplo. Las versiones duplicadas se rechazan, como en el capítulo ocho: un duplicado que el equipo todavía no puede eliminar recibe una entrada `skip` con un motivo, de modo que la excepción es visible y tiene una salida prevista. El archivo se verificó con cargo-deny 0.20.2, que rechaza claves y valores desconocidos al cargar el archivo. Los avisos de vulnerabilidad siempre hacen fallar la verificación, así que no hay clave para ellos (0.20.2 indica que la antigua clave `vulnerability` fue eliminada). `unmaintained` admite `all`, `workspace`, `transitive` o `none`. Una entrada ignore de avisos lleva un `id` y un `reason` (una versión retirada puede indicarse, en su lugar, con una especificación de crate) y no tiene campo de vencimiento, así que el vencimiento consta en el registro de excepción. Una entrada skip nombra un crate con un requisito de versión después de dos puntos, como en `syn:<3`. Es la forma documentada, no una errata: en una especificación de crate de cargo-deny, `:`

introduce un requisito de versión y `@` una versión exacta (`syn@2.0.0` significa `syn:=2.0.0`), y 0.20.2 acepta ambas (documentación de cargo-deny). La lista `targets` contiene todos los destinos para los que la matriz de compilación de CI genera entregables, de modo que también se evalúan las dependencias específicas de cada plataforma. La entrada del registro es la URL del índice de crates.io que Cargo escribe en `Cargo.lock`.

```
# deny.toml (completo)
# Configuración de cargo-deny para refservice.
# Verificado con cargo-deny 0.20.2 (cargo deny check pasa), Rust 1.100.0, noviembre de 2026.

[graph]
# Todos los destinos que entrega la plantilla de CI (matriz de compilación del Apéndice A).
targets = [
    "x86_64-unknown-linux-gnu",
    "aarch64-unknown-linux-gnu",
    "x86_64-pc-windows-msvc",
    "aarch64-apple-darwin",
]
all-features = false

[advisories]
# Los avisos de vulnerabilidad siempre hacen fallar el control; también los crates sin
mantenimiento y las versiones retiradas (yanked).
unmaintained = "all"
yanked = "deny"
# Cada entrada de ignore necesita un registro de excepción (Apéndice A) y un vencimiento.
ignore = []

[licenses]
allow = [
    "MIT",
    "Apache-2.0",
    "BSD-3-Clause",
    "ISC",
    "Unicode-3.0",
]
confidence-threshold = 0.9

[bans]
multiple-versions = "deny"
wildcards = "deny"
# Una sola pila TLS para el entorno tecnológico (prohibición de ejemplo; el servicio no depende
de TLS).
deny = [{ crate = "openssl", use-instead = "rustls" }]
# Cada skip necesita un motivo y un plan de salida; elimínelo cuando upstream converja.
skip = [
    { crate = "syn:<3", reason = "proc-macro build dependency of tracing-attributes; remove when
it moves to syn 3" },
]

[sources]
```

```
unknown-registry = "deny"
unknown-git = "deny"
allow-registry = ["https://github.com/rust-lang/crates.io-index"]
allow-git = []
```

El control de CI

Sintaxis de GitHub Actions; los mismos trabajos pueden adaptarse a cualquier sistema de CI. Cada paso de Cargo que resuelve dependencias usa `--locked`, de modo que CI falla en lugar de cambiar `Cargo.lock` en silencio. cargo-deny se instala en la versión exacta con la que se verificó `deny.toml`, en lugar de tomarse de una acción con versión flotante. Desde rustup 1.28, `rustup toolchain install` sin argumentos instala la cadena de herramientas que nombra `rust-toolchain.toml`; esto se verificó con rustup 1.29.3, sin cambiar la cadena de herramientas por defecto. El trabajo `msrv` demuestra la promesa de `rust-version` en ese compilador exacto. El trabajo de semver se aplica solo a bibliotecas y compara con la etiqueta de la última versión publicada, no con `main` (capítulo nueve), en los pull requests y en la propia etiqueta de versión. Llama directamente a la CLI de cargo-semver-checks y elige la línea base por crate: la etiqueta anterior más cercana que coincide con `orders-client-v*`, excluidas las etiquetas de prelanzamiento como `orders-client-v2.0.0-rc.1` (documentación de git describe). Si no hay una etiqueta de versión anterior, el trabajo lo indica y se omite; cuando hay línea base, una verificación fallida hace fallar el trabajo. El trabajo de release se ejecuta solo para una etiqueta de versión y solo después de que se superen los trabajos de control, MSRV, semver y compilación. Es un esqueleto: complete sus propios pasos de publicación, firma y atestación. Añada lo que añada, mantenga la regla del capítulo diez: promover el artefacto que produjo el trabajo de compilación, verificado por su digest, y nunca recompilar para promover. En producción, fije cada acción por el SHA completo de su commit; aquí se usan etiquetas para que la plantilla sea legible.

```
# Verificado con Rust 1.100.0 estable, noviembre de 2026. Fije cada acción por su SHA de commit
completo.
name: ci
on:
  pull_request:
  push:
    branches: [main]
    tags: ["v*", "*-v*"]  # una release es una etiqueta de versión; las bibliotecas etiquetan por
crate, p. ej. orders-client-v1.4.2
permissions:
  contents: read

jobs:
  gate:
    runs-on: ubuntu-24.04
```

```
    steps:
      - uses: actions/checkout@v7
      - name: Instalar la cadena de herramientas que indica rust-toolchain.toml (rustup 1.28 o
posterior)
        run: rustup toolchain install
      - run: cargo fmt --all --check
      - run: cargo clippy --workspace --all-targets --locked -- -D warnings
      - run: cargo test --workspace --locked
      - name: Instalar la versión de cargo-deny con la que se verificó deny.toml
        run: cargo install cargo-deny --locked --version 0.20.2
      - name: Control de admisión (destinos y features provienen de [graph] en deny.toml)
        run: cargo deny --locked check

  msrv:     # comprueba rust-version de Cargo.toml con ese compilador exacto
    runs-on: ubuntu-24.04
    steps:
      - uses: actions/checkout@v7
      - run: rustup toolchain install 1.85.0 --profile minimal
      - run: cargo +1.85.0 check --workspace --all-targets --locked

  semver:   # solo bibliotecas: compara la API pública con la última etiqueta de release
    runs-on: ubuntu-24.04
    steps:
      - uses: actions/checkout@v7
        with: { fetch-depth: 0 }       # historial y etiquetas, para la línea base
      - run: rustup toolchain install
      - run: cargo install cargo-semver-checks --locked   # fije --version a la que verificó
      - name: Comparar con la última etiqueta de release estable de este crate
        shell: bash
        run: |
          set -euo pipefail
          CRATE=orders-client
          if ! TAG=$(git describe --tags --abbrev=0 --match "$CRATE-v*" --exclude "$CRATE-v*-*"
HEAD^ 2>/dev/null); then
            echo "No earlier stable $CRATE release tag: first release, semver check skipped."
            exit 0
          fi
          echo "Baseline: $TAG"
          cargo semver-checks --package "$CRATE" --baseline-rev "$TAG"

  build:
    needs: gate
    strategy:
      matrix:
        include:
          - { os: ubuntu-24.04, target: x86_64-unknown-linux-gnu }
          - { os: ubuntu-24.04-arm, target: aarch64-unknown-linux-gnu }
          - { os: windows-2025, target: x86_64-pc-windows-msvc }
          - { os: macos-15, target: aarch64-apple-darwin }
    runs-on: ${{ matrix.os }}
    steps:
      - uses: actions/checkout@v7
      - run: rustup toolchain install
      - run: rustup target add ${{ matrix.target }}
      - run: cargo build --release --locked --target ${{ matrix.target }}
```

```
  release:  # ESQUELETO: complete sus pasos de publicación, firma y atestación. Solo etiquetas de
versión, tras todas las comprobaciones
    if: startsWith(github.ref, 'refs/tags/v')
    needs: [gate, msrv, semver, build]   # un servicio sin API pública elimina semver
    runs-on: ubuntu-24.04
    steps:
      - run: echo "skeleton: add publish, sign and attest steps (Chapters Eight and Ten)"
```

