

GDCD IN CODE

BOOK 1

GOVERNANCE-DRIVEN CONTRACT DESIGN

Governing Rust by Contract

Owning the bits, from the compiler to the cluster.

Most teams choose a language and then discover what it costs. This guide starts the other way: from the outcome, through the guarantees, to the contracts that hold them. It shows what Rust's compiler really proves, what tooling must enforce around it, and what people and operations still own, from the first decision record to a service running on Kubernetes. It is honest about where Rust does not fit.

Juan Cardena

Enterprise AI & Agentic Systems Architect · blog.jcardena.com

CONTENTS

00 Why This Book 3

PART I · THE DECISION

01 Own the Bits: What Rust Changes and What It Does Not 6

02 The Decision Record and Its Renewal 12

03 When Not to Use Rust 18

PART II · THE CODE DOOR

04 The Compiler Contract 23

05 The Exception Contract: Unsafe, FFI and Build-Time Code 30

06 The Verification Contract 37

PART III · THE SUPPLY-CHAIN DOOR

07 The Pin: Dependencies and Toolchain 44

08 The Admission Gate 51

09 The Interface Promise 59

PART IV · THE RUNTIME DOOR

10 Shipping the Artifact 66

11 Running Under Contract 73

PART V · RUN THE DECISION

12 The Operating Model 80

APPENDICES

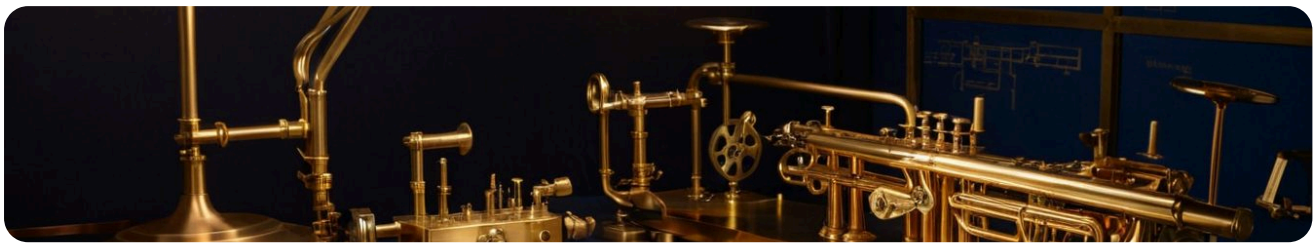
A Templates and the Reference Service 88

B The Three Tiers in Other Languages 104

C Glossary and Source Register 108

PREFACE

Why This Book



Most language debates are arguments about taste that end in a migration nobody can undo. This book is about the part that outlives the debate: the decision itself, who owns it, what it must deliver, and how the estate proves, year after year, that it still holds. Rust is the case study. Governance is the subject.

I have spent more than 25 years building web platforms, software, data systems and, lately, AI and agentic systems. In that time I have watched teams adopt languages for good reasons and bad ones. The good reasons are rarely about syntax. They are about control: where memory lives, what a binary carries, what each dependency brings in, how a service behaves at the 99th percentile on a bad day. I call that **owning the bits**. Rust is one language that lets a team own them without a garbage collector, and it puts a compiler in the loop that refuses whole classes of mistakes before the code ever runs.

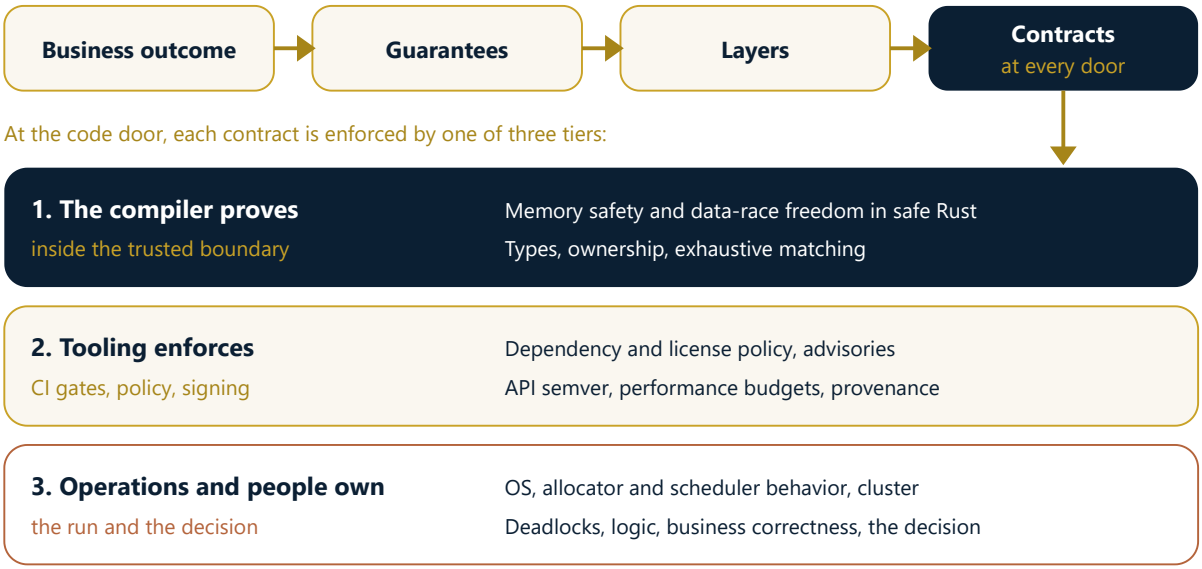
That refusal is valuable, and it is also easy to oversell. A compiler proves some things. It does not prove that your dependencies are honest, that your service drains cleanly when a node is retired, or that the business logic is right. So the book is built on one honest picture of where each guarantee comes from.

Three tiers, one picture

The series this book belongs to, Governance-Driven Contract Design (GDCD), starts from a business outcome, turns it into guarantees, places those guarantees in layers, and enforces them as contracts at every door. In the AI books the line is "the model proposes, the contract decides". At the code door the same idea reads **"the developer proposes, the compiler decides"**, but only for what the compiler can actually check. Everything else needs a different enforcer. The series also has a slogan for the software layer, **"Every line holds"**. In this book it is bounded: a line holds only as far as one of the three tiers enforces it, and it is a target the estate checks with evidence, not a promise that code cannot fail.

FIGURE 0.1

The three tiers, mapped onto the GDCD stack



Every chapter asks which tier holds a guarantee. The closer a guarantee moves to Tier 1, the cheaper and more certain the enforcement. Moving a guarantee up a tier, from people to tooling or from tooling to the compiler, is most of what good engineering governance does.

The tiers are the spine of the book. Each chapter names a contract, says which tier enforces it, who owns it, what happens when it fails, and what evidence shows it ran. The tiers also apply to any language. Appendix B maps them onto TypeScript, Go, .NET and Java, so a reader who never ships a line of Rust can still use the method.

Who this is for

The reader I have in mind is a CTO, a CIO, an enterprise or platform architect, an engineering lead or a CISO. You are accountable for a language decision, or you will be asked to approve one. You may not write Rust yourself. You need to know what it buys, what it costs, how to govern it once it is in the estate, and when to say no.

Every chapter opens with its point in plain words. A CTO can read the lead and the first section and stop. The engineering detail follows for the people who will build the gates.

What this book is not. It is not a Rust tutorial: the official book and its companions do that well, and I point to them. It is not a benchmark brochure: there are no cross-language speed contests here, and the only footprint figures come from a small reference service in Appendix A, built and measured with a method you can rerun. It is not a hiring pitch or a claim that Rust should replace everything. Chapter Three is about when not to use it.

How the book is organized

Part I is the decision: what owning the bits means, how to write the decision record so it can be renewed or reversed, and where Rust does not fit. Part II is the code door: the compiler contract, the exception contract for unsafe code, FFI and build-time code, and the verification contract. Part III is the supply-chain door: pinning dependencies and toolchains, the admission gate for what may enter a build, and the promise a public interface makes. Part IV is the runtime door: shipping the artifact and running it under contract. Part V puts it together as an operating model.

Reading path	Chapters	What you leave with
CTO path	Preface, 1, 2, 3, 8, 11, 12	A decision record you can sign and renew, the admission gate, the runtime evidence, and the operating model
Architect depth	Parts II to IV (Chapters 4 to 11)	The contracts, their enforcement points and the templates in Appendix A

Chapters 4 to 11 each end with a Contract card: the contract, the threat it addresses, its owner, where it is enforced, how it fails, who may waive it and until when, and the evidence that it ran and that it helped. Every chapter from 1 to 12 ends with a short checklist.

Tools appear where they are the standard way to do something, and full configuration files live in Appendix A with the toolchain version they were checked against. Tools change. The contract is what should still hold when they do.

> *Tools are fashion. Trust is the work.*

PART I · THE DECISION

CHAPTER ONE

Own the Bits: What Rust Changes and What It Does Not



A runtime makes promises on your behalf. A garbage collector decides when memory comes back. A virtual machine decides how much of itself ships with your service. Most of the time those promises are good ones, and accepting them is the right call. Rust is a way to take some of those promises back and make them yourself, with a compiler that checks a part of your work. That is a real gain, with a real price, and both belong in the decision.

Here is the short version for a CTO. Rust gives a team control over memory, layout, allocation and binary size without a garbage collector, and its compiler proves memory safety and freedom from data races for code written in safe Rust. Those proofs hold inside a boundary that you must govern. The strongest public evidence for memory-safe languages comes from C and C++ codebases, so it argues for leaving C and C++. The case against managed runtimes such as Java, .NET or Go is different and narrower: it is about predictability and footprint, and it has costs in build time, skills and ecosystem maturity.

What owning the bits means

I use "owning the bits" for **governed accountability** over what a service is made of and how it behaves. It is not a feeling of control, and it is not a claim that the team writes everything itself. A team owns the bits when it can answer, with evidence, five questions about any service it runs:

- **Memory:** where is it allocated, who frees it, and what is the worst case?
- **Binary:** what does the artifact contain, and where did each part come from?
- **Dependencies:** what did each one bring in, who approved it, and what may it do at build time?
- **Behavior under load:** what does the 99th percentile look like, and what is it bounded by?
- **Accountability:** who owns each of these answers, and when were they last checked?

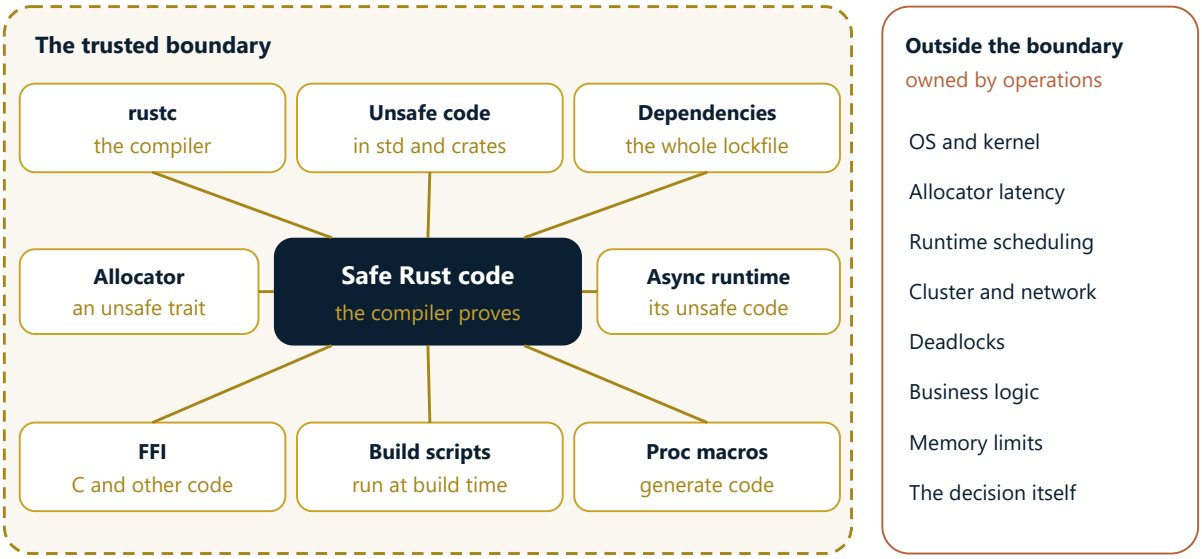
Any language can answer those questions with enough effort. Rust makes some of the answers cheaper, because the compiler checks them, and makes others more direct, because no garbage collector or virtual machine sits between the code and the machine. The fifth question is never answered by a language. It is answered by an operating model, which is why this is a book about governance.

The trusted boundary

The compiler's guarantees are precise, and they are conditional. Safe Rust cannot cause undefined behavior, such as use after free or a data race, but only if everything it stands on is also correct. That foundation is what I call the **trusted boundary**.

FIGURE 1.1

Safe-Rust guarantees depend on everything inside the boundary



The compiler checks your safe code, and trusts the rest. Every box inside the dashed line can break the guarantee if it is wrong. Every item on the right is outside what the compiler can see. The allocator and the async runtime appear on both sides: their soundness is trusted, their behavior under load is operated.

Eight things sit inside the boundary. The **compiler** itself, which has bugs like any software. Every **unsafe abstraction**, in the standard library and in crates, whose authors promise that the safe interface they expose cannot be misused. Every **dependency** in the lockfile, because their unsafe code is now your unsafe code. Every **FFI** call into C or another language, where Rust takes the foreign code's word for its behavior. Every **build script** (build.rs), which runs arbitrary code on the build machine. Every **procedural macro**, which runs at compile time and writes code into your crate. The **global allocator**, because `GlobalAlloc` is an unsafe trait: safe code trusts that it hands back valid memory. The **async runtime**, whose executor, timers and I/O are built on unsafe code that safe futures rely on. For those last two, the line is between soundness and behavior. Whether the allocator and the runtime are correct is inside the boundary. How fast the allocator is, whether it returns memory to the operating system, and how the runtime schedules tasks are outside it, and belong to operations. Part II and Part III of this book are, in large part, the contracts that govern these eight things.

The evidence, with its scope

The case for memory-safe languages rests on public data, and that data has a specific scope. Microsoft's Security Response Center reported that about 70% of the vulnerabilities Microsoft assigns a CVE each year are memory safety issues (MSRC, 2019). The Chromium

project reports that around 70% of its high-severity security bugs are memory safety problems, which it describes as mistakes with C and C++ pointers (Chromium Project, 2020). Google's Android team reported that as new development moved to memory-safe languages, the memory-safety share of Android vulnerabilities fell from 76% in 2019 to 24% in 2024 (Google Security Blog, 2024). CISA, the NSA and partner agencies have published guidance urging manufacturers to plan a move to memory-safe languages (CISA et al., 2023), and the US Office of the National Cyber Director made the same case (ONCD, 2024).

All of this is about codebases written in C and C++. It is strong evidence for leaving C and C++, or for writing new code in a memory-safe language next to them, which is what Android did. **It does not support rewriting Java, .NET, Go or TypeScript services in Rust.** It is not evidence for leaving those languages. Those languages are already memory safe in the sense these reports measure. Using the 70% figure to argue for rewriting a Java service in Rust misreads the source, and a careful reviewer will notice.

What Rust buys against managed runtimes

Against a managed runtime, the argument is about predictability and control, not about memory safety. Four gains are real:

- **No garbage collector pauses.** A value is dropped when its owner goes out of scope, so no collector decides when to pause the service. That removes one source of latency variation, not all of them: destructors take time, freed memory may not return to the operating system, and reference-counted cycles (`Rc`) leak. The rest is measured. Modern collectors such as ZGC have made pauses very short, so this is a difference in kind of control, not a claim that managed services are slow.
- **Small, single binaries.** A Rust service compiles to one executable with no garbage collector or virtual machine to install, which helps keep container images small and startup quick. The binary may still link the C runtime or system libraries, so call it self-contained only after inspecting its dynamic dependencies. .NET NativeAOT, GraalVM native image and Go also produce single native binaries, so this is a strength Rust shares rather than owns.
- **Data-race freedom in safe code.** The Send and Sync traits let the compiler reject code that would share mutable state between threads unsafely. The type systems of Go and Java do not prevent data races: Go ships a dynamic race detector, and Java relies on synchronization discipline, testing and analysis tools.
- **Control of layout and allocation.** The team decides how data is laid out in memory and when it is allocated. That matters for parsers, proxies, codecs and anything on a hot path.

I do not put numbers on these gains in this book, because cross-language benchmarks depend heavily on how each version was written and tuned, and rarely match a given workload. Appendix A measures footprint on a reference service, with the method stated first; Chapter Eleven gives the latency method.

What it costs

The costs are just as real, and a decision record that omits them will be reversed by the first budget review.

- **Compile times.** Rust builds are slower than Go builds, and large workspaces need caching, careful dependency budgets and more CI capacity.
- **Learning curve.** Ownership and lifetimes take time to become fluent in. Productivity dips before it recovers; measure the dip in a pilot rather than assume a figure.
- **Hiring pool.** Experienced Rust engineers are usually harder to find than Java, C# or Go engineers. Check your own hiring market, and plan to grow them.
- **A younger ecosystem in places.** Networking, serialization and command-line tooling are mature. Some enterprise integrations, reporting formats and vendor SDKs are thinner or missing. Chapter Three covers what to do about that.

> **What Rust does not guarantee.** Safe Rust does not prevent **deadlocks**, or **race conditions in general**: it prevents data races, not logic races where two correct operations happen in the wrong order. It does not prevent **memory and resource leaks**, which the Rust Reference lists as behavior not considered unsafe. It does not prevent **panics**, and a panic in a service is still an outage. Async code can be **cancelled** at any await point, leaving work half done. **Integer overflow** panics in debug builds and wraps silently in release builds by default. Running **out of memory** aborts the process by default. And nothing in the language bounds **latency** by itself: the allocator, the async scheduler, the kernel and the cluster all have a say. Every item here is a contract for tooling or for operations.

The overflow point surprises people, so it is worth one example. A team that wants overflow to fail loudly in production can say so in the release profile, or make the intent explicit in the code:

```
# Cargo.toml
[profile.release]
overflow-checks = true   # panic on overflow in release, as in debug

// src/stock.rs
pub fn add_units(stock: u32, incoming: u32) -> Option<u32> {
    stock.checked_add(incoming)   // None on overflow, never a silent wrap
}
```

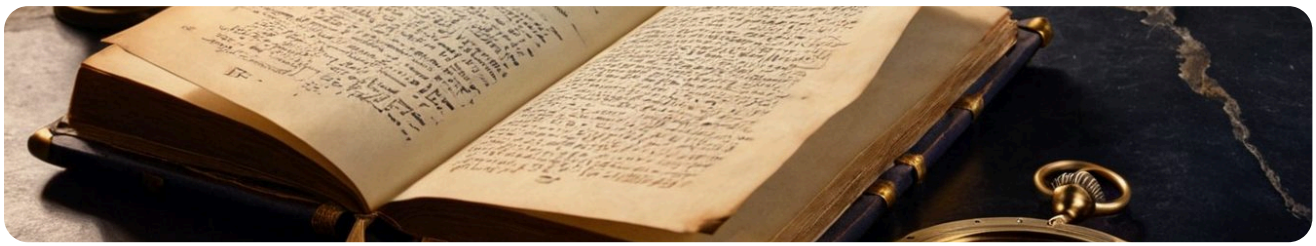
That is the pattern of the whole book. The language gives a default. The team decides whether the default matches the guarantee it promised, writes the decision down, and makes a machine enforce it where it can. Where the developer proposes and the compiler decides, the line holds. Where the compiler cannot see, someone must own it.

CONTRACT CHECKLIST

- The decision names the business outcome Rust serves, in terms of memory, binary, dependencies, behavior under load and accountability.
- Memory-safety evidence is cited with its scope: C and C++ codebases, not managed runtimes.
- The trusted boundary is written down: compiler version, unsafe code, dependencies, FFI, build scripts, proc macros, the allocator and the async runtime, each with an owner.
- Each item in "What Rust does not guarantee" is assigned to a tooling or operations contract.
- Costs (build time, learning curve, hiring, ecosystem gaps) appear in the decision record next to the gains.

CHAPTER TWO

The Decision Record and Its Renewal

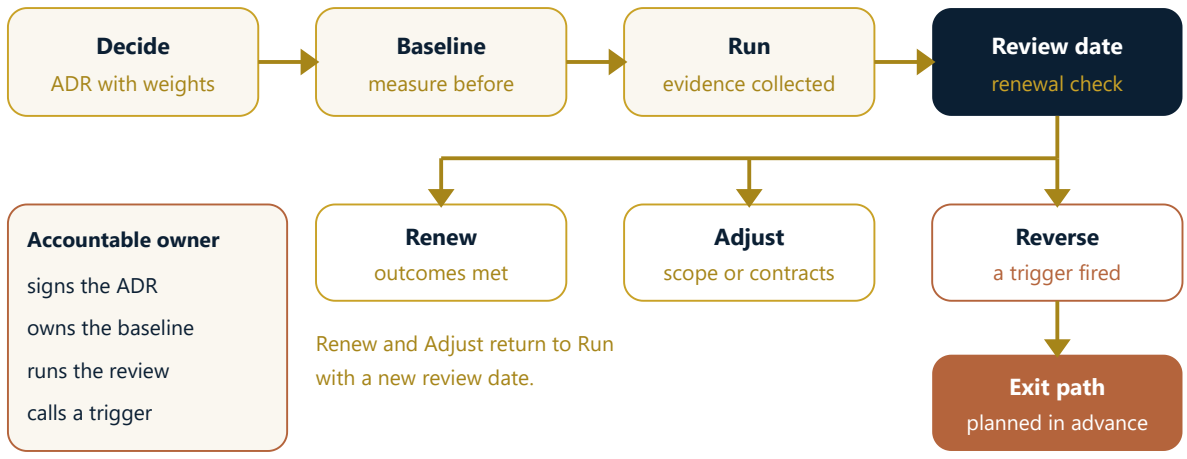


Language decisions are usually made once, in a meeting, and then defended for a decade by people who were not in the room. The fix is not a better meeting. It is a decision record that states what the choice must deliver, how anyone will know, when it will be checked again, and what would make the estate change course. A decision with a review date and an exit path is a contract. A decision without them is a habit.

The short version: write an architecture decision record (ADR) before the first service ships. Score the options against criteria whose weights are fixed before anyone scores. Give every alternative a fair hearing in writing. Count the total cost, not only the runtime bill. Then attach a renewal contract: expected outcomes, a baseline, a review date, an accountable owner, reversal triggers and an exit path. The rest of this chapter is the template.

FIGURE 2.1

A decision that can be renewed, adjusted or reversed



The review date is part of the decision, not an afterthought. Renewal is a choice the owner makes on evidence. Reversal is only cheap if the exit path was designed before it was needed.

The template

An ADR for a language choice is short. Its value is in what it forces people to write down. The full template, ready to copy, is in Appendix A.

Field	What to write
ID, title, status	"ADR-0000: Rust for latency-critical services". Status is proposed, accepted, renewed, adjusted or reversed.
Accountable owner	One named role, not a committee. The person who will run the review.
Business outcome	The outcome the choice serves, in business terms. For example: a checkout path whose 99th percentile stays inside its budget during peak.
Guarantees required	What must be true, and which tier enforces each: compiler, tooling or operations (Chapter One).
Scope	Which services, which teams, which kinds of new code. What is explicitly out of scope.
Options considered	Every serious alternative, with the fair case for it written by someone who would choose it.
Criteria and weights	Fixed before scoring. Scores and the reasoning behind each.
Decision and costs accepted	What was chosen and what the estate knowingly pays for it.
Renewal contract	Expected outcomes, baseline, review date, reversal triggers, exit path (below).
Evidence location	Where the measurements and the review minutes live.

Criteria with weights fixed in advance

The order matters more than the numbers. Agree the criteria, agree the weights, then score. If the weights move after the scores are in, the record is a rationalization. I do not give weights here, because they depend on the outcome the estate is buying. The placeholders show the shape.

Criterion	Weight	What counts as evidence
Predictable tail latency and resource use	w1	Measured on a reference workload with a stated method
Safety at edges that parse untrusted input	w2	Threat model, history of the component class
Team skills and hiring	w3	Current skills inventory, local hiring market, training plan
Ecosystem fit for the domain	w4	Named libraries and SDKs needed, their maturity and owners
Build and CI cost	w5	Build times and CI minutes from a pilot, not estimates
Operational maturity	w6	Profiling, tracing and debugging tools the operators already know
Interop with the existing estate	w7	Interfaces to existing services, data formats, identity
Long-term maintainability	w8	Release cadence, upgrade effort, who maintains the toolchain

Weights w1 to w8 sum to 100 and are signed by the owner before scoring begins.

A fair paragraph for each alternative

A decision record earns trust when its alternatives are described by someone who would choose them. Each language below wins somewhere, and the record should say where.

Go. Fast builds, a simple concurrency model with goroutines and channels, a strong networking standard library and single static binaries. Its garbage collector is designed for low pause times. For network services where the team values speed of delivery and readability over control of memory, Go is often the better choice.

C# and .NET. High productivity, mature tooling, a large library ecosystem and deep enterprise integration. NativeAOT compiles apps ahead of time into a self-contained native executable with fast startup and a smaller footprint, with some limits on reflection and runtime code generation. A team already strong in .NET pays a real price to leave it.

Java. Decades of operational maturity, profiling and monitoring that operators know well, and a large, mature library ecosystem. ZGC is a low-pause collector built for large heaps, and GraalVM native image builds standalone native executables for faster startup, under a closed-world assumption that needs care with reflection. Java is a sound default for many back-office and data services.

C++. A deep base of existing code, libraries and expertise in many domains, and full control over memory and layout. Where a large, well-tested C++ codebase exists, the realistic question is often how to write new components in a memory-safe language next to it, not whether to rewrite it.

Zig. A young systems language with explicit allocation and simple C interop. It has not reached version 1.0, and the language and standard library still change between releases. Watch it; do not yet bet an enterprise estate on it.

TypeScript and Node.js. The natural language of the browser and of the services that sit closest to it. Shared types between front end and back end, a large package ecosystem and a large hiring pool make it strong for user interfaces and I/O-bound edge services. It is a poor fit for CPU-heavy hot paths, which is often where Rust joins it (Chapter Three).

Total cost, not runtime cost

The runtime bill is the easiest number to find and the least likely to decide the outcome. The record should carry a line for each of these, from a pilot where possible:

- **Hiring and onboarding:** time to fill a role, and time until a new engineer ships safely.
- **Training:** courses, pairing and the productivity dip during the learning period, measured in the pilot.
- **CI minutes and build hardware:** Rust builds need more compute and good caching.
- **Toolchain upkeep:** stable Rust releases every six weeks, and someone must own upgrades (Chapter Seven).
- **Dependency vetting:** review effort for new crates and their unsafe code (Chapter Eight).
- **Runtime cost:** compute and memory, measured on the reference service in Appendix A, not quoted from a benchmark.

The renewal contract

This is the part most decision records leave out, and the part that makes the record a contract.

RENEWAL CONTRACT FIELDS

- **Expected outcomes:** what will be measurably better, for example tail latency inside budget at peak, or fewer memory-safety findings in a parser that replaced C code.
- **Baseline:** the same measurements, taken before the change, with the method written down.
- **Review date:** a calendar date set by the owner (six to twelve months out is a planning assumption, not a rule), with the review on the owner's calendar.
- **Accountable owner:** the role that runs the review and decides to renew, adjust or reverse.
- **Reversal triggers:** conditions agreed in advance, such as roles unfilled for two hiring cycles, build times beyond a stated budget, or outcomes not met at two reviews.
- **Exit path:** how services would move off Rust if a trigger fires: interfaces kept language neutral, a named target stack, and an estimate of the effort.

EXAMPLE: A REVIEW THAT ADJUSTS

Consider a team whose record promised tail latency inside budget for three services. At the first review, two services meet the outcome and one, a mostly CRUD service, shows no gain and slower delivery. The owner renews Rust for the two hot-path services and adjusts scope: new CRUD services return to the estate's managed stack. That is the record working, not failing.

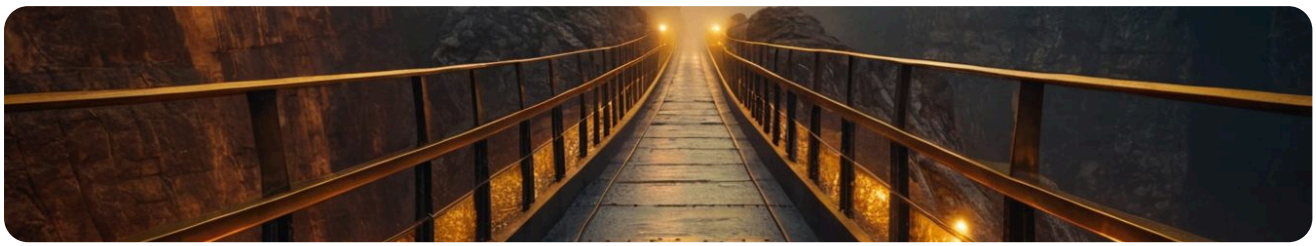
An exit path is easiest when interfaces are language neutral from the start: versioned HTTP or gRPC contracts, schemas in a shared registry, and no service that depends on another's internal types. Chapter Nine covers that promise in detail.

CONTRACT CHECKLIST

- An ADR exists before the first Rust service ships, with one accountable owner.
- Criteria and weights were signed before scoring, and each alternative has a fair written case.
- Total cost lines include hiring, onboarding, CI and build hardware, toolchain upkeep and vetting.
- A baseline was measured with a stated method before the change.
- The review date, reversal triggers and exit path are written in the record.

CHAPTER THREE

When Not to Use Rust



The most credible thing a Rust advocate can say is where Rust should not go. A language that is right for everything is a language nobody has evaluated. Saying no to Rust in the right places is what protects the places where it earns its cost.

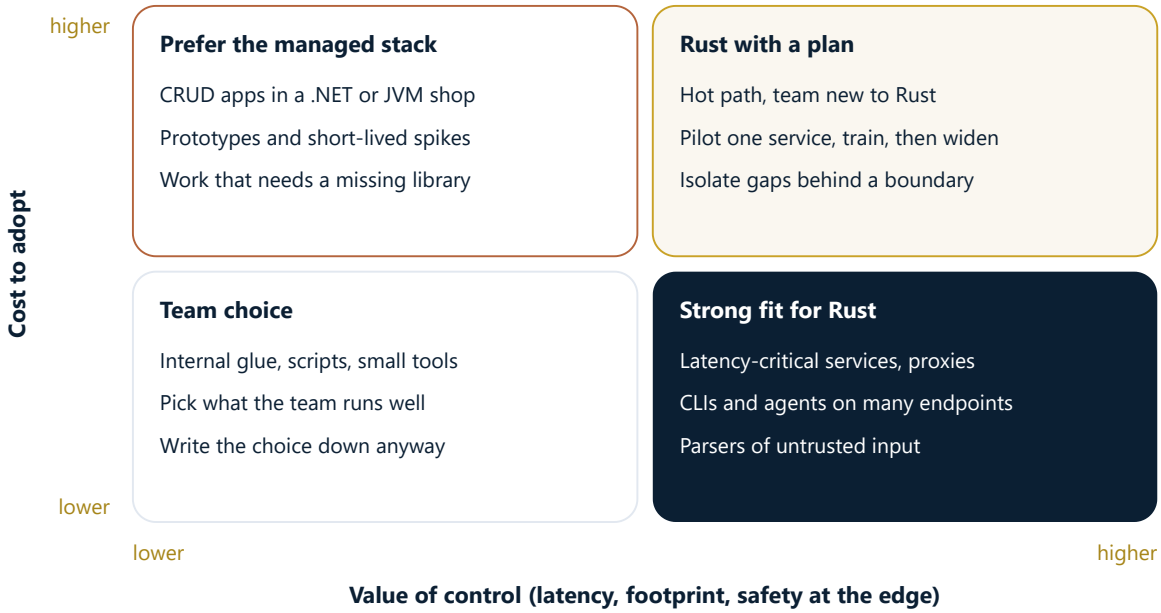
The short version for a CTO: Rust fits where control over latency, memory and footprint is worth paying for, and where the code will live long enough to repay the learning curve. It is usually the wrong choice for prototypes, for line-of-business applications in a team that is already strong in .NET or the JVM, and for user interfaces. Many estates end up with a split: a Rust core for the hot paths and TypeScript for the interface. When a needed library exists only in another language, isolate it behind a versioned boundary with an owner and an expiry date. And a team with no platform team can start with five contracts, listed at the end of this chapter.

The fit matrix

Two questions place most workloads. How much is control worth here: tail latency, memory footprint, safety at edges that parse untrusted input? And how much will adoption cost here: team skills, missing libraries, pressure to ship this quarter?

FIGURE 3.1

Fit matrix: value of control against cost to adopt



Place the workload, then the team. The same service can sit in different quadrants for different teams. The matrix is an input to the decision record in Chapter Two, not a substitute for it.

Three placements deserve a closer look, because they are where estates most often get it wrong.

Prototypes. A prototype exists to learn something quickly and then be thrown away. Rust's compiler asks for decisions about ownership and error handling that a prototype does not need yet. Build the spike in whatever the team writes fastest. If the idea survives and the workload lands in the right half of the matrix, write the production service in Rust with what the spike taught you.

Teams deep in .NET or the JVM. A team with years of C# or Java, a working deployment pipeline, profilers the operators trust and a backlog of business features has a lot to lose. For CRUD services, workflow engines and integration code, the managed runtime is usually the better tool. Its garbage collector is a reasonable price for productivity, and NativeAOT or GraalVM native image can close much of the startup and footprint gap where that matters. Rust may still earn a place for one hot component, called over a clean interface.

User interfaces. Rust can target WebAssembly, and there are Rust UI frameworks. But for most enterprise front ends, TypeScript has the frameworks, the component libraries, the design tooling and the people. Choosing Rust for the interface usually trades a large ecosystem for a small gain.