El Dockerfile

Multietapa, con cargo-chef para que la capa de dependencias quede en caché hasta que cambien `Cargo.toml` o `Cargo.lock`. La etapa final es una imagen distroless con el entorno de ejecución de C, sin shell y sin gestor de paquetes, que se ejecuta como el usuario numérico no root 65532, de modo que Kubernetes puede verificar `runAsNonRoot` (capítulo diez). Sustituya cada etiqueta por un digest antes de producción. Se confirmó que existen las etiquetas de las imágenes base y la versión de cargo-chef. Este archivo es una plantilla revisada: la compilación de la imagen todavía no se ha ejecutado para esta edición, así que no está verificada y no se indica el tamaño de la imagen. Con los marcadores `<digest>` tal como están escritos, la imagen no se compila; sustituya cada uno por el digest que fije.

```
# syntax=docker/dockerfile:1
# Plantilla, revisada en noviembre de 2026; compilación de la imagen pendiente
# Sustituya cada <digest> por el digest que fije para esa etiqueta.
FROM rust:1.100.0-slim-bookworm@sha256:<digest> AS chef
RUN cargo install cargo-chef --locked --version 0.1.78
WORKDIR /app

FROM chef AS planner
COPY . .
RUN cargo chef prepare --recipe-path recipe.json

FROM chef AS builder
COPY --from=planner /app/recipe.json recipe.json
# Solo dependencias: en caché hasta que cambien Cargo.toml o Cargo.lock.
RUN cargo chef cook --release --locked --recipe-path recipe.json
COPY . .
RUN cargo build --release --locked --bin refservice

# Compilación con glibc, así que la imagen de ejecución es distroless cc. Para una compilación
estática con musl
# (--target x86_64-unknown-linux-musl), use gcr.io/distroless/static-debian12:nonroot
# y copie desde target/x86_64-unknown-linux-musl/release/.
FROM gcr.io/distroless/cc-debian12:nonroot@sha256:<digest> AS runtime
COPY --from=builder /app/target/release/refservice /usr/local/bin/refservice
USER 65532:65532
EXPOSE 8080
ENV PORT=8080 RUST_LOG=info
ENTRYPOINT ["/usr/local/bin/refservice"]
```

El Deployment de Kubernetes

Las solicitudes (requests) son datos de entrada para la planificación, no uso medido; fíjelas a partir de su propia medición y revíselas. El límite de memoria es el contrato con el nodo. La disponibilidad (readiness) pasa a falso ante SIGTERM solo porque el propio servicio la cambia; el servicio de referencia lo hace cuando llega su señal de apagado. La ruta de liveness lanza una tarea trivial y espera su resultado, de modo que, si el entorno de ejecución deja de planificar trabajo, la comprobación falla. El periodo de gracia debe ser más largo que la pausa de preStop más el tiempo de terminación ordenada (drain). El servicio de referencia limita a 20 segundos su terminación ordenada, dentro del periodo de gracia de 30 segundos menos la pausa de 5 segundos, y luego termina aunque todavía haya solicitudes en curso. El hook `preStop` usa la acción integrada `sleep`, porque la imagen distroless no tiene shell (estable desde Kubernetes 1.34). Este Deployment es Burstable a propósito: no hay límite de CPU (capítulo once, sobre la limitación de CPU o throttling) y el límite de memoria está por encima de la solicitud. Para la clase Guaranteed, fije límites de CPU y memoria iguales a sus solicitudes.

```
# Plantilla, revisada en noviembre de 2026 (Kubernetes 1.34 o posterior para la acción sleep)
apiVersion: apps/v1
kind: Deployment
metadata:
  name: refservice
  labels: { app.kubernetes.io/name: refservice }
spec:
  replicas: 2
  selector:
    matchLabels: { app.kubernetes.io/name: refservice }
  template:
    metadata:
      labels: { app.kubernetes.io/name: refservice }
    spec:
      terminationGracePeriodSeconds: 30
      automountServiceAccountToken: false
      securityContext:
        runAsNonRoot: true
        seccompProfile: { type: RuntimeDefault }
      containers:
        - name: refservice
          image: registry.example.com/refservice@sha256:<digest>
          ports: [{ name: http, containerPort: 8080 }]
          env: [{ name: RUST_LOG, value: info }]
          resources:
            requests: { cpu: 50m, memory: 32Mi }   # sustitúyalos por valores medidos
            limits:   { memory: 64Mi }               # Burstable a propósito; véase más arriba
          lifecycle:
            preStop:
              sleep: { seconds: 5 }   # los endpoints se actualizan mientras aún llega tráfico
          startupProbe:
            httpGet: { path: /healthz, port: http }
```

```
            periodSeconds: 2
            failureThreshold: 15
          livenessProbe:
            httpGet: { path: /healthz, port: http }
            periodSeconds: 10
            failureThreshold: 3
          readinessProbe:
            httpGet: { path: /readyz, port: http }
            periodSeconds: 5
            failureThreshold: 2
          securityContext:
            allowPrivilegeEscalation: false
            readOnlyRootFilesystem: true
            capabilities: { drop: ["ALL"] }
```

El registro de decisión (ADR)