Workloads move. A placement is not permanent. A back-office service can become a hot path when a new channel sends it far more traffic, and a Rust pilot can stay in the costly quadrant because the team never got the training it was promised. The review date in the decision record is where placements are checked again, with the same two questions and fresh evidence.

A common split: Rust core, TypeScript interface

Many teams settle on a pattern that plays to both strengths. The core that does heavy work, such as parsing, routing, compression, search or policy evaluation, is Rust. The interface and the thin services right behind it are TypeScript. The two meet at a versioned contract.

- **Over the network:** an HTTP or gRPC API with a schema in source control. Types for TypeScript are generated from the schema, not written by hand, so the two sides cannot drift silently.
- **In the browser:** a Rust library compiled to WebAssembly for one heavy task, with a small, typed interface the UI calls.
- **On the desktop:** a Rust process with a web-technology interface on top, again speaking a versioned message format.

The split works because each side is governed by the contracts that suit it, and the boundary between them is itself a contract with an owner. Chapter Nine treats that interface promise in full.

The ecosystem-gap exception

Sooner or later a Rust estate needs something that exists only in another language: a vendor SDK, a document format, a protocol library with years of fixes behind it. The wrong answers are to write a quick binding and hope, or to drop the decision record and let each team pick. The right answer is a governed exception.

> **HYPOTHETICAL EXAMPLE: A REPORTING FORMAT WITH NO RUST LIBRARY**
>
> Consider a team whose Rust service must produce documents in a format that only a mature .NET library handles correctly. The team does not bind to that library through FFI inside the Rust process. It runs a small .NET service beside the Rust service and talks to it over a versioned protocol: a request schema, a response schema and a version number in every message. The exception record names an owner, the reason, the boundary, and an expiry date when it will be reviewed: renew it if no Rust option has matured, retire it if one has.

THE EXCEPTION PATTERN

- **Isolate:** put the foreign code behind a process or protocol boundary, not inside the Rust address space, unless FFI is the only option and it goes through the exception contract in Chapter Five.
- **Version:** the boundary carries an explicit version, so either side can change without surprising the other.
- **Own:** one named owner for the exception, separate from the owner of the Rust service.
- **Expire:** every exception has a review date, and an expired exception is visible in the register.

This keeps the exception honest. It is allowed, it is visible, and it does not quietly turn into a second platform.

Minimum viable governance

Most of this book assumes a platform team that owns the toolchain, the CI templates and the contract register. Many organizations do not have one. A small team adopting Rust can still govern it, with a starter set of five contracts that are small enough to put in place before the first service ships. Chapter Twelve shows how to grow from this tier to full governance.

Starter contract	Tier	What it takes
1. A decision record	People	The Chapter Two ADR, with an owner and a review date
2. A pinned toolchain and lockfile	Tooling	rust-toolchain.toml and a committed Cargo.lock in every repository; libraries also test against the latest compatible versions on a schedule
3. A CI gate that blocks	Tooling	Format, lints as errors, tests, all with the lockfile enforced
4. Dependency admission	Tooling	Advisories and licenses checked on every build
5. No unsafe by default	Compiler	unsafe_code forbidden in application crates; any exception recorded with an owner

In practice the first three contracts are a few lines each. A short excerpt follows; the full templates, with the toolchain version they were verified against, are in Appendix A.

```
# rust-toolchain.toml: pin the exact stable release the estate approved
[toolchain]
channel = "1.100.0"
components = ["clippy", "rustfmt"]

# Cargo.toml: make "no unsafe" a compiler rule, not a convention
[lints.rust]
unsafe_code = "forbid"

# CI gate: every step must pass, and a failure blocks the merge
cargo fmt --check
cargo clippy --all-targets --locked -- -D warnings
cargo test --locked
cargo deny check advisories licenses
```

The version shown is only an example; the point is that the version is pinned and changed on purpose. Rust 1.100.0 is the current stable release as this edition is written (released 2026-11-12, per the Rust release notes). The 2024 edition has been stable since Rust 1.85, which matters when a team decides which edition its crates use (Chapter Seven). Note what the starter set leaves out: supply-chain vetting, signing and attestation, an unsafe register, API checks, runtime budgets. Those come later, as the estate grows. The starter set is enough to make the decision visible, the build repeatable and the obvious failures blocking.

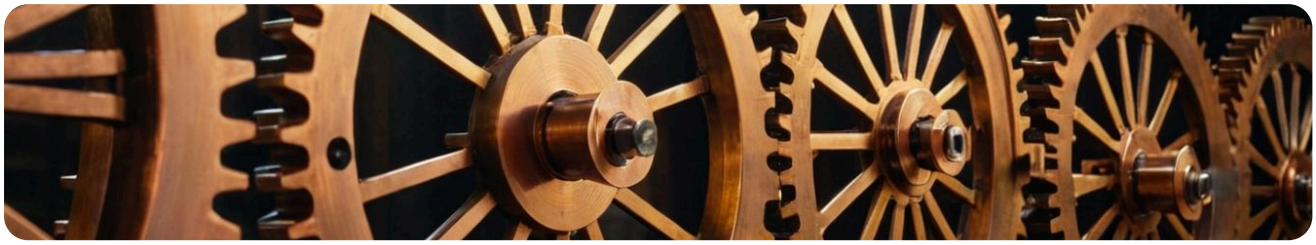
Saying no is part of governance too. A decision record that says "Rust for these workloads, not for those" is stronger than one that says "Rust everywhere", because it can be checked, and because the teams on the other side of the line can see that their stack was taken seriously.

CONTRACT CHECKLIST

- Each candidate workload is placed on the fit matrix before a language is chosen, and the placement is in the ADR.
- Prototypes are built in the fastest stack and rewritten deliberately, not promoted by accident.
- Where Rust and TypeScript meet, the boundary is a versioned schema and client types are generated.
- Every ecosystem-gap exception is isolated behind a versioned boundary, with an owner and an expiry date.
- Teams without a platform team run the five starter contracts from day one.

CHAPTER FOUR

The Compiler Contract

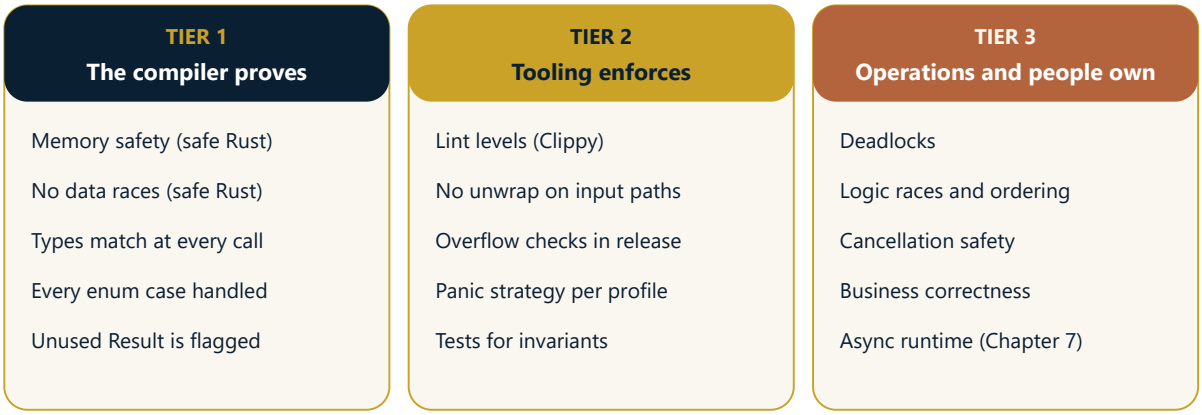


Most governance runs after the code is written: a review, a scan, a test in staging. Rust moves part of it earlier, into the build itself. The compiler refuses to produce a binary when certain rules are broken, and nobody can skip that step by being busy. This chapter is about using that refusal on purpose. The contract is simple to state: the important rules of the business live in types, every failure is a value someone must handle, and crashes follow a written policy. The risk it removes is a class of defects that would otherwise reach production quietly: mixed-up identifiers, impossible states, ignored errors, and unsafe sharing between threads. The cost is real too: slower builds, a learning period for the team (a planning assumption; measure it), and design work up front that some teams would rather defer.

If you are a CTO, the one thing to keep is the boundary. Inside safe Rust, the developer proposes and the compiler decides, but only about memory, types and data races. It decides nothing about deadlocks, business logic or what happens when a request is cancelled halfway. Those stay with tooling and with people, and this chapter ends by saying so plainly.

FIGURE 4.1

The code door, split into its three tiers



The compiler column is short on purpose. It is the only column with proofs. The other two need owners, gates and evidence, which is what the rest of Part II provides.

Types as contracts: make illegal states unrepresentable

A type is a promise the compiler checks at every call site, in every build, forever. That makes types the cheapest contract an estate can write. The practical rule is old and still good: make illegal states unrepresentable. If a value must never be confused with another, give it its own type. If a thing can only be in one of a few states, say so in the type, and let the compiler refuse everything else.

```
#[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) does not compile: the types differ.
```

This is the newtype pattern. It costs nothing at run time, because the wrapper compiles away, and it removes a whole family of bugs where two numbers of the same shape are swapped. The same idea scales to states. An enum lists every state a thing may be in, and each state carries only the data that makes sense for it.

```
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,
    }
}
```

A pending payment has no settlement reference, so no code can read one by mistake. And when someone adds a new state, say `Disputed`, every `match` that does not handle it stops compiling. That is the contract doing review work for you: the change cannot merge until each decision point has made a decision. There is one way to weaken it. A wildcard arm (`_ => false`) silently absorbs new states. For domain enums, many teams ban that with the Clippy lint `wildcard_enum_match_arm`, which is off by default and must be opted in. Public enums in shared libraries are a different case, covered with interface promises in Chapter 9.

Ownership: who may change what, and when

Ownership is the rule that each value has one owner, and that at any moment you may have either many readers or one writer, never both. Lifetimes are how the compiler checks that a borrowed reference never outlives what it points to. For a governance reader the meaning is simple. The question "who is allowed to change this data right now?" has an answer the compiler enforces, instead of an answer written in a comment. That is where memory safety and data-race freedom in safe Rust come from. It is also why I expect the first months to be slower, a planning assumption to measure in the pilot: the team is learning to state ownership up front, where other languages let it stay implicit until something breaks.

Errors as values, and a panic policy

Rust has no exceptions that travel invisibly up the stack. A function that can fail returns `Result`, and the caller has to do something with it. `Result` is marked `#[must_use]`, so ignoring one draws a compiler warning, and the `?` operator passes an error up to the caller in one visible character.

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

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

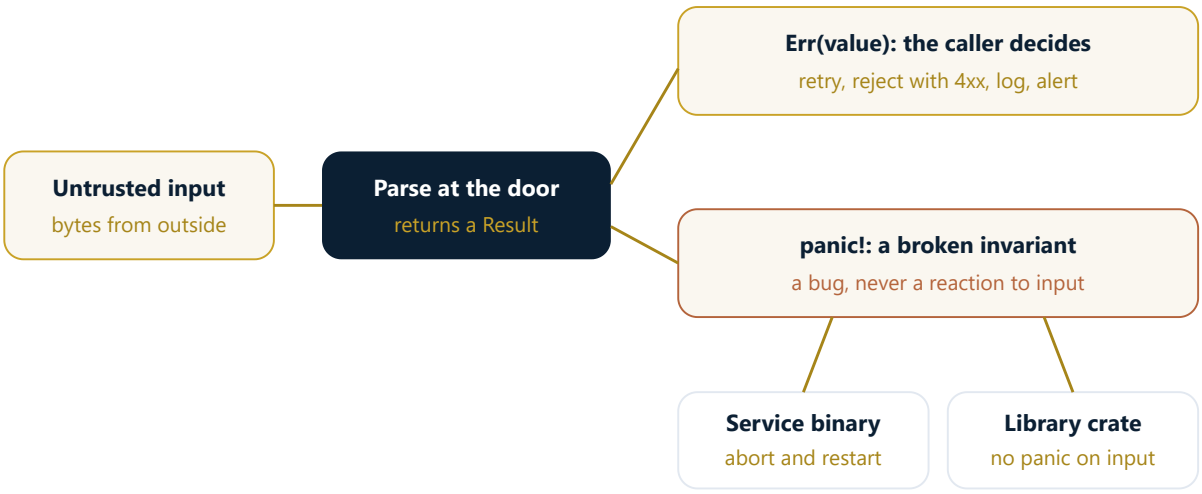
pub fn parse_name(input: &str) -> Result<Name, ParseError> {
    let s = input.trim();
    if s.is_empty() { return Err(ParseError::Empty); }
    if s.len() > 64 { return Err(ParseError::TooLong(s.len())); }
    Ok(Name(s.to_owned()))
}

pub fn load(input: &str) -> Result<Record, ParseError> {
    let name = parse_name(input)?; // the error goes to the caller
    Ok(Record { name })
}
```

Every failure path is now in the signature, which means it is in code review. Panics are the other channel. A panic means "this should be impossible", and it stops the thread. By default the stack unwinds; with `panic = "abort"` in the release profile the whole process stops at once. The trouble starts when panics are used for ordinary bad input, often through `unwrap()`. Then one malformed request can take down a worker, and an attacker who finds it has a denial of service.

FIGURE 4.2

Two channels for failure, and who decides



Expected failure is data; a panic is a bug report. The service may choose to abort and restart clean. The library may not choose for its callers.

So the panic policy needs two rules, not one. For a service binary, `panic = "abort"` is acceptable when the platform restarts the process and in-flight work is safe to retry. It makes a broken invariant loud and the recovery path uniform, and it can shrink the binary. Two cautions: with abort, destructors do not run, so buffered logs and traces must be flushed by design rather than on the way out; and Cargo ignores the setting for tests, build scripts and proc macros. For library code, the rule is stricter: no panic may be reachable from input. A library does not know whether its caller aborts, unwinds or is serving thousands of other users in the same process. It returns `Result` and lets the caller decide.

```
# Service binary: Cargo.toml
[profile.release]
panic = "abort"          # service binaries only
overflow-checks = true   # opt in: overflow panics in release too

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

```
# Library crate: Cargo.toml (no panic reachable from input)
[lints.clippy]
unwrap_used = "deny"
expect_used = "deny"
panic = "deny"
indexing_slicing = "warn"
```

These levels are a starting point, not a standard. The full templates are in Appendix A, which has the same two tables: the service binary and the library (library: `unwrap_used`, `expect_used` and `panic` denied). Exceptions go through `#[expect(lint, reason = "...")]`, stable since Rust 1.81, so each waiver carries its reason in the code and warns when it is no longer needed. Chapter 5 gives those waivers an owner and an expiry.

Send, Sync and async, at the governance level

Two marker traits carry the concurrency contract. A type is `Send` if it may move to another thread, and `Sync` if it may be shared between threads by reference. The compiler works these out and refuses to compile code that breaks them. A single-threaded reference counter (`Rc`) cannot be handed to another thread by accident. A plain value cannot be mutated from two threads without a lock or an atomic. This is the data-race freedom in the first column of Figure 4.1, and it is a proof, not a convention.

Async adds a layer the language does not own. Rust defines futures and `async` syntax, but the scheduler that runs them comes from a library, the async runtime. Most network services use one, and the choice shapes thread model, timers, I/O, which crates fit together,

and how teams debug in production. That makes the runtime an estate decision, not a per-team taste. Chapter 7 puts it in the dependency contract as an estate-standard crate. One practical rule belongs here: holding a lock guard across an `.await` is a common source of stalls, and Clippy's `await_holding_lock` lint catches the standard library case.

What the compiler does not cover

> **Restating the limits.** Safe Rust does not prevent deadlocks: two threads can wait on each other's locks forever. It does not prevent logic races, where every access is synchronized but the order of events is still wrong. It does not make async code cancellation safe: a future dropped at an `.await` can leave work half done, and the runtime's documentation on cancel safety must be read per operation. It does not prevent panics, memory leaks or running out of memory. And integer overflow panics in debug builds but wraps silently in release builds, unless `overflow-checks` is enabled or the code uses `checked_`, `saturating_` or `wrapping_` arithmetic on purpose.

None of this weakens the case. It places it. The compiler takes a large class of defects off the review table, so the people at that table can spend their attention on the classes it leaves behind. Other languages draw this line in different places, and Appendix B maps it for TypeScript, Go, .NET and Java.

> *The developer proposes, the compiler decides: about memory, types and data races. Every line holds only as far as that proof reaches. Past it, the contract needs owners.*

Contract card

Field	The compiler contract
Contract	Domain rules live in types; failures are values; panics follow a written policy per crate kind
Threat addressed	Swapped identifiers, impossible states, ignored errors, crash on bad input, unsafe sharing, silent overflow
Owner	Platform team owns the lint and profile baseline; each crate owner owns its domain types
Enforcement point	The compiler on every build; Clippy at deny levels in the CI gate; workspace release profile
Failure behavior	The build fails and the change cannot merge
Waiver authority and expiry	Local `#[expect]` with a reason, approved by the crate owner; baseline changes by the platform team; expiry per the Chapter 5 lifecycle
Evidence the control ran	CI logs of build and Clippy with the pinned toolchain; the count and age of `expect` and `allow` attributes
Evidence the outcome improved	Production crashes from panics and incidents from invalid state, against the baseline in the decision record
Review cadence	On each toolchain bump, and quarterly for the lint baseline

CONTRACT CHECKLIST

- Identifiers and quantities that must not be mixed have their own types, and domain states are enums without wildcard arms.
- Library crates deny `unwrap_used`, `expect_used` and `panic` on code paths reachable from input.
- Each service records its panic strategy and, if it aborts, how logs and traces are flushed and how the process restarts.
- Release profiles state `overflow-checks` explicitly, and money or quota arithmetic uses checked operations.
- The async runtime is named once for the estate, in the dependency contract (Chapter 7).
- The team can list, without looking, what the compiler does not prove.

CHAPTER FIVE

The Exception Contract: Unsafe, FFI and Build-Time Code

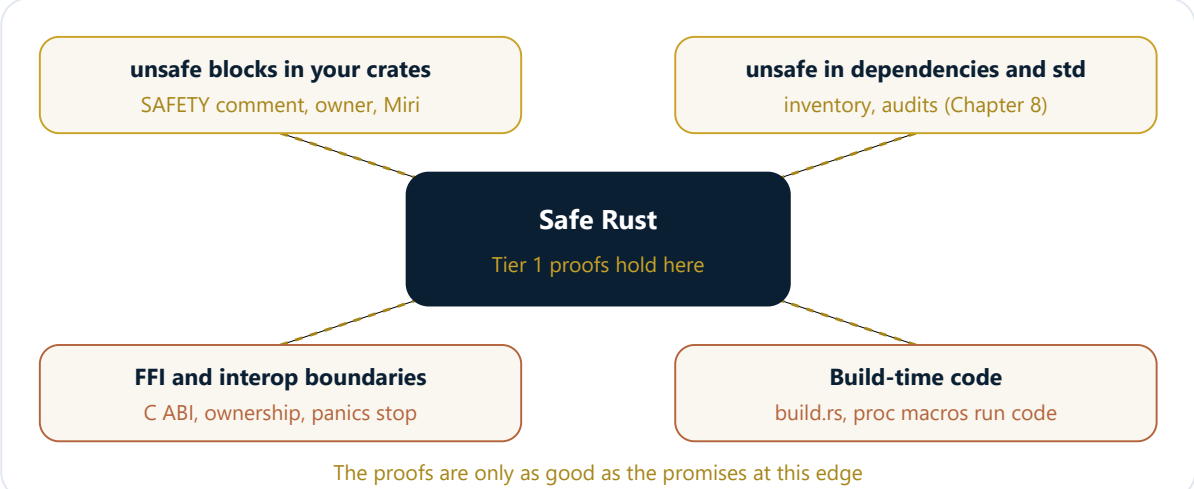


Every proof in Chapter 4 rests on code the compiler cannot check. The standard library uses `unsafe` inside. So do many popular crates. Calls into C, C# or a JVM cross a line the compiler cannot see past, and build scripts run arbitrary code on the build machine before any of it is compiled. None of that is a scandal. It is how a systems language reaches the hardware and the rest of the world. The contract in this chapter treats each of these places as a signed exception: a named person states what must be true, someone else checks it, and the exception has an expiry date like any other waiver. The risk it removes is unowned unsafety, the kind nobody remembers approving. The cost is review time, some nightly-only tooling, and the discipline to keep a register current.

If you are a CTO, keep one sentence: the guarantees of safe Rust are only as strong as the promises made at its edge, so every promise at that edge needs an owner and evidence.

FIGURE 5.1

The trusted boundary and its four edges



Four edges, one rule. Gold edges are about memory: a human proves what the compiler cannot. Copper edges are about trust: code from outside runs with your privileges or across a line no type crosses.

What unsafe means, and what it does not

An `unsafe` block does not switch off the borrow checker or the type system. The Rust book lists the five main operations it permits: dereferencing a raw pointer, calling an unsafe function (including a foreign one), reading or writing a mutable static, implementing an unsafe trait, and reading a union field. Inside the block, the developer takes over one job from the compiler: proving that those operations cannot cause undefined behavior. That is the exact shape of a waiver. The contract is suspended for a stated reason, and someone signs for the reason.

The language already gives you most of the paperwork. By convention, an unsafe function documents its preconditions under a `# Safety` heading, and every unsafe block carries a `// SAFETY:` comment that says why its preconditions hold here.

```
/// Returns the first byte without a bounds check.
///
/// # Safety
/// `buf` must not be empty.
pub unsafe fn first_unchecked(buf: &[u8]) -> u8 {
    // SAFETY: the caller guarantees buf.len() > 0,
    // so index 0 is in bounds.
    unsafe { *buf.get_unchecked(0) }
}
```

The inner `unsafe` block is required in spirit and, since the 2024 edition (Rust 1.85), warned about by default: the `unsafe_op_in_unsafe_fn` lint asks for an explicit block even inside an unsafe function, so each risky operation is marked where it happens. The 2024 edition also requires `unsafe extern` on foreign declarations, which puts the signature of every foreign function under the same rule. Inside such a block, an item may be marked `safe`, and callers then use it without an `unsafe` block; items without a marker stay unsafe to call. The `safe` marking is itself an unchecked promise about the foreign code, so it needs the same review as any unsafe block.

Turning the convention into a gate

A convention becomes a contract when a build fails without it. Three lint settings do most of the work, set at crate or workspace level in the `[lints]` table.

```
[lints.rust]
unsafe_code = "forbid"            # crates with no reason for unsafe
unsafe_op_in_unsafe_fn = "deny"

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

The difference between the two strong levels matters. `forbid` cannot be overridden lower down, so it fits the large majority of crates that have no reason to touch raw memory. `deny` can be relaxed with a local attribute, so it fits the few crates that genuinely need unsafe: there, it stays denied at the crate level and is allowed only in one named module, which is where the register entry points. The Clippy lint `undocumented_unsafe_blocks` belongs to the opt-in restriction group; it fails any unsafe block without a `SAFETY` comment. The shared lint template in Appendix A sets `unsafe_code` to `deny`, so a crate that needs unsafe can allow it in one named module; crates that never need unsafe tighten it to `forbid`, as in the excerpt above.

The unsafe register

The lints prove the comment exists. They cannot prove it is true. That takes a person and some evidence, and the record of both is the unsafe register: one row per unsafe module, kept next to the code so it changes in the same review.

Field	Example (hypothetical)
Location	crates/codec/src/simd.rs, module `fast_path`
Owner	A named lead on the codec team
Invariant	Input length is a multiple of 16; pointers come from a live slice
Why unsafe	Vector instructions; the safe version is measured slower in the reference benchmark
Reviewer	A second engineer trained on the Rustonomicon, not the author
Evidence	`cargo miri test -p codec` clean; fuzz target run on schedule with no findings
Expiry	A date, and earlier if a safe standard library API covers the case

Miri is the Rust project's interpreter for detecting undefined behavior, such as out-of-bounds access, use after free and invalid values. It runs only on nightly, it checks only the paths the tests execute, and it cannot run most foreign calls. Sanitizers such as AddressSanitizer are still unstable compiler options in Rust, so they too usually need a nightly toolchain. So the evidence is "the tests that exercise this module pass under Miri", not "this module is proven sound". Chapter 6 puts both tools in the verification contract.

Unsafe you did not write

`forbid(unsafe_code)` applies to your crate, not to its dependencies. A service with no unsafe of its own can still run a great deal of it through the crates it pulls in, and that is normal: well-reviewed unsafe in a widely used crate is often safer than a homemade safe alternative. The contract is about knowing, not about zero. The community tool `cargo-geiger` is one way to see it: it reports unsafe usage per dependency in the build. The decisions that follow, which crates are trusted, who audited them and how a new one is admitted, belong to the admission gate in Chapter 8.

FFI: the line no type crosses

Interop is where Rust earns its place in an existing estate, and where the compiler's view ends. The C ABI is the common language: almost every other runtime can call a C function, and Rust can export one.

Direction	Usual path	Contract point
C or C++ calls Rust	`extern "C"` functions; `cbindgen` generates the header	The header is generated, never written by hand
Rust calls C	`bindgen` generates declarations from C headers	Calls are unsafe unless a declaration is marked `safe`; every declaration, marked or not, needs a register entry
C# calls Rust	C ABI plus P/Invoke	Who frees memory is written down
JVM calls Rust	JNI, or Java's Foreign Function and Memory API	Threads attached and detached by rule
Kotlin, Swift, Python	UniFFI (Mozilla) generates bindings from one definition	Interface definition versioned (Chapter 9)
TypeScript, browser	WebAssembly with `wasm-bindgen`	The sandbox, not the type, is the boundary

Three rules hold for every row. First, memory allocated on one side is freed on the same side; a Rust library that hands out a buffer also exports the function that releases it. Second, a panic must not cross the boundary. Since Rust 1.81, a panic that unwinds out of an `extern "C"` function aborts the process, which is safe but abrupt, so boundary functions catch failure with `std::panic::catch_unwind` and return an error code instead. That works only when panics unwind: `catch_unwind` cannot catch a panic in a build with `panic = "abort"`, where the process ends at the panic. A library meant for foreign callers is built and tested with unwinding, or must not panic on any path the boundary reaches. Third, the generated bindings are build outputs with versions, not files people edit. The interface is a contract with another team, and Chapter 9 treats it as one.

Code that runs while you build

Two Rust features run code on the build machine. A `build.rs` script runs before its package compiles, often to find a C library or generate code. A procedural macro runs inside the compiler while your code compiles. Both run with the full rights of whoever starts the build: they can read files, read environment variables and reach the network. That makes them a supply-chain risk in the plain sense. A compromised dependency does not need to reach production to do harm; running on a developer laptop or a CI runner that holds credentials is enough.

The build machine is part of the attack surface. Build in CI runners that hold no long-lived secrets. Fetch dependencies in one step, then build with `--locked --offline` so Cargo itself does not reach the network. That flag governs Cargo only: it does not sandbox build scripts, proc macros or the processes they start, which can still open connections. Deny network access to the build step in the runner or container sandbox, and treat `--offline` as a second check, not the control. Keep an inventory of dependencies that have build scripts or are proc macros, and look at them first in every dependency audit.

One lifecycle for every exception

Unsafe modules, foreign calls, build scripts, lint waivers and emergency toolchain bumps look different, but they fail the same way: someone approves them once and nobody looks again. So the estate runs one lifecycle for all of them.

FIGURE 5.2

The exception lifecycle



An exception that cannot expire is a policy change in disguise. The loop back is deliberate: renewal is never automatic.

Each record is a few lines of text in the repository, so it travels with the code. The full template is in Appendix A; an excerpt:

```
[[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"
```

Decay review is the step most programs skip. Code around an exception changes, the owner moves teams, a new standard library API makes the unsafe unnecessary. So review is triggered by any of four events: the expiry date, a change to the file, a toolchain bump, or a change of owner. The CI gate reads the records and fails when one has expired, which turns forgetting into a red build instead of a quiet risk. The waiver authority depends on the kind: the crate owner for a lint waiver, the platform team for a toolchain bump, security for unsafe and build scripts in anything that faces untrusted input.

Contract card

Field	The exception contract
Contract	Every unsafe module, foreign boundary, build script and waiver is a signed, expiring exception
Threat addressed	Undefined behavior behind safe interfaces; memory and panic errors at FFI; malicious or careless build-time code
Owner	The named owner on each record; the platform team owns the lifecycle and the lints
Enforcement point	`unsafe_code`, `unsafe_op_in_unsafe_fn` and `undocumented_unsafe_blocks` lints; CI check of exception records; offline, secret-free builds
Failure behavior	Undocumented unsafe or an expired record fails the build
Waiver authority and expiry	By kind: crate owner, platform team or security; every record carries a date
Evidence the control ran	Lint results in CI; Miri and fuzz run logs per record; the expiry check output
Evidence the outcome improved	Fewer unsafe modules over time; memory-safety defects found before release rather than after
Review cadence	On each trigger (expiry, file change, toolchain bump, owner change), and a quarterly register review

CONTRACT CHECKLIST

- Crates without a need for unsafe use `forbid(unsafe_code)`; the rest deny it outside one named module.
- Every unsafe block has a `SAFETY` comment, enforced by lint, and every unsafe module has a register row with owner, reviewer and evidence.
- Every foreign boundary states who frees memory and converts panics to errors before they cross.
- Builds run without long-lived secrets and without network access after the fetch step.
- Every exception record has an expiry date, and the CI gate fails on an expired one.

CHAPTER SIX

The Verification Contract



A green pipeline is a claim, and most estates never check whether the claim is true. The compiler covers what Chapter 4 described. Everything else that the code door promises, from lint policy to the behavior of an unsafe module, is covered only if some check runs, fails when it should, and leaves a record. That is the verification contract: each promise has a check, each check is itself tested by a change that must fail it, and the estate keeps two kinds of evidence apart, proof that the control ran and proof that outcomes got better. The risk it removes is false confidence, the gate that has quietly stopped gating. The cost is CI time, a few nightly-only jobs on a schedule, and the habit of breaking your own pipeline on purpose.

If you are a CTO, ask your teams one question from this chapter: when did each gate last fail on a change that was planted to fail it? A gate with no answer is not yet evidence.

FIGURE 6.1

The verification ladder: what each rung finds

Technique	What it finds	Toolchain, when
Compiler	type, ownership, Send and Sync errors	stable, each build
Clippy and rustfmt	policy breaks and suspicious patterns	stable, every PR
Unit, integration, doc	the behavior someone thought to check	stable, every PR
Property tests	invariants across generated inputs	stable, every PR
Fuzzing (cargo-fuzz)	crashes on inputs nobody imagined	nightly, scheduled
Miri	undefined behavior in executed unsafe	nightly, scheduled
loom	bad thread interleavings, by model	stable, targeted

None of them proves the absence of bugs. Each leaves evidence that it ran.

Cost and depth rise down the ladder. Copper rows need the nightly toolchain, so they run as scheduled jobs on a pinned nightly, never as a merge requirement on whatever nightly is current.

Lint and format as written policy

Clippy is the standard Rust linter, and rustfmt the standard formatter. Treat both as policy, not taste. Formatting has no right answer, which is exactly why it should be settled once by a tool and never argued in review again. Lints are different: each level is a decision about risk. Clippy's correctness group is denied by default because those lints usually mean a bug. The style, complexity, performance and suspicious groups warn. The pedantic, restriction and nursery groups are off unless you opt in, and the restriction group holds the lints that encode estate rules, such as `unwrap_used` or `undocumented_unsafe_blocks` from the last two chapters.

Since Rust 1.74, Cargo reads lint levels from a `[lints]` table in `Cargo.toml`. That moves policy out of scattered attributes and command-line flags into one reviewed file. A workspace sets the policy once and each member opts in.

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

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

# each member crate's Cargo.toml
[lints]
workspace = true
```

The negative priority lets the whole group be set first and single lints override it. In CI, `-D warnings` turns the remaining warnings into failures, so the policy file decides what blocks a merge. Two governance notes. A change to this file is a policy change and is reviewed by the platform team, not merged by whoever finds a lint annoying. And new Clippy versions add lints, so a toolchain bump can turn a green build red overnight; that is one reason Chapter 7 pins the toolchain and treats each bump as a planned change.

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

These three lines are the core of the merge gate. The full CI template is in Appendix A.

Tests, and what each kind proves

Rust builds three kinds of tests into Cargo. Unit tests live next to the code and may test private functions. Integration tests live in the `tests/` directory and see only the crate's public interface, so they test the contract that callers depend on. Doc tests are the code examples in documentation comments of a library crate: `cargo test` compiles and runs them, so an example that drifts from the real API fails the build. That makes documentation part of the interface promise in Chapter 9.

All three share a limit: they check the cases someone thought of. Property tests widen that. Instead of one input, the test states a rule that must hold for every input, and a generator tries many of them. `proptest` is a widely used crate for this, and it shrinks a failing input to a minimal example before reporting it.

```
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));
    }
}
```

The first property is the library panic rule from Chapter 4, written as a test: no input reaches a panic. The second is the most common shape of all, a round trip, and it fits parsers, codecs and serialization formats.

Fuzzing, Miri and loom

Fuzzing goes further than property tests. A coverage-guided fuzzer watches which code paths each input reaches and mutates inputs to reach new ones, for hours or days. `cargo-fuzz` drives libFuzzer and requires the nightly toolchain. A target is a few lines:

```
#![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);
    }
});
```

Fuzzing belongs on every parser of untrusted input and every unsafe module in the register. It runs as a scheduled job, keeps its corpus between runs, and turns each crash it finds into a regular test so the bug cannot return.

Miri, from Chapter 5, runs the test suite in an interpreter that detects undefined behavior. It is nightly-only, it sees only executed paths, and it cannot run most foreign calls, so its evidence is scoped to the tests it ran. Where it cannot run the code, use fuzzing, sanitizers, model tests or targeted review instead. loom is a model checker for concurrent code: a test written against loom's versions of threads, atomics and locks runs many times, each time with a different interleaving, under the C11 memory model. It is the right tool for a hand-written lock-free structure or a custom synchronization primitive, and the wrong tool for an ordinary service; it finds bad interleavings in small, focused models. A clean run is evidence only for the model it explored: the code under test, with loom's versions of the primitives,

within the bounds the test sets (for example a limit on preemptions). It says nothing about code paths, thread counts or primitives outside that model. Neither tool finds deadlocks in business logic or logic races between services. Those stay with design review and with operations.