Un registro por cada decisión de lenguaje o plataforma. La sección de renovación es lo que lo mantiene como contrato y no como simple memorando (capítulo dos).

```
# Plantilla, revisada en noviembre de 2026
# ADR-NNN: <título de la decisión>
Estado:      propuesto | aceptado | renovado | ajustado | revertido
Fecha:       YYYY-MM-DD
Responsable: <un rol, no el nombre de una persona>
Aprobadores: <comité de arquitectura o equivalente>

## Contexto
El resultado de negocio, la carga de trabajo y la restricción que obliga a elegir.

## Criterios (pesos acordados antes de puntuar)
| Criterio                  | Peso   | Rust | Alternativa A | Alternativa B |
| Consumo de recursos       |   <w>  |      |               |               |
| Habilidades y contratación |   <w>  |      |               |               |
| Encaje en el ecosistema   |   <w>  |      |               |               |
| Costo de compilación y CI  |   <w>  |      |               |               |

## Alternativas consideradas
Un párrafo equilibrado por alternativa: dónde gana y dónde pierde para esta carga de trabajo.

## Decisión
Qué se eligió, con qué alcance y qué queda explícitamente fuera de alcance.

## Consecuencias
Costos aceptados (tiempo de compilación, formación, contratación) y contratos creados.

## Contrato de renovación
Resultados esperados:      <medibles>
Línea base:                <valor y cómo se midió>
Fecha de revisión:         YYYY-MM-DD
Disparadores de reversión: <condiciones que reabren esta decisión>
Ruta de salida:            <cómo saldría de ella el entorno tecnológico y a qué costo>
```

Una entrada del registro de unsafe

Una entrada por cada punto de uso de `unsafe` del que es responsable la organización. El invariante es la promesa que el compilador no puede verificar; la evidencia es cómo la verifica el equipo en su lugar.

```
# Plantilla, revisada en noviembre de 2026
[[unsafe]]
id = "UR-0007"
crate = "codec"
location = "crates/codec/src/simd.rs: decode_block"
owner = "codec team lead"
reason = "SIMD decode; the safe version missed the latency budget in profiling"
invariant = "input length is a multiple of 16 and the pointer is 16-byte aligned"
safety_comment = true            # un comentario // SAFETY: declara el invariante
reviewed_by = "platform security"
reviewed_on = "2026-11-01"
evidence = ["cargo miri test -p codec", "fuzz target decode_block, scheduled run"]
next_review = "2027-05-01"
```

Un registro de excepción

Toda excepción sigue un mismo ciclo de vida: solicitud, responsable, autoridad de excepción, evidencia, vencimiento, revisión de degradación de los controles (capítulo cinco). El mismo registro cubre código unsafe, FFI, scripts de compilación, excepciones de lint, avisos ignorados y cambios de emergencia en la cadena de herramientas.

```
# Plantilla, revisada en noviembre de 2026
[[exception]]
id = "EX-0042"
kind = "advisory-ignore"    # unsafe | ffi | build-script | lint-waiver | advisory-ignore |
toolchain
subject = "RUSTSEC-YYYY-NNNN in crate <name> <version>"
requested_by = "payments team"
owner = "payments team lead"
approved_by = "platform security"     # autoridad de excepción; nunca quien la solicita
reason = "vulnerable function is not reachable from our code; no fixed release yet"
compensating_control = "input size limited at the gateway"
evidence = ["call graph review 2026-11-10", "cargo deny check output attached"]
created = "2026-11-12"
expires = "2027-02-12"                # sin vencimiento, no hay aprobación
on_expiry = "gate fails again; renew with fresh evidence or remove"
```

El servicio de referencia

Qué es. Un pequeño servicio HTTP escrito para este libro: `refservice`, unas 130 líneas de Rust en la edición 2024. Usa axum 0.8 sobre tokio 1 (entorno de ejecución multihilo), emite registros como líneas JSON mediante `tracing` y expone tres rutas: `GET /healthz`

(liveness), `GET /readyz` (readiness, que pasa a falso cuando empieza el apagado) y `GET /hello` (un cuerpo de texto fijo). Se apaga de forma ordenada con Ctrl-C y, en Unix, con SIGTERM: readiness pasa a falso, las solicitudes en curso disponen de hasta 20 segundos para completarse y el proceso termina. Las trazas y las métricas se omiten a propósito: el servicio existe para medir el consumo de recursos, y un servicio real las añade (capítulo once). Está construido con las plantillas de servicio anteriores (su propio crate restringe `unsafe_code` a `forbid`) y, en Rust 1.100.0, pasa `cargo fmt --check`, `cargo clippy --all-targets -- -D warnings` y `cargo deny check` con cargo-deny 0.20.2 y el deny.toml anterior (avisos, prohibiciones, licencias y orígenes, todo correcto; la única advertencia es que ninguna dependencia usa la licencia ISC permitida). No hace trabajo real, así que sus cifras son un mínimo para un servicio de este tipo, no una previsión para el suyo.