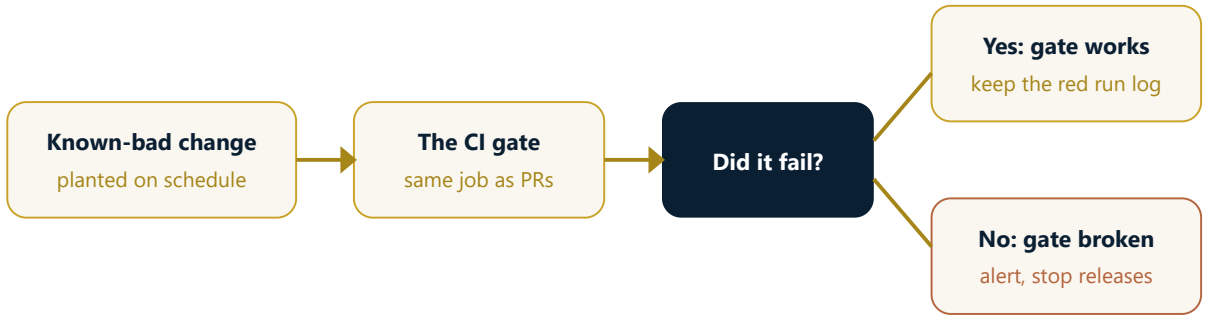
Check	Required where	Evidence it leaves
fmt, Clippy, tests	Every crate, every merge	CI run log with toolchain version
Property tests	Parsers, codecs, domain rules	Test names mapped to invariants
Fuzzing	Untrusted input, register entries	Hours run, corpus size, findings
Miri	Every unsafe register entry, where Miri supports the code; otherwise fuzzing, sanitizers, model tests or targeted review	Scheduled run log per crate
loom	Custom concurrency primitives	Model test results

A gate must itself be tested

Gates fail silently in boring ways. A job is marked optional during an outage and never switched back. A path filter skips the crate that changed. A lint is allowed at the top of a file and forgotten. The pipeline stays green, and green looks like evidence. The only reliable test of a gate is a change that must fail it.

FIGURE 6.2

Testing the gate with a known-bad change



A control that never fails is not evidence of anything.

The red run is the evidence. A scheduled job opens a change that breaks each rule on purpose and checks that the gate rejects it. If the gate passes it, the gate is the incident.

In practice, a small set of canary changes covers the rules that matter: an `unwrap()` on an input path in a library crate, an unsafe block without a `SAFETY` comment, a misformatted file, a failing test, an expired exception record, a dependency the admission gate should reject (Chapter 8). A scheduled job applies each one on a throwaway branch, runs the same gate as real merges, and records that it went red. The same idea applies to tests themselves. Mutation testing changes the code in small ways, such as flipping a comparison, and checks that some test fails; `cargo-mutants` is a community tool that does this for Rust. A surviving mutant points at a test suite that runs but does not check.

Two kinds of evidence

Evidence that the control ran is cheap and necessary: a CI log, a fuzz job's hours, a Miri run per crate, a canary that went red. It answers an auditor's first question. It does not answer the business question, which is whether the estate is better for it. That needs outcome evidence, measured against the baseline in the decision record from Chapter 2: defects that escaped to production, crashes from panics, findings caught before release compared with after, and the time from a finding to its fix. Coverage percentage is control evidence, not outcome evidence; a suite can cover every line and check nothing. I keep the two in separate columns on the Contract card for that reason. Teams that report only the first column end up optimizing activity. Teams that report only the second cannot show why the numbers moved.

> *The compiler decides what it can prove. For everything else, the contract decides only if its gate can still say no, and the estate has seen it say no recently.*

Contract card

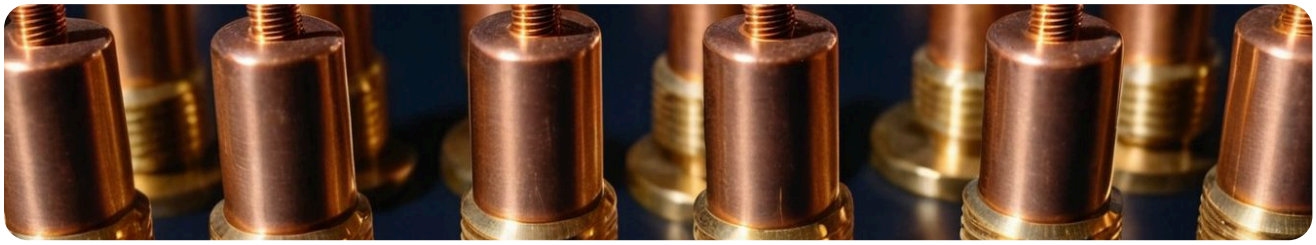
Field	The verification contract
Contract	Every code-door promise has a check; every gate is proven to fail on a planted bad change
Threat addressed	Gates that silently stopped gating; tests that run but do not check; undefined behavior and crashes on unexpected input
Owner	Platform team owns the gate, lint policy and canaries; crate owners own their tests, properties and fuzz targets
Enforcement point	Merge gate (fmt, Clippy, tests on stable); scheduled jobs (fuzz, Miri on a pinned nightly); scheduled canary run
Failure behavior	A failing merge check blocks the merge; a canary that passes the gate pages the gate owner and holds releases
Waiver authority and expiry	Platform team only, for a named check and a dated period, through the Chapter 5 lifecycle
Evidence the control ran	CI logs with toolchain version; fuzz hours and corpus; Miri logs; the red canary runs
Evidence the outcome improved	Escaped defects, panic crashes and time to fix, against the decision record baseline
Review cadence	Canaries on a fixed schedule; lint policy on each toolchain bump; outcome evidence quarterly

CONTRACT CHECKLIST

- Lint levels live in a workspace `[lints]` table, and changes to it are reviewed by the platform team.
- The merge gate runs fmt, Clippy with `-D warnings` and all tests on the pinned stable toolchain.
- Every parser of untrusted input has property tests and a scheduled fuzz target; every unsafe register entry has a scheduled Miri run where Miri supports the code, and otherwise fuzzing, sanitizers, model tests or targeted review.
- A scheduled canary job proves each gate rule still fails a known-bad change, and keeps the red run as evidence.
- Reports show control evidence and outcome evidence in separate columns, with the outcome measured against a baseline.

CHAPTER SEVEN

The Pin: Dependencies and Toolchain



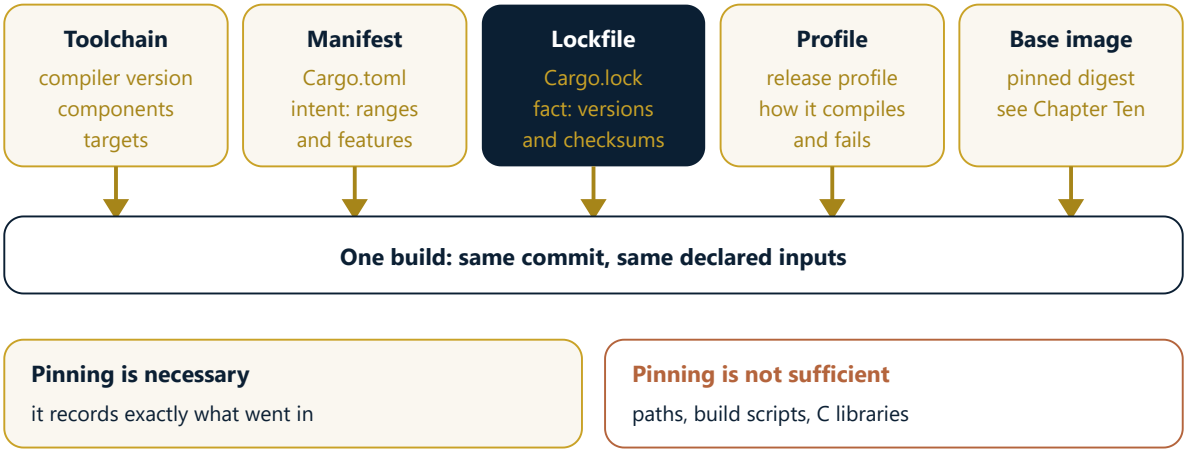
Most of the code inside a Rust service was not written by the team that ships it. It arrives through a manifest, is fixed in place by a lockfile, and is turned into machine code by a compiler version that someone chose, or forgot to choose. The pin is the contract that says which code and which compiler, exactly, and who may change them. Without it, two builds of the same commit can be two different products.

The short version for a CTO: commit the lockfile, pin the toolchain in a file, publish a short list of estate-standard crates, and treat a change to any of the three as a change to the product. That is cheap. A pin records what went in; whether it should have been let in is the admission gate in Chapter Eight.

The compiler proves nothing about versions; it checks whatever source it is given. Tooling enforces the pin: Cargo resolves and locks dependencies, rustup reads the toolchain file, and CI refuses a build whose lockfile is stale. People own the choices: which crates are standard, when the toolchain moves, and who may sign off on nightly.

FIGURE 7.1

Five pins, one build



Each pin is a file, so each change is reviewed. What the pins do not cover (copper) is why reproducibility must be tested, not assumed.

The lockfile is the fact

Cargo.toml states intent. A requirement such as `serde = "1"` means "any compatible 1.x release". Cargo.lock states fact: the exact version of every crate in the graph, direct and transitive, with a checksum for each one downloaded from a registry.

The old advice was to commit the lockfile for binaries and ignore it for libraries. In August 2023 the Cargo team changed that guidance: projects should do what is best for them, with committing the lockfile as the starting point, and `cargo new` stopped adding Cargo.lock to `.gitignore` for libraries (Rust Blog, 2023). One detail matters for libraries. Cargo uses only the lockfile of the top-level workspace being built, so a library's committed lockfile protects its own CI, not its consumers.

THE LOCKFILE CONTRACT

- Every repository commits Cargo.lock, binaries and libraries alike.
- CI builds with `--locked`, so a stale lockfile fails instead of silently resolving new versions.
- Dependency updates arrive as their own pull requests, so each one passes the admission gate.
- Libraries also test on a schedule against the latest compatible versions, as the Cargo team advises.

Workspaces: one place to change a version

A workspace is a set of crates that share one lockfile and one build directory. Since Rust 1.64, a workspace can declare dependencies once in `[workspace.dependencies]` and have members inherit them, and it can share metadata such as edition and rust-version through `[workspace.package]`. One version line, one review.

```
# Cargo.toml at the workspace root (excerpt; full template in Appendix 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
```

Set the resolver explicitly at the root. A virtual workspace (a root with no package of its own) has no edition to infer it from. Resolver "2" is the default for the 2021 edition. Resolver "3", the default for the 2024 edition, requires Rust 1.84 or later and makes resolution aware of each crate's rust-version (Cargo Book, resolver).

Features unify, and that is a governance hazard

Cargo features are compile-time switches. When several crates in one build use the same dependency, Cargo builds it once with the union of every feature anyone asked for. The Cargo Book says features should therefore be additive. One crate's request changes what every other crate in the same build receives.

So the feature set depends on what is built together. Resolver 2 stops unifying across build dependencies, proc macros, inactive targets and unbuilt dev dependencies, but packages built in one command still unify. A binary built alone with `-p` and the same binary built inside `cargo build --workspace` can carry different features. If the tested binary and the shipped binary are built differently, you tested a different product.

EXAMPLE: THE FEATURE NOBODY ASKED FOR

Consider a team that enables a verbose diagnostics feature on a shared logging crate for one internal tool. Release builds run across the whole workspace, so the feature quietly reaches the public service too, and no diff in the service showed it. The fix: build release artifacts per package, the same way in CI and release; `cargo tree -e features -i <crate>` shows who turned a feature on.

A dependency budget and estate-standard crates

Each direct dependency brings its own dependencies, maintainers, unsafe code and build scripts. A budget makes that cost visible: the platform team sets a ceiling of direct dependencies per service, and each new one needs a one-line reason in the pull request, including why an existing standard crate is not enough. `cargo tree --duplicates` shows crates built in more than one version.

The second half of the budget is an estate-standard list: one chosen crate per role, owned by the platform team. Typical roles are the async runtime, HTTP client and server, serialization, TLS, error handling and tracing. The async runtime matters most, because it is not a library you call but a scheduler your code runs inside. Two runtimes in one service means two schedulers and confusion about who owns a task. Tokio is the common choice for network services, and widely used crates such as hyper, axum and tonic are built on it. smol is a smaller alternative. async-std was discontinued by its maintainers in 2025, who recommend smol (RustSec RUSTSEC-2025-0052, 2025). That is the kind of change the list exists to absorb: one owner acts once.

The toolchain file and the nightly rule

A `rust-toolchain.toml` file at the repository root tells rustup which compiler to use. It is overridden only by an explicit choice on the command line, the `RUSTUP_TOOLCHAIN` variable or a directory override (rustup book, overrides).

```
# rust-toolchain.toml (example pin; bump on the estate cadence)
[toolchain]
channel = "1.100.0"
components = ["rustfmt", "clippy"]
targets = ["x86_64-unknown-linux-musl", "aarch64-unknown-linux-gnu"]
profile = "minimal"
```

Stable Rust ships every six weeks. Pin an exact stable version and move it on a set cadence, as an ordinary pull request that runs the full gate. The estate pin may therefore lag the newest stable release by up to one cadence step; that is deliberate, and a security fix in the

compiler or standard library is the reason to use the emergency bump path instead of waiting. Move the lockfile and the toolchain in separate pull requests, so a failure points at one cause.

Unstable features need a nightly compiler, and nightly changes daily. My rule: nightly only with a named owner, a written reason, a dated pin such as `nightly-2026-11-01` and an exit condition, usually "when the feature reaches stable". The `RUSTC_BOOTSTRAP` environment variable lets a stable compiler accept unstable features; CI should reject builds that set it outside the approved exception. This is the Chapter Five exception lifecycle, applied to the compiler.

MSRV and editions

The minimum supported Rust version (MSRV) is a promise to consumers. Declare it in `rust-version` and Cargo refuses older compilers. Since Rust 1.84 the resolver can prefer dependency versions whose own rust-version fits yours; resolver "3" makes that the default (Cargo Book, resolver). Only a CI job on that exact compiler proves the MSRV: `cargo +1.85.0 check --workspace --locked`. Raising it is possibly breaking under the Cargo SemVer guide, so Chapter Nine makes it part of the interface promise.

Editions are opt-in language revisions chosen per crate, and crates on different editions link together, so migration with `cargo fix --edition` can go one crate at a time. The 2024 edition has been stable since Rust 1.85. New crates start on the current edition; existing crates migrate on a dated plan.

Release profiles are a contract about failure

Some release settings change behavior, not only size and speed.

```
[profile.release]
lto = true            # whole-program ("fat") optimization; slower builds
codegen-units = 1     # better optimization; less parallel compilation
strip = true          # strip symbols from the shipped binary
panic = "abort"       # no unwinding; the process ends on panic
overflow-checks = true  # stated, not defaulted: panic instead of wrapping
```

`panic = "abort"` means a panic ends the process at once: no unwinding, no destructors, and `catch_unwind` cannot catch it. Tests ignore the setting, because the test harness needs unwinding (Cargo Book, profiles). Integer overflow wraps in release builds by default; overflow-checks turns silent overflow into a panic. Defaults have also moved: since Rust 1.77, release builds without debug info strip debuginfo by default, except on Windows MSVC

targets, where Rust 1.77.1 turned the change off (Rust Blog, 2024). Set strip explicitly so the contract does not depend on a default. For what these settings do to size and speed, see the reference service in Appendix A.

The limits of reproducibility

Pinning is necessary, not sufficient. The lockfile pins Rust sources by checksum. It does not pin C libraries from the system, build scripts that read the environment or download files, or absolute paths embedded in panic messages and debug info unless remapped with `--remap-path-prefix`. A profile option to trim paths is still unstable at the time of writing. Do not claim bit-for-bit reproducible builds until you have built the same commit twice, on two machines, in two different directories, and compared the hashes. If they differ, the honest claim is "pinned inputs".

> *The developer proposes a version. The lockfile records it. The gate in the next chapter decides whether it may come in.*

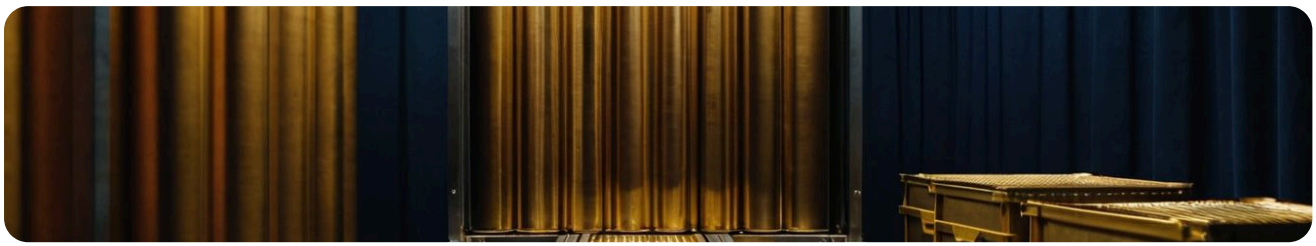
Contract card	The pin
Contract	Every build uses a committed lockfile, a pinned toolchain, an explicit release profile and only estate-standard crates for standard roles.
Threat addressed	Silent drift: versions, compilers or features reaching production unreviewed.
Owner	Platform team owns the toolchain pin, the standard-crate list and the profile template. Product teams own their lockfiles.
Enforcement point	CI: `--locked` builds, toolchain file present, MSRV job, rejection of `RUSTC_BOOTSTRAP`, budget check on new direct dependencies.
Failure behavior	Fail closed: a stale lockfile, a missing toolchain file or an unapproved nightly blocks the merge.
Waiver authority and expiry	Platform lead for nightly and off-list crates, with a named owner, reason and expiry of at most one quarter (example estate policy).
Evidence the control ran	CI logs with `--locked` and toolchain version per build; the nightly register.
Evidence the outcome improved	Fewer duplicate crate versions over time; no production incident traced to an unreviewed version change.
Review cadence	Toolchain bump on each set cadence; standard-crate list quarterly; nightly exceptions monthly.

CONTRACT CHECKLIST

- Cargo.lock is committed in every repository, and CI builds with `--locked`.
- A rust-toolchain.toml pins an exact stable version, and nightly use has an owner, reason, dated pin and exit condition.
- The workspace sets its resolver explicitly and inherits dependency versions from one table.
- Release artifacts are built per package, the same way in CI and release, and feature sets are inspectable.
- The rust-version field is declared and proved by a CI job on that exact compiler.
- The release profile sets panic, strip and overflow behavior explicitly; reproducibility claims are tested first.

CHAPTER EIGHT

The Admission Gate



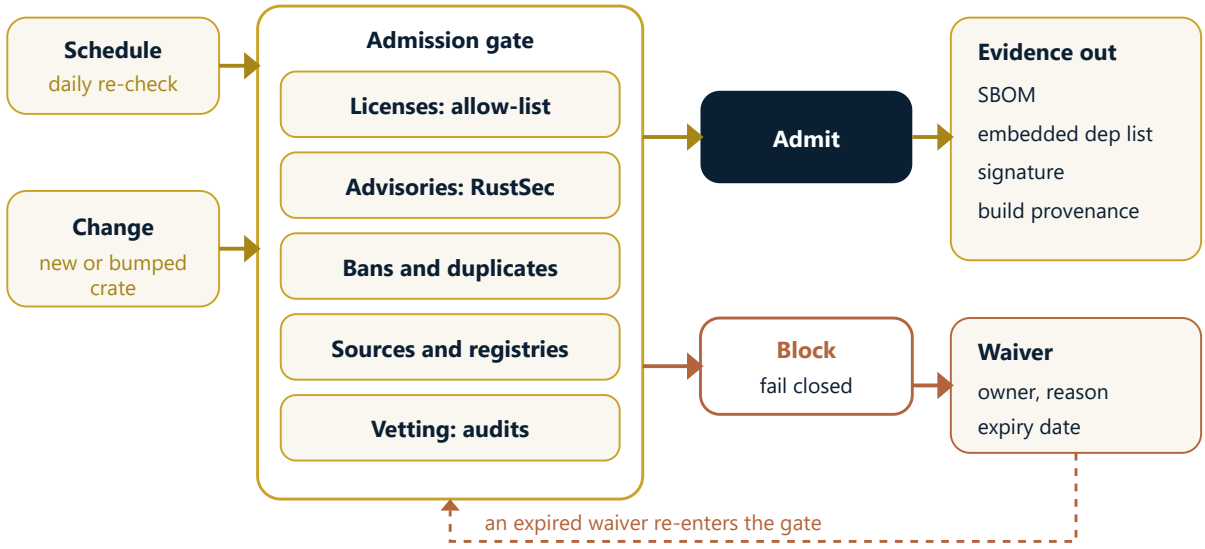
A lockfile records what entered the build. It never asks whether it should have. Every crate that enters is code that will run in production, and through build scripts and proc macros, some of it runs on the build machine before production ever sees it. The admission gate is the contract that answers the question at the door: may this code come in, under which rules, and who said yes?

For a CTO, the gate is five questions asked automatically on every change and again every day: is the license acceptable, is the version known to be vulnerable, is the crate banned or duplicated, did it come from an approved source, and has someone the estate trusts reviewed it. A failing answer blocks the merge. A sanctioned exception has an owner and an expiry date. And the gate produces evidence on the way out: a bill of materials, a signature and a record of how the artifact was built.

The compiler proves almost nothing at this door. Safe Rust inside a crate is memory safe, but the crate may contain unsafe code, call C, or run arbitrary code at build time (Chapter Five). Tooling enforces the rules. People own the rules themselves: legal owns the license list, security owns advisories and waivers, and the platform team owns sources, bans and the gate's own health.

FIGURE 8.1

The admission gate, its exits and its evidence



The gate runs twice. Once when a dependency changes, and again every day, because advisories are published after code has been admitted. Blocks go to a waiver with an expiry, never to a silent pass.

The five questions as written rules

Question	Rule	Owner	Common instance
License	Only licenses on the allow-list; unknown means no	Legal	cargo-deny licenses
Advisories	No known vulnerability; unmaintained and unsound flagged	Security	cargo-deny advisories, cargo-audit
Bans	No banned crates, no wildcard requirements, duplicates justified	Platform	cargo-deny bans
Sources	Approved registries only; git sources by exception	Platform	cargo-deny sources
Vetting	Each version reviewed internally or by an auditor the estate trusts	Security with product teams	cargo-vet

cargo-deny covers four of the five questions from one file, `deny.toml`, and exits with an error when a check fails, which is all a CI gate needs. An excerpt, tested with cargo-deny 0.20.2, which rejects unknown keys and values at load time; older releases wrote `name` and `version` where it writes `crate`:

```
# deny.toml for cargo-deny 0.20.2 (excerpt; full template in Appendix 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"   # the default is "warn"; deny, and skip with a reason
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"
```

Licenses and advisories

A license is a legal promise, so the allow-list is written by legal and expressed in SPDX identifiers. A crate whose license cannot be detected with confidence fails. Exceptions are per crate, with a reason, never a widening of the list.

Advisories come from the RustSec Advisory Database, a community database of security advisories for crates published on crates.io. cargo-audit and the advisories check in cargo-deny both read it. Current cargo-deny versions treat every vulnerability advisory as an error; the configuration decides how unmaintained and unsound advisories are scoped and whether yanked versions fail (cargo-deny docs). Run the check on every pull request, and run it daily against the main branch and against the lockfile of every release still deployed. An advisory published today applies to the binary you shipped last month.

Bans, duplicates and sources

Bans encode estate decisions: one TLS stack, one runtime, no crate the security team has rejected. The `use-instead` field turns a ban into guidance. Duplicate versions of the same crate mean larger binaries, longer builds and more code to vet. Denying duplicates is the

goal; a `skip` entry with a reason is the honest way to live with one while an upstream catches up. Wildcard version requirements are banned outright, because they make the manifest promise nothing.

Sources answer where code may come from. The rule I recommend is crates.io, or the estate's private registry or mirror, and nothing else by default. A git dependency is code that bypassed every registry check, so it needs an exception with an owner, a pinned revision and an exit plan.

Vetting: someone you trust has read it

The checks above are about metadata. Vetting is about the code. cargo-vet records, for each crate version, that a person has audited it against stated criteria. Two criteria are built in: `safe-to-run`, meaning the crate can be built, run and tested on a workstation or in controlled automation without surprising consequences, and `safe-to-deploy`, meaning it will not introduce a serious security vulnerability into production software exposed to untrusted input; the second implies the first (cargo-vet book). Audits are stored in the repository, and audits of a version diff let a reviewer read only what changed.

No estate can audit everything alone, so cargo-vet can import the audit sets other organizations publish. Its registry lists, among others, Mozilla, Google and the Bytecode Alliance. Importing an audit is a trust decision, not a technicality: the platform team decides whose audits count, and records that decision in the configuration.

```
# supply-chain/config.toml (excerpt)
[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"
```

On first adoption, every existing dependency lands in an exemptions list. That list is the vetting debt, written down. The contract is that it only shrinks: new crates need an audit or an imported one, and the exemptions count is reported every month.

One door to the outside: mirror or vendor

A gate is easier to hold when there is only one door. Two patterns work. A private registry mirror makes the estate's builds fetch only from a source the platform team controls. Vendoring, with `cargo vendor`, copies every dependency's source into the repository and prints the source-replacement configuration that tells Cargo to use it. Both survive a registry

outage and the removal of a crate upstream. Vendoring also puts third-party changes in the diff, at the cost of repository size. Either way, the gate runs before code reaches the mirror or the vendor directory, not after.

Evidence on the way out: SBOM, signatures, provenance

An SBOM (software bill of materials) lists what an artifact contains. cargo-cyclonedx generates a CycloneDX SBOM from the lockfile and Cargo metadata. It reflects the features and target of the shipped build only when you pass it the same feature and target arguments as that build. cargo-auditable takes a different approach: it embeds a compressed list of the binary's dependencies in a section of the binary itself, so cargo-audit, Trivy, Grype and Syft can scan a shipped binary without its source (cargo-auditable docs). The project is clear that this is visibility, not protection against supply-chain attacks. Use both: the SBOM travels with the release, and the embedded list cannot be separated from the binary.

A signature says who released the artifact. Provenance says how it was built. Sigstore's cosign signs container images and attaches attestations such as an SBOM, and its keyless mode ties the signature to a workload identity from an OIDC provider and records it in a public transparency log. SLSA describes provenance in levels: at Build L1 provenance exists, at L2 it is signed and produced on a hosted build platform, at L3 that platform is hardened so build steps cannot reach the signing secret (SLSA v1.0, 2023). A deploy that verifies none of this has received the evidence and ignored it; Chapters Ten and Eleven close that loop.

```
# Same target and features as the shipped build
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"

# Build provenance, GitHub Actions. The job needs:
#   permissions: { id-token: write, attestations: write, packages: write }
#   (packages: write only because push-to-registry is true)
- uses: actions/attest@v4            # no predicate or SBOM input: SLSA build provenance
  with:
    subject-name: registry.example.com/service
    subject-digest: ${{ steps.build.outputs.digest }}
    push-to-registry: true
# At deploy: provenance must come from the expected repository and workflow
gh attestation verify "oci://$IMAGE@$DIGEST"   --repo org/service   --signer-workflow
org/service/.github/workflows/release.yml
```

The provenance step follows GitHub's artifact attestations documentation (2026); with no SBOM or predicate input, `actions/attest` produces a SLSA build provenance attestation. `actions/attest-build-provenance@v4` is now a wrapper over it, so existing workflows that use it keep working, and GitHub recommends `actions/attest` for new ones. The

verify flags are those `gh attestation verify --help` lists (gh 2.81): `--repo` or `--owner` scopes the lookup, and `--signer-workflow` pins the workflow that signed. A note on the `skip` entry above: in cargo-deny a crate spec takes `name@version` for an exact version and `name:requirement` for a version requirement, so `syn:<3` means "any syn below 3" (checked with 0.20.2). Elsewhere, an SLSA provenance generator verified with `slsa-verifier`, or the CI platform's own attestation service, fills the same role.

Failure behavior and the waiver path

The gate fails closed. A failing check blocks the merge, and a gate that cannot reach its advisory database fails rather than passing on stale data. Severity changes the response, not the rule: a new critical advisory against a deployed release opens an incident, while an unmaintained notice opens a ticket with a due date.

Waivers are where gates are usually lost. The authority is split by question: legal waives licenses, security waives advisories and vetting, the platform lead waives bans and sources. Every waiver carries an owner, a reason, a compensating control and an expiry. As an example estate policy, I use a ceiling of one quarter, renewable once with fresh approval; set your own and write it down. In cargo-deny 0.20.2 an advisory ignore entry takes an `id` and a `reason` (a yanked version can instead be named by crate spec) and has no expiry field, so the expiry lives in the register, not in `deny.toml`. Record the waiver in a register and reference its identifier in the `ignore` or `skip` entry, as in the excerpt above, and let a CI job fail when a register entry passes its date.

Prove the gate blocks

A gate that has never blocked anything is a gate nobody has tested. Keep one fixture per rule, each breaking exactly one thing: a banned crate, a disallowed license, a git source, a version with a known advisory. Keep one clean control fixture as well, which must pass. A scheduled job runs the gate against each one. A plain "did it fail" test is not enough, because a missing tool, a bad configuration or a failed database fetch also fails; the job must check that each bad fixture fails for its own reason, by matching the exact `code` of an error diagnostic in cargo-deny's JSON output. This is the rule from Chapter Six: a control must be tested to prove it blocks.

```
# gate-canary.sh: the control must pass; each bad fixture must fail for its own reason
set -u
run() { cargo deny --locked --format json --manifest-path "fixtures/$1/Cargo.toml" check $2 2>&1;
}
errors() {  # one line per error: its diagnostic code, or tool-error for a config, fetch or tool
failure
  python3 -c '
import json, sys
for line in sys.stdin:
    try: d = json.loads(line)
    except ValueError: continue
    f = d.get("fields", {})
    if d.get("type") == "log" and f.get("level") == "ERROR": print("tool-error")
    elif d.get("type") == "diagnostic" and f.get("severity") == "error": print(f.get("code"))'
}
run clean "" >/dev/null || { echo "GATE BROKEN: clean control failed" >&2; exit 1; }
check_blocks() {  # fixture, check, expected code
  out=$(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"
```

The codes are the ones cargo-deny 0.20.2 emitted, as JSON lines on standard error, for fixtures built this way: `banned`, `rejected`, `source-not-allowed` and `vulnerability`. A configuration error, an advisory database that cannot be fetched or a missing manifest shows up as an error log line instead, and the script reports it as a broken tool, not as a block; the exit code alone cannot tell these apart. The script was run against the four fixtures and a clean control, and against a broken configuration, a missing ban and an unreachable database, each reported correctly. Confirm the codes when you bump the pinned cargo-deny version. Keep the fixtures inside the repository: cargo-deny 0.20.2 looks for `deny.toml` from the fixture's directory upward, so every fixture is judged by the estate's own file. If the repository root is a Cargo workspace, give each fixture an empty `[workspace]` table so Cargo treats it as its own root.

> *The developer proposes a dependency. The gate decides, in writing, every day, and a person with authority signs every exception.*

Contract card	The admission gate
Contract	No crate enters the build unless its license, advisories, ban status, source and audit status pass written rules; every release carries an SBOM, a signature and provenance.
Threat addressed	Vulnerable, malicious, unlicensed or unreviewed third-party code reaching production; artifacts of unknown origin.
Owner	Platform team owns the gate. Legal owns the license list. Security owns advisories and vetting.
Enforcement point	Pull request CI, a daily scheduled run on main and on deployed lockfiles, and signature verification at deploy.
Failure behavior	Fail closed, including when the advisory database cannot be fetched. Critical advisories on deployed releases open an incident.
Waiver authority and expiry	Legal for licenses, security for advisories and vetting, platform lead for bans and sources; example estate policy: maximum one quarter, renewable once.
Evidence the control ran	Gate output stored per build; daily run history; the canary job passing its control and failing each fixture for the expected reason; SBOM and attestation per release.
Evidence the outcome improved	Vetting exemptions shrinking month over month; time from advisory publication to fixed release falling.
Review cadence	Waiver register monthly; allow-lists, bans and imported audit sources quarterly.

CONTRACT CHECKLIST

- One configuration file encodes license, advisory, ban and source rules, and CI fails on any error.
- Advisory checks run on every pull request and daily against main and every deployed release.
- Vetting is recorded, imported audit sources are a written trust decision, and the exemptions list only shrinks.
- Every release has an SBOM, an embedded dependency list, a signature and provenance, and deploys verify them.
- Every waiver has an owner, a reason, a compensating control and an expiry, and expired waivers fail the build.
- A canary job proves, on a schedule, that a clean control passes and that each known-bad fixture fails with its expected diagnostic.

CHAPTER NINE

The Interface Promise



Inside one build, the compiler checks every call against every signature. Across a release boundary it cannot. When another team depends on your crate, or a TypeScript client depends on your service, the only thing that crosses is a promise: this version will not break you. The version number is the claim. The interface promise is the contract that makes the claim true, checks it before release, and gives callers time when it must change.

For a CTO: every public interface an estate exposes, whether a library, an API or a generated client, has an owner, a version that follows written rules, an automated check that the version tells the truth, a review that reads the interface diff and not only the code diff, and a deprecation clock. Breaking changes are allowed. Surprise breaking changes are not. This is where "Every line holds" is tested under change, and the honest version of the slogan is narrower: every line holds the promise its version number made, as far as the checks can see.

The tiers again. The compiler proves that the code compiles against the interface as it is now. Tooling enforces the comparison with what was promised before: semver checkers, schema diffs, regenerated bindings. People own what is public in the first place, what counts as a breaking change in behavior, and when a major release is worth its cost.