Cómo reconstruirlo a partir de este libro. Cree una carpeta con los cuatro archivos siguientes más `rust-toolchain.toml` y `deny.toml` exactamente como se mostraron antes en este apéndice. `Cargo.lock` se genera en la primera compilación; inclúyalo y versiónelo en el repositorio, y compile con `--locked` a partir de entonces. Un lockfile nuevo puede resolver versiones de parche más recientes que las que se indican en el método, así que registre las suyas junto a sus cifras.

```
# Cargo.toml (completo)
[package]
name = "refservice"
version = "0.1.0"
edition = "2024"
rust-version = "1.85"
publish = false
license = "MIT OR Apache-2.0"

[dependencies]
axum = "0.8"
tokio = { version = "1", features = ["macros", "rt-multi-thread", "net", "signal", "sync",
"time"] }
tracing = "0.1"
tracing-subscriber = { version = "0.3", features = ["env-filter", "fmt", "json"] }

[lints.rust]
unsafe_code = "forbid"
missing_debug_implementations = "warn"

[lints.clippy]
all = { level = "warn", priority = -1 }
unwrap_used = "warn"
expect_used = "warn"

[profile.release]
lto = true
```

```
codegen-units = 1
strip = true
panic = "abort"
overflow-checks = true
```

```
// src/main.rs (completo)
//! refservice: el servicio de referencia de "Gobernar Rust por contrato".
//! GET /healthz (liveness), GET /readyz (readiness), GET /hello.
//! Apagado ordenado con Ctrl-C y, en Unix, con SIGTERM, con un plazo máximo para completar las
solicitudes en curso.
//! Los registros son líneas JSON. Se omiten trazas y métricas: este ejemplo existe para
//! medir el consumo de recursos, y un servicio real las añade (capítulo once).

use std::{
    net::SocketAddr,
    sync::{
        Arc,
        atomic::{AtomicBool, Ordering},
    },
    time::Duration,
};

use axum::{Router, extract::State, http::StatusCode, routing::get};
use tracing::{info, warn};
use tracing_subscriber::EnvFilter;

/// Menor que terminationGracePeriodSeconds (30) menos la pausa de preStop (5).
const DRAIN_DEADLINE: Duration = Duration::from_secs(20);

#[derive(Clone, Debug)]
struct AppState {
    ready: Arc<AtomicBool>,
}

async fn healthz() -> (StatusCode, &'static str) {
    // La comprobación de liveness ejercita el entorno de ejecución asíncrono: lanza y espera una
tarea trivial,
    // de modo que un entorno de ejecución que ya no planifica trabajo hace fallar la probe.
    match tokio::spawn(async {}).await {
        Ok(()) => (StatusCode::OK, "ok"),
        Err(_) => (StatusCode::SERVICE_UNAVAILABLE, "runtime error"),
    }
}

async fn readyz(State(state): State<AppState>) -> (StatusCode, &'static str) {
    if state.ready.load(Ordering::Acquire) {
        (StatusCode::OK, "ready")
    } else {
        (StatusCode::SERVICE_UNAVAILABLE, "not ready")
    }
}

async fn hello() -> &'static str {
    "hello from refservice\n"
```

```
}

fn app(state: AppState) -> Router {
    Router::new()
        .route("/healthz", get(healthz))
        .route("/readyz", get(readyz))
        .route("/hello", get(hello))
        .with_state(state)
}

async fn shutdown_signal(state: AppState) {
    let ctrl_c = async {
        if let Err(e) = tokio::signal::ctrl_c().await {
            tracing::error!(error = %e, "failed to listen for Ctrl-C");
        }
    };

    #[cfg(unix)]
    let terminate = async {
        match tokio::signal::unix::signal(tokio::signal::unix::SignalKind::terminate()) {
            Ok(mut s) => {
                s.recv().await;
            }
            Err(e) => tracing::error!(error = %e, "failed to listen for SIGTERM"),
        }
    };

    #[cfg(not(unix))]
    let terminate = std::future::pending::<()>();

    tokio::select! {
        () = ctrl_c => {},
        () = terminate => {},
    }

    // Primero deja de anunciar disponibilidad y luego deja que terminen las solicitudes en
curso.
    state.ready.store(false, Ordering::Release);
    info!("shutdown signal received, draining");
}

#[tokio::main]
async fn main() -> Result<(), Box<dyn std::error::Error>> {
    tracing_subscriber::fmt()
        .json()
        .with_env_filter(
            EnvFilter::try_from_default_env().unwrap_or_else(|_| EnvFilter::new("info")),
        )
        .init();

    let port: u16 = std::env::var("PORT")
        .ok()
        .and_then(|p| p.parse().ok())
        .unwrap_or(8080);
    let addr = SocketAddr::from(([0, 0, 0, 0], port));
```

```
    let state = AppState {
        ready: Arc::new(AtomicBool::new(false)),
    };
    let listener = tokio::net::TcpListener::bind(addr).await?;
    state.ready.store(true, Ordering::Release);
    info!(%addr, version = env!("CARGO_PKG_VERSION"), "refservice listening");

    let (drain_tx, drain_rx) = tokio::sync::oneshot::channel::<()>();
    let server = axum::serve(listener, app(state.clone())).with_graceful_shutdown(async move {
        shutdown_signal(state).await;
        let _ = drain_tx.send(());
    });
    // La espera está acotada: una vez iniciado el apagado, deja de esperar tras DRAIN_DEADLINE.
    let deadline = async {
        if drain_rx.await.is_ok() {
            tokio::time::sleep(DRAIN_DEADLINE).await;
        } else {
            std::future::pending::<()>().await;
        }
    };
    tokio::select! {
        res = server => res?,
        () = deadline => warn!("drain deadline passed, exiting with requests in flight"),
    }

    info!("refservice stopped");
    Ok(())
}
```

```
# measure/client.py: el cliente de solicitudes usado en el paso 4
"""Envía N solicitudes GET /hello secuenciales por una conexión keep-alive; comprueba cada código
de estado."""
import http.client
import sys

n = int(sys.argv[1]) if len(sys.argv) > 1 else 10000
port = int(sys.argv[2]) if len(sys.argv) > 2 else 8080
conn = http.client.HTTPConnection("127.0.0.1", port, timeout=5)  # una solicitud bloqueada hace
fallar la ejecución
ok = 0
for _ in range(n):
    conn.request("GET", "/hello")
    resp = conn.getresponse()
    resp.read()
    ok += resp.status == 200
conn.close()
print(f"{ok}/{n} returned 200")
sys.exit(0 if ok == n else 1)
```

```
# measure/measure.ps1: pasos 2 a 5 del método que sigue
# Medición rápida del consumo de recursos de refservice (Apéndice A). Windows, PowerShell 5.1 o
7.
# Ejecútelo desde la carpeta refservice después de: cargo +1.100.0 build --release --locked
```

```
# Una ejecución que falla (puerto ocupado, servicio no disponible, proceso terminado, error del
cliente) detiene el script.
$ErrorActionPreference = "Stop"; $ProgressPreference = "SilentlyContinue"
$exe = ".\target\release\refservice.exe"; $port = 8080
if (Get-NetTCPConnection -LocalPort $port -State Listen -ErrorAction SilentlyContinue) { throw
"port $port is already in use" }
"Binary size: {0} bytes" -f (Get-Item $exe).Length
function Wait-Ready($p, [double]$deadlineSec = 5) {
    $clock = [System.Diagnostics.Stopwatch]::StartNew()   # monotónico, tiempo total
    while ($clock.Elapsed.TotalSeconds -lt $deadlineSec) {
        if ($p.HasExited) { throw "refservice exited with code $($p.ExitCode) before it was
ready" }
        try {
            $r = Invoke-WebRequest -UseBasicParsing -TimeoutSec 1 "http://127.0.0.1:$port/readyz"
            if ($r.StatusCode -eq 200) { return }
        } catch { }
        Start-Sleep -Milliseconds 100
    }
    throw "refservice was not ready within $deadlineSec seconds"
}
foreach ($run in 1..5) {
    $env:RUST_LOG = "warn"; $env:PORT = "$port"
    $p = Start-Process -FilePath $exe -PassThru -WindowStyle Hidden
    try {
        Wait-Ready $p
        Start-Sleep -Seconds 3                      # periodo en reposo, medido desde la
disponibilidad
        $p.Refresh()
        $idleWs = [math]::Round($p.WorkingSet64 / 1KB); $idlePb =
[math]::Round($p.PrivateMemorySize64 / 1KB)
        $threads = $p.Threads.Count
        python .\measure\client.py 10000 $port
        if ($LASTEXITCODE -ne 0) { throw "client failed in run $run (exit $LASTEXITCODE)" }
        Start-Sleep -Seconds 2
        if ($p.HasExited) { throw "refservice exited during run $run" }
        $p.Refresh()
        $useWs = [math]::Round($p.WorkingSet64 / 1KB); $usePb =
[math]::Round($p.PrivateMemorySize64 / 1KB)
        "run {0}: idle WS {1} KB, PB {2} KB; after WS {3} KB, PB {4} KB; threads {5}" -f $run,
$idleWs, $idlePb, $useWs, $usePb, $threads
    } finally {
        if (-not $p.HasExited) { Stop-Process -Id $p.Id -Force }
        $p.WaitForExit()
    }
}
```

El método.

Esta es una medición rápida (smoke test) del consumo de recursos en Windows, no un benchmark de producción. Solo mide el tamaño del binario y la memoria. No mide la latencia: un cliente secuencial que espera cada respuesta es el patrón de bucle cerrado contra el que advierte el capítulo once, así que mida la latencia de su propio servicio con el método del capítulo once.

1. Compilación: `cargo +1.100.0 build --release --locked` con el perfil de release anterior, en Windows 11, destino `x86_64-pc-windows-msvc`, con las dependencias resueltas por el lockfile versionado en el repositorio a axum 0.8.9, tokio 1.54.1, tracing 0.1.44, tracing-subscriber 0.3.23 (con su característica `json`).
2. Tamaño del binario: el tamaño del archivo `refservice.exe` según lo indica el sistema de archivos.
3. Memoria en reposo: iniciar el binario con `RUST_LOG=warn`, consultar `/readyz` hasta que devuelva 200 (el script falla si el puerto está ocupado, si el proceso termina o si no se alcanza la disponibilidad dentro de un plazo total de 5 segundos, medido con un reloj monotónico; una consulta que siga en curso al vencer el plazo puede añadir hasta 1 segundo, su tiempo de espera), esperar 3 segundos y leer el conjunto de trabajo (working set) y los bytes privados del proceso desde el sistema operativo (PowerShell `Get-Process`: `WorkingSet64`, `PrivateMemorySize64`).
4. Memoria después del uso: un cliente de Python en la misma máquina envía 10.000 solicitudes secuenciales `GET /hello` por una sola conexión keep-alive a 127.0.0.1 y verifica cada código de estado. Esperar 2 segundos y volver a leer la memoria. Las solicitudes solo sirven para ejercitar el proceso; no se cronometran.
5. Repetir cada vez desde un proceso nuevo, cinco ejecuciones en total, e informar el rango. El script siempre detiene el proceso, se interrumpe en la primera ejecución fallida (un error del cliente o un proceso que terminó) y solo se informan las ejecuciones completas. El script anterior hace los pasos 2 a 5: `powershell -File measure\measure.ps1` desde la carpeta del servicio.

Hardware. Una CPU de escritorio x86_64 de 16 núcleos (32 hilos de hardware) con 64 GB de RAM, Windows 11, medido el 15 de noviembre de 2026. Por defecto, el entorno de ejecución multihilo de Tokio inicia un hilo de trabajo (worker) por cada unidad de paralelismo disponible para el proceso, según indica `std::thread::available_parallelism` (documentación de Tokio); aquí fueron 32, y el proceso tuvo 36 hilos en todas las ejecuciones. Un contenedor con límite de CPU, o con un número explícito de hilos de trabajo, tendría menos.

Medición	Resultado	Notas
Tamaño del binario de release (Windows x86_64)	1.315.840 bytes (unos 1,3 MB)	Un solo ejecutable sin recolector de basura ni máquina virtual que instalar; aun así enlaza el entorno de ejecución de C del sistema
Conjunto de trabajo en reposo, 3 segundos después de la disponibilidad	6,0 MB (6.080 a 6.116 KB)	Cinco ejecuciones; bytes privados 2,5 MB (2.540 a 2.584 KB)
Conjunto de trabajo después de 10.000 solicitudes secuenciales	6,1 MB (6.224 a 6.256 KB)	Cinco ejecuciones; bytes privados 2,7 MB (2.748 a 2.788 KB); todas las solicitudes devolvieron 200

La imagen de contenedor no se compiló para esta edición, así que no se informa su tamaño.

> **Interprete estas cifras en sentido estricto.** Describen un único servicio trivial en una sola máquina, con el cliente en el mismo host. No son una comparación con ningún otro lenguaje ni un plan de capacidad. Use el método, no las cifras: construya su propio servicio con sus propias versiones fijadas y su propio perfil, mídalo de la misma forma bajo su propia carga y anote el resultado en el registro de decisión como línea base.

No medido. Latencia de cualquier tipo, el tamaño de la imagen de contenedor, la memoria residente en Linux dentro de un contenedor y en un pod de Kubernetes, el uso de CPU, el comportamiento bajo carga concurrente y el tiempo de arranque. Son las mediciones que un equipo debería añadir para su propio servicio antes de fijar solicitudes y límites.

APÉNDICE B

Los tres niveles en otros lenguajes

Los tres niveles no son una idea propia de Rust. Todo lenguaje reparte sus garantías entre lo que demuestran el compilador o el entorno de ejecución, lo que las herramientas deben hacer cumplir y lo que todavía corresponde a operaciones y a las personas. Lo único que cambia es dónde cae la línea. Este apéndice traza esa línea para TypeScript, Go, C# sobre .NET y Java, de modo que un equipo que trabaje con más de un lenguaje pueda escribir un solo conjunto de contratos y completar el punto de aplicación para cada lenguaje.

Lea las tablas como un mapa, no como una clasificación. Un recolector de basura sitúa la seguridad de memoria en el nivel 1 en los cuatro lenguajes, y esa es una fortaleza real: el recolector demuestra algo que Rust demuestra con la propiedad (ownership). Lo que difiere es el resto: si el valor nulo forma parte del sistema de tipos, si las carreras de datos se previenen o solo se detectan y cuánto de la cadena de suministro revisan las herramientas por defecto. Cada lenguaje tiene además una vía de escape, como Rust tiene `unsafe`, y a ella se aplica el mismo contrato de excepción.