What the public API is

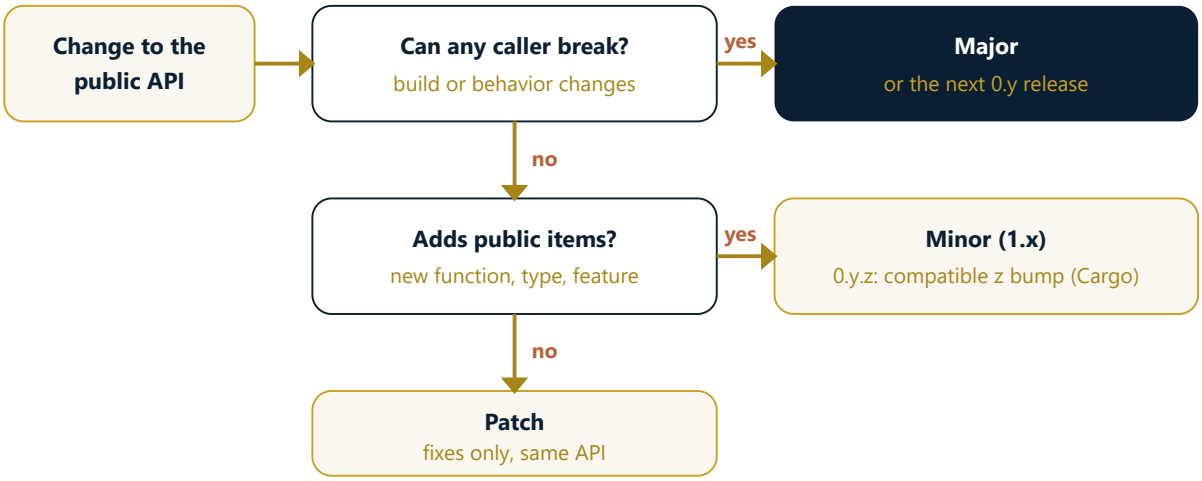
The public API of a Rust crate is larger than its `pub fn` list. It includes public types and their fields, enum variants, trait definitions and implementations, the auto traits a type happens to have (a type that stops being `Send` breaks every caller who moved it across threads), the crate's Cargo features and its minimum supported Rust version. Behavior is part of it too, and no tool sees all of behavior. The cheapest governance is a smaller surface: default to `pub(crate)`, and make each new public item a decision.

SemVer as Cargo reads it

Cargo treats a plain requirement such as `"1.4"` as a caret requirement: any version from 1.4.0 up to, but not including, 2.0.0. Below 1.0 the rule shifts. Cargo treats only the left-most non-zero number as the compatibility boundary, so in 0.y.z a breaking change bumps y, a compatible addition bumps z, and in 0.0.z every change is treated as breaking (Cargo Book, SemVer compatibility). Note the vocabulary: on 0.y.z the compatible release is a z bump, which Cargo accepts as compatible, even though SemVer itself calls z the patch number. Many widely used crates live at 0.x for years, which is fine, but an estate should know that a 0.4 to 0.5 move is a breaking upgrade by definition. For internal libraries, my rule is to publish 1.0 once a second team depends on the crate. The number then means what readers expect.

FIGURE 9.1

Which number moves



Cargo on 0.y.z: a y bump breaks, a z bump stays compatible. In 0.0.z, every bump breaks.

The first question decides most cases. "Can any caller break" includes behavior, which no tool fully checks. When in doubt, the change is major, or it ships behind a new name and the old one is deprecated.

Additive and breaking, by the book

The Cargo Book's SemVer compatibility chapter is the reference an estate should adopt rather than rewrite. A few of its rules decide most reviews:

Change	Cargo guide
Adding new public items	Minor (glob imports can rarely conflict)
Adding a new Cargo feature	Minor
Adding an enum variant, enum not non_exhaustive	Major
Adding a public field when no private field exists	Major
Adding a trait item with no default	Major
Removing a Cargo feature	Major
Adding non_exhaustive to an existing enum, variant, or struct with no private fields	Major
Raising the minimum supported Rust version	Possibly breaking; the guide suggests treating it as minor

The last row is a policy choice, so write it down. My recommendation: an MSRV bump is a minor release with a release note, never a patch, and the MSRV stays at or behind the estate's pinned toolchain from Chapter Seven.

non_exhaustive: room to grow, at a price

The `#[non_exhaustive]` attribute tells other crates that more variants or fields may come. Callers outside the crate must add a wildcard arm when they match, and cannot build the struct with a literal. Adding a variant later becomes a minor change.

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

// In a calling crate, the wildcard arm is required:
match status {
    OrderStatus::Pending => hold(order),
    OrderStatus::Paid | OrderStatus::Shipped => release(order),
    _ => escalate(order), // future variants land here, by contract
}
```

Notice the trade. Chapter Four praised exhaustive matching: add a variant, and the compiler finds every place that must handle it. `non_exhaustive` gives that check up at the boundary in exchange for freedom to evolve. Inside the defining crate the attribute has no effect. Choose deliberately: for a published library with unknown callers, mark enums that will grow. For crates in one workspace, where every caller changes in the same pull request, leave them exhaustive and let the compiler do the work. Decide early, because adding the attribute later is itself a breaking change.

Check the claim in CI

cargo-semver-checks compares the current crate against a baseline, by default the previous release on crates.io, or a git revision, and reports changes that break SemVer. It exits with one code for violations and another when it could not run, so CI can tell a broken promise from a broken check (cargo-semver-checks docs). It does not catch everything yet. Its maintainers list gaps such as changes to a field's or parameter's type, some generics and lifetime changes, and breakage that appears only with certain features. Custom registries are not supported as a baseline, so private crates compare against a git revision. Pick the baseline by purpose. The release gate compares against the last released version, because that is what consumers have: `--baseline-version` for a crate on crates.io, or `--baseline-rev` with the last release tag for a private crate, selected per crate by a tag pattern, with prereleases excluded; a first release has no baseline, so CI says so and skips (the full job is in Appendix A). A comparison against `main` is useful feedback on a pull request, but it is not the release gate, since `main` may already hold unreleased breaks.

```
# Release gate: compare with this crate's last stable release tag (private crate)
TAG=$(git describe --tags --abbrev=0 --match 'orders-client-v*' --exclude 'orders-client-v*-*'
HEAD^)
cargo semver-checks --package orders-client --baseline-rev "$TAG"   # e.g. orders-client-v1.4.2
# or, for a crate published on crates.io, the last released version
cargo semver-checks --package orders-client --baseline-version 1.4.2

# Pull request feedback only: compare with the target branch
cargo semver-checks --package orders-client --baseline-rev origin/main
```

A tool that catches most breaks still needs a human. Make the interface diff part of review: a listing of the public API before and after, produced by a tool such as cargo-public-api (it needs a nightly toolchain installed to generate rustdoc JSON, so the nightly rule from Chapter Seven applies). A pull request labeled breaking needs the interface owner's approval, a major version, and a migration note in the changelog.

Deprecation has a clock

Removing something is a major change. Announcing it should not be. The `#[deprecated]` attribute makes every caller's build warn, with a version and a note.

```
#[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())
}
```

THE DEPRECATION CONTRACT

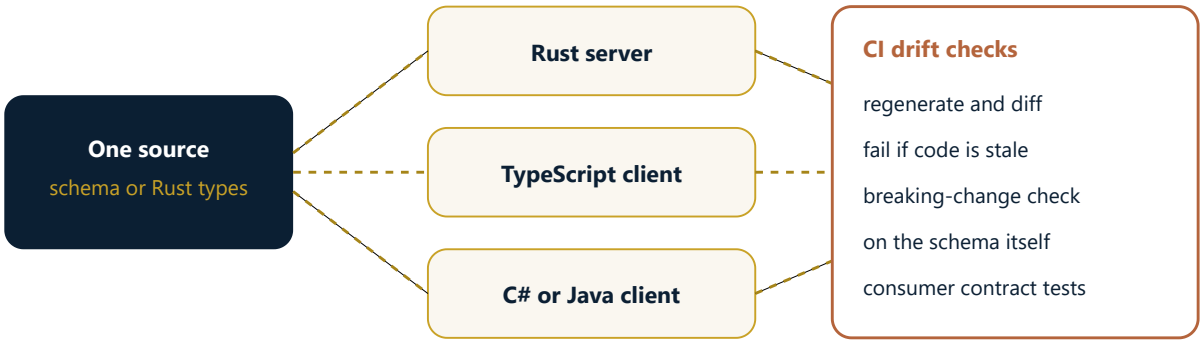
- Deprecate in a minor release, with the replacement already available.
- Keep the deprecated item for a stated minimum, for example two minor releases or one quarter, whichever is longer.
- Remove only in a major release, with a migration note.
- Record each deprecation, its owner and its removal date in the interface register, and tell known consumers directly.

Contract drift across languages

The hardest promises cross languages. A Rust service serves a TypeScript front end, a C# integration and a Java batch job. The Rust types change, the compiler is satisfied, and the clients break at run time, because nothing compiled them together. That is contract drift, and the fix is structural: one source of truth, generated everything else, and a CI check that fails when they disagree.

FIGURE 9.2

One source, generated edges, checked drift



Drift is a missing check, not a missing document. Whether the schema or the Rust types are the source, every other edge is generated from it, and CI proves the generated code is current.

There are two sound designs. Schema first: the interface is written in OpenAPI or Protocol Buffers, and the Rust server and every client are generated from it. Code first: the Rust types are the source, a schema is generated from them and committed, and the diff of that file is reviewed like code. Either way, the schema gets its own breaking-change check; for Protocol Buffers, `buf breaking` compares a schema against a previous version. For in-process boundaries the same rule holds with different generators: cbindgen writes C headers, wasm-bindgen writes JavaScript glue, UniFFI writes Kotlin, Swift and Python bindings. Generated files are regenerated in CI and the build fails if they differ from what is committed. Chapter Five covers the safety side of those boundaries; this chapter covers their stability.

Docs and doc tests are the readable contract

Callers read the documentation, not the type signatures. Rust makes that part checkable: code examples in doc comments compile and run as tests under `cargo test` for library crates, so an example that drifts from the API fails the build.

```
/// Parses an order id such as "ORD-1042".
///
/// # Errors
/// Returns `ParseError::Format` if the prefix is missing.
///
/// ```
/// use orders_client::OrderId;
/// let id: OrderId = "ORD-1042".parse()?;
/// assert_eq!(id.number(), 1042);
/// # Ok::<(), orders_client::ParseError>(())
/// ```
```

Set `missing_docs` to warn or deny on published crates, and use Clippy's lints for missing Errors, Panics and Safety sections, so the contract names what can fail. Documentation that states behavior is also the only place behavior is promised in writing, which matters for the cases no tool checks.

> *The developer proposes a change. The compiler decides whether it builds. The interface promise decides whether it ships as a patch, a minor or a major, and who must be told.*

Contract card	The interface promise
Contract	Every public interface has an owner and a version that follows the Cargo SemVer rules; breaking changes are major, announced and documented.
Threat addressed	Surprise breakage of consumers; cross-language drift that compiles but fails at run time.
Owner	The owning product team for each interface; the platform team owns the checks and the register.
Enforcement point	Pull request CI: semver check, regenerate-and-diff, schema breaking-change check, doc tests; interface owner approval on breaking changes.
Failure behavior	A detected break without a major version blocks the merge; a check that could not run also blocks.
Waiver authority and expiry	Interface owner with the platform lead, for one named release only, with consumers notified in writing.
Evidence the control ran	Semver and drift check results per pull request; the deprecation register; changelogs with migration notes.
Evidence the outcome improved	Fewer consumer incidents traced to interface changes; deprecations removed on schedule.
Review cadence	Interface register and deprecation clock quarterly; checker version and known gaps at each toolchain bump.

CONTRACT CHECKLIST

- Each public interface has a named owner and appears in the interface register.
- CI runs a semver check against the last release, and its known gaps are covered by human review of the API diff.
- Enums and structs meant to grow are marked non_exhaustive before 1.0, as a deliberate choice.
- Deprecations carry a version, a replacement and a removal date, and removal happens only in a major release.
- Cross-language edges are generated from one source, and CI fails when generated code is stale or the schema breaks.
- Doc examples run as tests, and published crates document their errors, panics and safety conditions.

CHAPTER TEN

Shipping the Artifact



Everything the earlier chapters proved, pinned and admitted ends up in one file. A binary or an image is the thing that actually runs, and the compiler has no say over how it is built, where it is copied, or which version a machine ends up holding. That makes the artifact a new door with its own contract: built once, from pinned inputs, signed, promoted by digest, and always able to go back to the version before it.

If you stop reading here, keep one rule. **Never rebuild to promote.** The bytes you tested are the bytes you ship. Every other rule in this chapter protects that one, for desktop tools, command line tools, daemons and container images alike.

Where the three tiers sit at this door

The compiler proves things about the code, not the shipping. A memory-safe binary built on a laptop and copied by hand to a server is still memory-safe, and still ungoverned.

- **The compiler proves** what it proved before: memory safety and data-race freedom in safe Rust, inside the trusted boundary from Chapter One.
- **Tooling enforces** how the artifact is built: the pinned toolchain from Chapter Seven, `--locked` builds, pinned base images, signatures, SBOMs and provenance from Chapter Eight.
- **Operations and people own** where it goes: release channels, rollout pace, the decision to roll back, and the CPU and operating system the binary meets in the field.

Desktop tools, command line tools and daemons

A single native binary, with no garbage collector or virtual machine to install, is one of the reasons teams choose Rust, and Go teams know the same benefit well. It may still link the C runtime or other system libraries, so call it self-contained only after inspecting its dynamic

dependencies on each target (for example with `ldd` on Linux). It is also a responsibility. Nothing installs it for you, so the release process has to do what a package manager or a platform would otherwise do.

Code signing. Each operating system has its own trust chain. Windows uses Authenticode signatures, macOS expects a Developer ID signature and notarization by Apple before Gatekeeper lets users open the binary without warnings, and Linux estates usually sign packages for their distribution or sign the release files with Sigstore. Treat signing keys as tier-three assets: held in a hardware module or a managed signing service, used only by the release pipeline, never on a developer machine.

Cross-compilation targets. Rust names every platform by a target triple, such as `x86_64-pc-windows-msvc`, `aarch64-apple-darwin` or `x86_64-unknown-linux-musl`, and `rustup target add` installs the standard library for each. The Rust project publishes target tiers: Tier 1 targets are guaranteed to work, Tier 2 targets are guaranteed to build. Write the list of supported targets into the contract and check each target's tier before you promise it to users. Linkers and C toolchains for foreign targets are the usual friction; tools such as `cross`, which builds inside containers, or `cargo-zigbuild` reduce it.

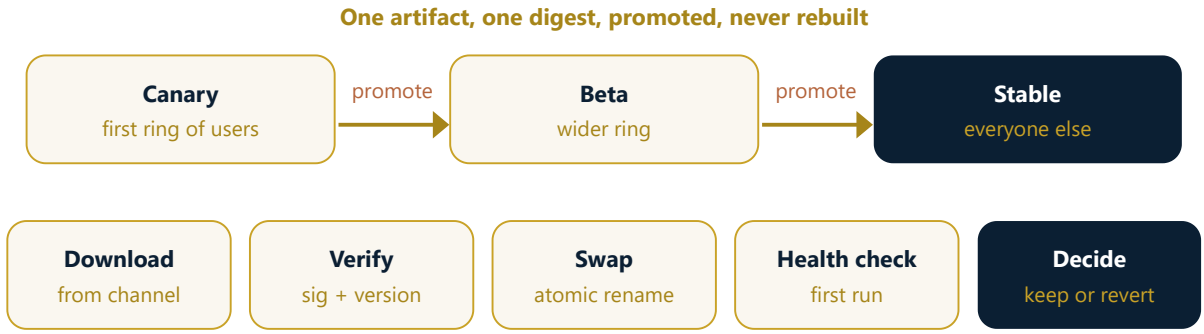
```
# Supported release targets (excerpt; full CI matrix in Appendix A)
rustup target add x86_64-unknown-linux-gnu aarch64-unknown-linux-gnu
rustup target add x86_64-pc-windows-msvc aarch64-apple-darwin
cargo build --release --locked --target aarch64-unknown-linux-gnu
```

Self-update with rollback. A tool that updates itself is a deployment system in miniature, and it needs the same contract as one. The update must verify a signature and a version before it touches anything. The swap must be atomic, which on most systems means writing the new file beside the old one and renaming it into place. The previous binary must stay on disk until the new one has started and passed a health check. If the check fails, the tool goes back without asking the user. Add a minimum supported version the server can enforce, and a way to pause updates for everyone.

Versioned release channels. A channel such as canary, beta or stable is a pointer to a version, not a separate build. Promotion moves the pointer. The artifact, and its digest, stay the same.

FIGURE 10.1

Channels promote one artifact; self-update can always go back



The previous binary stays on disk until the new one passes its health check.

A channel is a pointer, not a build. Promotion moves the pointer to the same digest. The client side of the contract is the five steps below it, and the last step can always choose the old version.

Container images

For services, the artifact is an image. The contract is the same: pinned inputs, a small surface, a known user, an exact reference.

Multi-stage builds. The build stage holds the toolchain, the source and the build cache. The runtime stage holds the binary and almost nothing else. Compilers and shells stay behind, out of the image an attacker would land in.

Dependency-layer caching. Rust builds spend most of their time compiling dependencies, and a naive Dockerfile recompiles all of them whenever any source file changes. `cargo-chef` splits the build in two: a planner stage writes a recipe of the dependency graph, and a cook stage builds only the dependencies from that recipe. The cook layer is reused until the dependencies change. Its documentation warns that every stage must use the same Rust version, or the cache will not behave as expected, which is one more reason the toolchain pin from Chapter Seven matters.