Nivel 1: lo que demuestran el compilador o el entorno de ejecución

Lenguaje	Demostrado por defecto	No demostrado (y la vía de escape)
Rust (como referencia)	Seguridad de memoria y ausencia de carreras de datos en código seguro; tipos; `match` exhaustivo; no existe el nulo (`Option`)	Interbloqueos, carreras lógicas, fugas, pánicos, desbordamiento en compilaciones de release. Vía de escape: `unsafe`, FFI, scripts de compilación
TypeScript	Con `strict` activado, verificación de tipos en tiempo de compilación, incluidas las comprobaciones de null y undefined (`strictNullChecks`). El sistema de tipos no es sólido (sound), y esto es deliberado: el manual de TypeScript enumera los casos en que acepta código cuya seguridad no puede demostrar, así que la verificación estricta es evidencia fuerte, no una demostración. El entorno de ejecución de JavaScript es seguro en memoria gracias a la recolección de basura	Los tipos se eliminan al compilar: en tiempo de ejecución no se comprueba ningún tipo, así que los datos que llegan de la red no ofrecen ninguna garantía hasta que se validan. Vías de escape: `any`, aserciones de tipo, aserciones de no nulo (`!`), `@ts-ignore`
Go	Tipos estáticos; seguridad de memoria mediante recolección de basura y comprobaciones de límites en tiempo de ejecución; las importaciones y variables sin usar son errores de compilación	No hay seguridad frente a nulos (desreferenciar nil provoca un pánico en tiempo de ejecución). Las carreras de datos no se previenen, y el modelo de memoria de Go advierte que las carreras sobre valores de varias palabras de máquina pueden corromper la memoria. Vías de escape: `unsafe`, cgo
C# / .NET	Seguridad de tipos y de memoria en código administrado mediante el CLR y la recolección de basura; comprobaciones de límites. Los tipos de referencia anulables añaden análisis del estado nulo	El análisis de anulabilidad produce advertencias, no errores, salvo que el proyecto las trate como errores; las carreras de datos no se previenen. Vías de escape: bloques `unsafe` (requieren `AllowUnsafeBlocks`), P/Invoke
Java	Verificación de tipos de `javac`, el verificador de bytecode de la JVM, seguridad de memoria mediante recolección de basura, comprobaciones de límites	El nulo no forma parte del sistema de tipos (`NullPointerException` en tiempo de ejecución); las carreras de datos no se previenen, aunque el Java Memory Model define su comportamiento. Vías de escape: JNI, la Foreign Function and Memory API (definitiva desde Java 22, con métodos restringidos), `sun.misc.Unsafe`

Nivel 2: lo que las herramientas deben hacer cumplir

Lenguaje	Control del código	Control de la cadena de suministro	La fijación de versiones
Rust	rustfmt, Clippy con `-D warnings`, tabla `[lints]`, pruebas, Miri	cargo-deny (avisos, licencias, prohibiciones, orígenes), cargo-audit, cargo-vet, cargo-semver-checks	`Cargo.lock`, `rust-toolchain.toml`, `rust-version`
TypeScript	`tsc --noEmit` con `strict` (más opciones como `noUncheckedIndexedAccess`); ESLint con typescript-eslint; validación de esquemas en tiempo de ejecución en cada puerta	`npm audit`; revisión de los scripts de instalación; verificación de procedencia (provenance) cuando el registro la ofrezca	`package-lock.json` incluido y versionado en el repositorio e instalado con `npm ci`; la versión de Node.js fijada (por ejemplo en `engines` y en la imagen de CI)
Go	`gofmt`, `go vet`, el detector de carreras (`go test -race`, que detecta las carreras que las pruebas llegan a ejecutar, no todas), staticcheck	`govulncheck`, que solo informa de vulnerabilidades en código que el módulo puede alcanzar; la base de datos de sumas de verificación (sum.golang.org)	`go.mod` y `go.sum`; las líneas `go` y `toolchain` de `go.mod`
C# / .NET	`<Nullable>enable</Nullable>` con `TreatWarningsAsErrors`; analizadores de código de .NET (`AnalysisLevel`, `EnforceCodeStyleInBuild`)	Auditoría de NuGet durante la restauración (activa por defecto desde el SDK de .NET 8; emite las advertencias NU1901 a NU1904, así que el control debe convertirlas en errores; los paquetes transitivos se auditan por defecto en proyectos cuyo destino es .NET 10); asignación de orígenes de paquetes	`packages.lock.json` con restauración en modo bloqueado; `global.json` fija el SDK; Central Package Management
Java	Error Prone (comprobaciones en tiempo de compilación como plugin de `javac`); SpotBugs (análisis de bytecode); un verificador de nulidad como NullAway	Verificación de dependencias de Gradle (`verification-metadata.xml`, sumas de verificación y firmas); OWASP Dependency-Check; reglas de Maven Enforcer	Bloqueo de dependencias de Gradle; en Maven, versiones explícitas y una sección de gestión de dependencias (Maven no tiene lockfile por defecto); un JDK fijado mediante toolchains

Nivel 3: lo que corresponde a operaciones y a las personas

Lenguaje	Ajustes de ejecución que se convierten en contratos	Responsabilidad de las personas en todos los lenguajes
Rust	Elección del asignador de memoria, entorno de ejecución asíncrono e hilos de trabajo, estrategia de `panic`, límite de memoria del contenedor	Lógica de negocio, interbloqueos, tiempos de espera y reintentos, capacidad, el registro de decisión y su renovación, el registro de excepciones
TypeScript	Tamaño del heap de Node.js (`--max-old-space-size`), latencia del bucle de eventos, la rama de versiones de Node.js y su ventana de soporte	
Go	`GOMEMLIMIT` y `GOGC`; `GOMAXPROCS` (ajustado por defecto a los límites del contenedor desde Go 1.25)	
C# / .NET	Modo del GC (servidor o estación de trabajo), límite estricto del heap, advertencias de NativeAOT y de recorte tratadas como impedimentos para publicar una versión	
Java	Tamaño del heap (`-Xmx` o `-XX:MaxRAMPercentage`), elección del recolector (G1 por defecto, ZGC para pausas breves), imagen nativa de GraalVM cuando importan el tiempo de arranque y el consumo de recursos	

CÓMO USAR ESTE APÉNDICE

- Escriba el contrato una sola vez, en términos independientes del lenguaje; complete el punto de aplicación para cada lenguaje.
- Cuando una garantía es de nivel 1 en un lenguaje y de nivel 2 en otro (nulos, carreras de datos), el control de nivel 2 debe bloquear, no advertir.
- Toda vía de escape (`unsafe`, `any`, cgo, JNI, P/Invoke) pasa por el mismo ciclo de vida de excepciones.
- Vuelva a revisar esta tabla cuando un lenguaje publique una versión principal; la línea se desplaza.

APÉNDICE C

Glosario y registro de fuentes

Un contrato solo funciona cuando todos interpretan sus términos de la misma manera. Este apéndice fija el significado de los términos que usa el libro y luego enumera las fuentes públicas que respaldan sus afirmaciones, con editor, año y dirección, para que el lector pueda verificar cada una.

Glosario de términos de contrato

Término	Significado en este libro
Contrato	Una regla escrita en un límite, con un responsable, un punto de aplicación, un comportamiento ante fallos y evidencia. Una regla sin estos cuatro elementos es solo un deseo.
Puerta	Cualquier lugar por el que algo entra en el sistema: el código en el repositorio, una dependencia en la compilación, un artefacto en un registro, una carga de trabajo en un clúster.
Tres niveles	La división de cada garantía en lo que el compilador demuestra (nivel 1), lo que las herramientas deben hacer cumplir (nivel 2) y lo que corresponde a operaciones y a las personas (nivel 3). El eje de todos los capítulos.
El desarrollador propone, el compilador decide	El principio de GDCD "el modelo propone, el contrato decide" aplicado solo al nivel 1. Es válido dentro del límite de confianza y en ningún otro lugar.
Límite de confianza	El conjunto de código de cuya corrección dependen las garantías de Rust seguro sin verificarla: el compilador, cada bloque `unsafe` y la abstracción que lo rodea, las dependencias, FFI, los scripts de compilación, las macros procedurales, el asignador global y el entorno de ejecución asíncrono.
Ficha de contrato	La tabla de una página que cierra los capítulos cuatro a once: contrato, amenaza que atiende, responsable, punto de aplicación, comportamiento ante fallos, autoridad de excepción y vencimiento, evidencia de que el control se ejecutó, evidencia de que el resultado mejoró, frecuencia de revisión.
Punto de aplicación	El lugar exacto donde se verifica un contrato: una opción del compilador, un trabajo de CI, un controlador de admisión, una sonda. Si no puede nombrarlo, el contrato no se está haciendo cumplir.
Comportamiento ante fallos	Lo que ocurre cuando la verificación falla: bloquear, advertir, avisar al personal de guardia, revertir. Se declara de antemano, no se decide durante el incidente.
Evidencia de que el control se ejecutó	Prueba de que la verificación se ejecutó (el registro de un trabajo de CI, una atestación). Distinta de la evidencia de que el resultado mejoró, y más débil.

Término	Significado en este libro
Evidencia de que el resultado mejoró	Prueba de que el resultado de negocio para el que existe el contrato realmente cambió (menos avisos de seguridad en producción, menos incidentes de un tipo determinado).
La fijación	Todo lo que fija de qué está hecha una compilación: la cadena de herramientas en `rust-toolchain.toml`, las versiones de las dependencias en `Cargo.lock`, la versión mínima de Rust admitida en `rust-version`, las imágenes base fijadas por digest.
Control de admisión	La verificación de nivel 2 que decide qué puede entrar en la compilación o en el clúster: avisos, licencias, crates prohibidos, orígenes permitidos, auditorías, firmas. Tiene un comportamiento ante fallos y una vía de excepción.
Promesa de interfaz	La API pública de un crate o servicio tratada como contrato: sus versiones siguen el versionado semántico y se verifica con una comparación de API durante la revisión.
Excepción	Cualquier paso deliberado fuera de un contrato por defecto: un bloque `unsafe`, una llamada FFI, un script de compilación, una excepción a un lint, un aviso ignorado, un cambio de emergencia en la cadena de herramientas.
Ciclo de vida de excepciones	La única vía que sigue toda excepción: solicitud, responsable con nombre, autoridad de excepción, invariante y evidencia declaradas, fecha de vencimiento, revisión de degradación de los controles. Una excepción sin vencimiento es un nuevo valor por defecto que nadie aprobó.
Registro de unsafe	La lista de todos los puntos de uso de `unsafe` de los que es responsable la organización, con responsable, invariante, revisión y evidencia de pruebas (por ejemplo, Miri).
Autoridad de excepción	El rol autorizado para aprobar una excepción. Nunca la persona que la solicita.
Revisión de degradación de los controles	La verificación programada de que una excepción, un contrato o una decisión siguen vigentes. Los elementos vencidos se renuevan con evidencia nueva o se eliminan.
Registro de decisión	Un registro de decisión de arquitectura (ADR) para la elección de lenguaje o herramienta: contexto, criterios, alternativas evaluadas con imparcialidad, decisión, consecuencias.
Contrato de renovación	La parte del registro de decisión que lo mantiene vivo: resultados esperados, línea base, fecha de revisión, disparadores de reversión y plan de salida.
Disparador de reversión	Una condición medible, acordada de antemano, que reabre la decisión.
Gobernanza mínima viable	El conjunto inicial de contratos para un equipo sin equipo de plataforma: la fijación, un control de CI (fmt, Clippy, pruebas, cargo-deny), una política de unsafe y un registro de decisión.

Término	Significado en este libro
Servicio de referencia	El pequeño servicio HTTP construido para este libro (Apéndice A). La única fuente de las cifras de consumo de recursos del libro, medidas con un método declarado.
Entorno tecnológico	Todo el código, los pipelines y las cargas de trabajo en ejecución de los que es responsable una organización, gobernados como un solo sistema.

Registro de fuentes

El 15 de noviembre de 2026 se comprobó que todas las direcciones estaban accesibles. Las estadísticas de seguridad de memoria de estas fuentes describen bases de código en C y C++ y se citan solo en ese ámbito.

Fuente	Editor	Año	Alcance y dirección
A proactive approach to more secure code	Microsoft Security Response Center	2019	Cerca del 70% de los CVE que Microsoft asigna cada año son problemas de seguridad de memoria (código C y C++). https://www.microsoft.com/en-us/msrc/blog/2019/07/a-proactive-approach-to-more-secure-code
Memory safety	The Chromium Projects	2020, actualizado	Cerca del 70% de los errores de seguridad de alta gravedad de Chromium son problemas de seguridad de memoria (código C++). https://www.chromium.org/Home/chromium-security/memory-safety/
Eliminating Memory Safety Vulnerabilities at the Source	Google Security Blog	2024	Android: la proporción de vulnerabilidades de seguridad de memoria cayó a medida que el código nuevo pasó a lenguajes seguros en memoria. https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html
Software Memory Safety (Cybersecurity Information Sheet)	National Security Agency	2022, revisado en 2023	Recomendación de preferir lenguajes seguros en memoria. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF

Fuente	Editor	Año	Alcance y dirección
The Case for Memory Safe Roadmaps	CISA, NSA y socios internacionales	2023	Pide a los fabricantes que publiquen hojas de ruta hacia la seguridad de memoria. https://www.cisa.gov/resources-tools/resources/case-memory-safe-roadmaps
Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development	CISA y NSA	2025	Hoja informativa conjunta sobre la adopción de lenguajes seguros en memoria. https://www.cisa.gov/resources-tools/resources/memory-safe-languages-reducing-vulnerabilities-modern-software-development
Back to the Building Blocks: A Path Toward Secure and Measurable Software	White House Office of the National Cyber Director	2024	Argumento de política pública a favor de los lenguajes seguros en memoria. https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf
The Rust Programming Language	The Rust Project	Edición actual	Propiedad, préstamo, manejo de errores, concurrencia. https://doc.rust-lang.org/book/
The Rust Reference	The Rust Project	Actual	Definición del lenguaje, incluido el comportamiento considerado indefinido. https://doc.rust-lang.org/reference/
The Rustonomicon	The Rust Project	Actual	Escritura y revisión de código `unsafe`. https://doc.rust-lang.org/nomicon/
The Rust Edition Guide: Rust 2024	The Rust Project	2025	La edición 2024, estable desde Rust 1.85. https://doc.rust-lang.org/edition-guide/rust-2024/index.html
The Cargo Book	The Rust Project	Actual	Manifiestos, lockfiles, workspaces, perfiles, `[lints]`. https://doc.rust-lang.org/cargo/
The rustup book	The Rust Project	Actual	Cadenas de herramientas (toolchains) y su fijación por proyecto con `rust-toolchain.toml`. https://rust-lang.github.io/rustup/
RustSec Advisory Database	Rust Secure Code Working Group	Actual	Avisos de seguridad para crates; cargo-audit. https://rustsec.org/
RUSTSEC-2025-0052: async-std has been discontinued	RustSec Advisory Database	2025	El aviso citado en el capítulo siete. https://rustsec.org/advisories/RUSTSEC-2025-0052.html

Fuente	Editor	Año	Alcance y dirección
cargo-deny	Embark Studios	Actual	Verificaciones de avisos, licencias, prohibiciones y orígenes, claves de configuración y códigos de diagnóstico. https://embarkstudios.github.io/cargo-deny/
cargo-vet	Mozilla	Actual	Registro y uso compartido de auditorías de dependencias. https://mozilla.github.io/cargo-vet/
cargo-semver-checks	Mantenedores del proyecto (código abierto)	Actual	Comparación de API según las reglas del versionado semántico. https://github.com/obi1kenobi/cargo-semver-checks
cargo-chef	Mantenedores del proyecto (código abierto)	Actual	Caché de capas de Docker para las dependencias de Rust. https://github.com/LukeMathWalker/cargo-chef
Distroless container images	Google (GoogleContainerTools)	Actual	Imágenes base mínimas sin shell ni gestor de paquetes. https://github.com/GoogleContainerTools/distroless
Resource Management for Pods and Containers	The Kubernetes Authors	Actual	Solicitudes, límites y cómo los usa el planificador. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
Configure Liveness, Readiness and Startup Probes	The Kubernetes Authors	Actual	Semántica de las sondas. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
Node-pressure Eviction	The Kubernetes Authors	Actual	Cómo ordena el kubelet los Pods para el desalojo: uso por encima de las solicitudes, prioridad, uso en relación con las solicitudes. https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/
Container Lifecycle Hooks; feature gate PodLifecycleSleepAction	The Kubernetes Authors	Actual	La acción de hook `sleep`: beta y activa por defecto en 1.30, estable en 1.34. https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/ y https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/

Fuente	Editor	Año	Alcance y dirección
Pod Lifecycle	The Kubernetes Authors	Actual	Terminación, SIGTERM y el periodo de gracia. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/
OpenTelemetry documentation	OpenTelemetry (CNCF)	Actual	Trazas, métricas, registros (logs) y sus exportadores. https://opentelemetry.io/docs/
OpenTelemetry Rust: status	OpenTelemetry (CNCF)	2026	Estado de las señales para Rust (capítulo once): trazas en beta, métricas y registros estables según la comprobación de noviembre de 2026. https://opentelemetry.io/docs/languages/rust/
Documenting Architecture Decisions	Michael Nygard, Cognitect	2011	El formato ADR original. https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
The Go Memory Model	The Go Authors	Actual	Las carreras de datos y sus efectos (Apéndice B). https://go.dev/ref/mem
Go 1.25 Release Notes	The Go Authors	2025	`GOMAXPROCS` ajustado a los límites del contenedor (Apéndice B). https://go.dev/doc/go1.25
govulncheck and Go vulnerability management	The Go Authors	Actual	Verificación de vulnerabilidades basada en la alcanzabilidad. https://go.dev/doc/security/vuln/
TSConfig reference: strict	Microsoft (TypeScript)	Actual	Opciones de verificación estricta de tipos. https://www.typescriptlang.org/tsconfig/#strict
TypeScript Handbook: Type Compatibility	Microsoft (TypeScript)	Actual	Declara que el sistema de tipos no es sólido por diseño, y dónde. https://www.typescriptlang.org/docs/handbook/type-compatibility.html
npm audit; package-lock.json	npm, Inc. (GitHub)	Actual	Verificación de avisos de seguridad y lockfiles. https://docs.npmjs.com/cli/commands/npm-audit y https://docs.npmjs.com/cli/configuring-npm/package-lock-json

Fuente	Editor	Año	Alcance y dirección
Nullable reference types	Microsoft Learn	Actual	Análisis del estado nulo en C#. https://learn.microsoft.com/en-us/dotnet/csharp/nullable-references
Auditing package dependencies for security vulnerabilities	Microsoft Learn (NuGet)	2025	Valores por defecto de la auditoría de NuGet y códigos de advertencia. https://learn.microsoft.com/en-us/nuget/concepts/auditing-packages
Error Prone; SpotBugs	Google; proyecto SpotBugs	Actual	Análisis estático para Java. https://errorprone.info/ y https://spotbugs.github.io/
Verifying dependencies	Gradle, Inc.	Actual	Comprobación de sumas de verificación y firmas. https://docs.gradle.org/current/userguide/dependency_verification.html
The Z Garbage Collector	Oracle (documentación de Java SE 21)	2023	Recolector de pausas breves (Apéndice B). https://docs.oracle.com/en/java/javase/21/gctuning/z-garbage-collector.html

LA SERIE

GDCD en código

GDCD en código aplica el Diseño de Contratos Guiado por la Gobernanza a la puerta del código: el lenguaje, las herramientas y la cadena de suministro por las que pasa todo sistema.

Libro 1 **Gobernar Rust por contrato**: ser dueños de los bits, del compilador al clúster. (Este libro.)

La base del método es *The Contract Decides: Governing AI at Every Door* (en inglés). El mismo método en una plataforma en la nube es *Gobernar Azure por contrato*. Todos los libros se pueden leer gratis en blog.jcardena.com/es/ebooks.

Juan Cardena

Arquitecto de IA empresarial y sistemas agénticos · blog.jcardena.com

Descargue la edición más reciente en blog.jcardena.com/es/ebooks/governing-rust-by-contract.

GDCD en código · Libro 1 · La confianza se gana en cada capa.

Traducción al español generada con herramientas de IA a partir del original en inglés y revisada en varias rondas con modelos de IA.