```
# Excerpt; full template in Appendix A. Replace each <digest> with a pinned sha256.
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"]
```

Minimal bases. A Rust binary linked against glibc runs on a distroless base that carries the C runtime and little else, such as the distroless `cc` image. A binary built for a musl target, such as `x86_64-unknown-linux-musl`, is statically linked by default and can run on `scratch`, which is empty, or on the distroless `static` image. Remember what empty means: `scratch` has no CA certificates, no time zone data and no shell. A service that calls TLS endpoints must bring its certificates, and nobody will `exec` into the container to debug it. Chapter Eleven's telemetry has to cover that gap.

> **musl is a trade-off, not a free win.** A static musl build can enable a minimal runtime image, such as `scratch` or distroless `static`; measure the size. In exchange, musl's default memory allocator can perform worse than glibc's under allocation-heavy, multithreaded load, which is why many teams set a different global allocator for musl builds. Name resolution and some C dependencies also behave differently. Decide by measuring your own service on both, with the footprint method in Appendix A and the latency method in Chapter Eleven, and record the result in the decision record.

Pinned base image digests. A tag such as `cc-debian12` moves. A digest does not. Pin every `FROM` line by digest, and let a bot open a pull request when the upstream image changes, so updates arrive through review and the admission gate instead of silently through the next build.

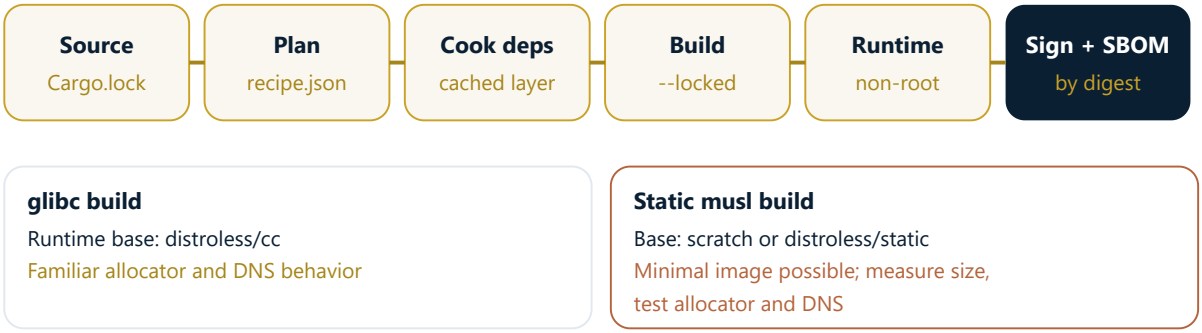
Non-root user. Run as a numeric, unprivileged user. The number matters: Kubernetes can verify `runAsNonRoot` only when the image declares a numeric user or the Pod sets `runAsUser`, which is why the excerpt uses `65532`, the distroless nonroot user, rather than the name.

A portable CPU target. `-C target-cpu=native` tells the compiler to use every instruction the build machine has. That is fine on a workstation. In a pipeline it ties the artifact to whatever CPU the CI runner had, and the binary can fail with an illegal instruction on an older node. Choose a named baseline, such as an x86-64 microarchitecture level, and make it part of the contract with the node pools.

```
# .cargo/config.toml (excerpt): an agreed baseline, not the build machine
[target.x86_64-unknown-linux-gnu]
rustflags = ["-C", "target-cpu=x86-64-v2"]
```

FIGURE 10.2

From pinned source to an image referenced by digest



The dependency layer rebuilds only when the recipe changes. Every base is pinned by digest.

Six steps, one output. The pipeline ends in an image reference with a digest, a signature and an SBOM. The runtime base is a choice between two paths, made by measurement and recorded.

The reference service, and why there are no numbers here

Readers ask how small and how fast. I will not answer with figures in a chapter, because a footprint number without its method is marketing, and a number from one estate does not transfer to another. Appendix A contains a small reference service written for this book, the full Dockerfile template (reviewed, not built for this edition) and the build settings above, and a footprint smoke measurement (binary size and memory) with its method: the hardware, the toolchain version and each step. It reports no latency; Chapter Eleven gives the method for that. Rerun both on your own hardware. The number that should drive your decision is the one you measure.

EXAMPLE: ONE IMAGE, TWO ENVIRONMENTS

Consider a team that builds an image in CI, tests it in staging, and then rebuilds it with a release tag for production. The rebuild picks up a new base image published overnight. Production now runs bytes nobody tested. Under the contract, the pipeline builds once, records the digest, and promotes that digest; the production manifest cannot name a tag, so the rebuild is not possible by accident.

> *The developer proposes. The compiler decides what it can. The pipeline decides which bytes leave the building, and it decides once.*

Contract card

Field	Shipping the artifact
Contract	Every artifact is built once from a pinned toolchain, a committed lockfile and digest-pinned bases; it is signed, carries an SBOM, and is promoted by digest. Self-updating binaries can always return to the previous version.
Threat addressed	Untested bytes in production, drifting base images, tampered or unsigned releases, root containers, binaries that crash on older CPUs, updates that cannot be undone.
Owner	Platform team owns the pipeline templates, signing service and base image list; each product team owns its targets, channels and rollback test.
Enforcement point	CI release template; registry and cluster admission that accept only signed images referenced by digest and running as non-root; client update logic that verifies signatures.
Failure behavior	Release blocked if any input is unpinned, unsigned or built with `target-cpu=native`; a failed post-update health check reverts automatically.
Waiver authority and expiry	Platform security lead; a waiver names the artifact and the missing control and expires within one release cycle.
Evidence the control ran	Signature and attestation records per digest, the pipeline log showing `--locked`, admission logs, a rollback drill result per channel.
Evidence the outcome improved	Share of production workloads running a digest that passed staging; rollbacks completed without a rebuild; no illegal-instruction crashes after node changes.
Review cadence	Quarterly, and whenever a base image family, signing key or supported target changes.

CONTRACT CHECKLIST

- Production manifests and release channels reference artifacts by digest, and promotion never rebuilds.
- Every base image is pinned by digest and updated only through a reviewed pull request.
- Images run as a numeric non-root user on a minimal base, and the glibc or musl choice is recorded with its measurement.
- Release builds use an agreed CPU baseline, never `target-cpu=native`.
- Every released binary is signed by the pipeline, and self-update verifies, swaps atomically and can revert.
- The supported target list is written down with each target's tier.

CHAPTER ELEVEN

Running Under Contract



A small, signed image is a promise about what was shipped. It says nothing yet about how the service behaves at three in the morning, under load, during a node drain or halfway through a bad release. That behavior is owned by operations, and it is where the earlier tiers either pay off or quietly stop mattering. The contract at this door is simple to say: the cluster knows what the service needs, the service tells the truth about its health, it stops cleanly when asked, it can be seen from outside, and it can go back.

For a CTO: Rust can make a service lean and its latency steady, but no language makes that visible. If the platform cannot see it, the estate cannot rely on it.

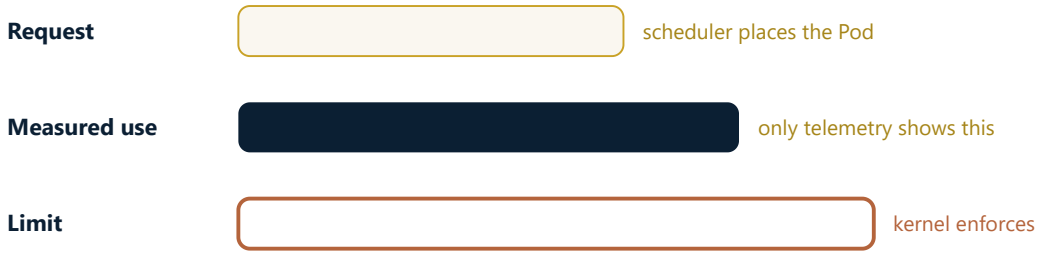
Requests are inputs, limits are enforcement, neither is measurement

The most common misreading in Kubernetes is to treat a Pod's resource numbers as a description of the service. They are not. The Kubernetes documentation is precise: the scheduler uses the **request** to decide which node a Pod is placed on. A container may use more than its request when the node has room, and usage above the request is what the kubelet looks at first when a node runs short of memory and it must evict. **Limits** are enforced: CPU limits by throttling, memory limits by the kernel with out-of-memory kills, applied reactively when the kernel detects memory pressure (Kubernetes documentation, Resource Management for Pods and Containers, 2026).

Neither number is measured use. The scheduler places the Pod by its requests. At runtime the CPU request becomes a weighting, so under contention the kubelet gives the container at least its CPU request; a memory request is mainly a scheduling input, not a hard reservation. A limit says where the kernel steps in. Only telemetry says what the service actually does. A small request written because "Rust is small" is a claim, and the scheduler will believe it.

FIGURE 11.1

Three different numbers, three different owners



Use can sit above the request. Memory over the limit risks an OOM kill; CPU is throttled.

Set requests from measured use, not from belief. The scheduler trusts the request. The kernel enforces the limit. The service's real behavior lives in the middle row, and only the telemetry from later in this chapter can fill it in.

Request and limit choices also set a Pod's quality of service (QoS) class. Under memory pressure the kubelet does not rank Pods by QoS class. It ranks them by whether usage exceeds requests, then by Pod priority, then by usage relative to requests (Kubernetes documentation, Node-pressure Eviction, 2026). QoS is a good estimate of that order for memory, not the rule itself. The table is about memory pressure: disk pressure ranks Pods by disk use, and inode or PID pressure by priority alone.

QoS class	When a Pod gets it	What it means under memory pressure
Guaranteed	Every container has CPU and memory requests equal to its limits.	Evicted last, with Burstable Pods under their requests, by priority. Another Pod's use alone does not evict it, but it is still evictable, lowest priority first, when system daemons overrun their reservation. The right default for latency-sensitive services.
Burstable	At least one container has a request or limit, but the Pod is not Guaranteed.	Ranked early when using more than its request; below its request, it ranks with Guaranteed Pods by priority.
BestEffort	No requests or limits at all.	Any use exceeds a zero request, so it ranks among the first. Not acceptable for anything with a contract.

CPU limits deserve a judgment of their own. Throttling does not crash a service; it slows it, and the slowdown appears in the tail. Some teams leave CPU unlimited for latency-sensitive services; others need hard CPU limits for fairness. Either is defensible if the choice follows the throttling metrics.

Three probes, three different questions

Probe	Question it answers	What Kubernetes does on failure	Rule
Startup	Has the service finished starting?	Holds off liveness and readiness until it succeeds; restarts the container if it never does.	Use it instead of long liveness delays.
Readiness	Should this Pod receive traffic now?	Removes the Pod from Service endpoints; does not restart it.	May reflect critical dependencies and draining.
Liveness	Is the process stuck beyond recovery?	Restarts the container.	Never checks a shared dependency such as a database.

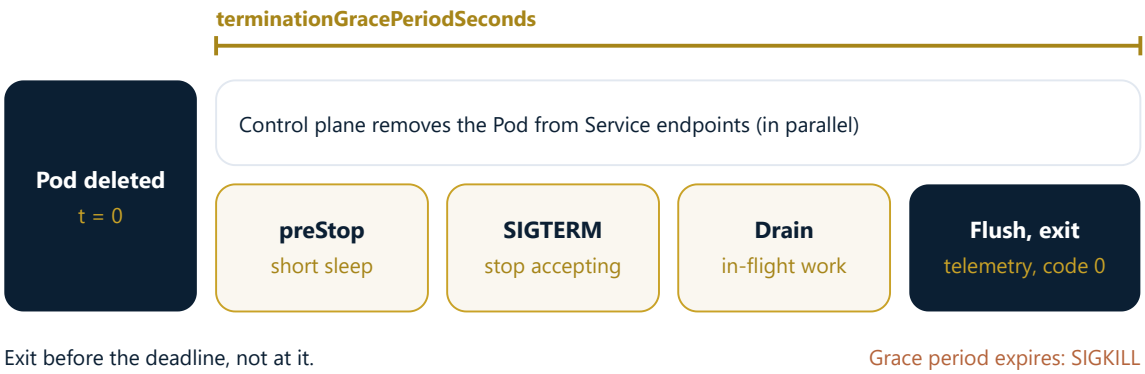
The liveness rule is the one teams break, as the example at the end of this chapter shows. Liveness should ask only whether this process can still make progress. Rust does not prevent deadlocks, so a liveness check that exercises the same executor as real requests is worth more than one that returns a constant. A simple form: the handler spawns a trivial task on the async runtime and awaits it, so a runtime that no longer schedules work fails the probe.

Stopping cleanly on SIGTERM

When a Pod is deleted, the kubelet runs any `preStop` hook and sends SIGTERM to the main process. When the grace period ends, remaining processes get SIGKILL. The grace period defaults to 30 seconds and is set by `terminationGracePeriodSeconds` (Kubernetes documentation, Pod Lifecycle, 2026). At the same time, and not in a guaranteed order, the control plane removes the Pod from Service endpoints, so for a short window new requests can still arrive after the signal.

FIGURE 11.2

Graceful shutdown inside the grace period



Two things happen at once. The endpoints update while the container shuts down. The short preStop pause covers the window in which traffic still arrives; the service then drains, flushes its telemetry and exits well inside the grace period.

In a Rust service on Tokio, the pattern is a shutdown future that resolves on SIGTERM, handed to the server so it stops accepting connections and finishes the ones it has. Tokio's signal support needs the `signal` feature. The excerpt below uses axum; other servers have the same 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");
}

// Stop accepting, finish in-flight requests, then return.
axum::serve(listener, app)
    .with_graceful_shutdown(shutdown_signal())
    .await?;
// Flush telemetry exporters here, before main returns.
```

Kubernetes does not change readiness when SIGTERM arrives. If the readiness probe should go false while the service drains, the application must flip it in its shutdown path, as the reference service in Appendix A does.

Two details catch teams. First, a distroless image has no shell, so a `preStop` hook cannot run `sleep` as a command. Use the built-in `sleep` action for lifecycle hooks, which is beta and enabled by default since Kubernetes 1.30 and stable since 1.34 (Kubernetes documentation,

feature gates, 2026), or wait briefly in process after SIGTERM before closing the listener. Second, background tasks spawned with `tokio::spawn` do not stop because the server stopped. Give them a cancellation signal, and bound the whole drain with a deadline shorter than the grace period minus the preStop pause.

```
# Deployment excerpt; full template in Appendix A. Resource values come from measurement.
# Burstable on purpose: no CPU limit (see throttling above). For Guaranteed, set
# cpu and memory limits equal to their 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 } }   # built-in action; no shell needed
    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 }
```

Being seen: logs, traces and metrics

A minimal image with no shell is only operable if it explains itself. Three signals carry that.

- **Structured logs.** One JSON event per line, with fields rather than sentences: request ID, trace ID, route, outcome. The `tracing` crate is the common instrumentation layer in Rust, and `tracing-subscriber` can emit JSON.
- **Traces.** `tracing` spans map naturally onto distributed traces, and the OpenTelemetry Rust crates export them over OTLP to whatever backend the estate runs. As of November 2026, the OpenTelemetry Rust page (https://opentelemetry.io/docs/languages/rust/) lists metrics and logs as stable and traces as beta. Check it before you standardize, and pin the OpenTelemetry crates together; they move as a set.
- **Metrics.** For every endpoint: rate, errors and duration, with duration recorded as a histogram. Add the runtime signals that explain the tail: CPU throttling from the cluster, memory against the limit, and queue depth or task counts from the async runtime.

```
// main.rs excerpt: JSON logs with span context
tracing_subscriber::fmt()
    .json()
    .with_current_span(true)
    .init();
```

p99, measured honestly

Predictable tail latency is one of the main reasons teams pick Rust over a garbage-collected runtime, and it is also the claim most often made without evidence. Modern collectors, such as ZGC on the JVM, keep pauses short, and a Rust service can still have a bad tail from lock contention, a blocking call on an async worker thread, allocator behavior or CPU throttling. A fair measurement has rules.

RULES FOR A LATENCY CLAIM

- Same workload, same hardware, same cluster settings for every candidate being compared.
- Warm up first, and discard the warm-up period from the results.
- Report percentiles (p50, p99, p99.9) from a histogram, never an average, and never average percentiles across instances; merge the histograms instead.
- Drive load at a fixed rate. A closed-loop tool that waits for each reply before sending the next request hides stalls; this is the coordinated omission problem, and it makes tails look better than users experience.
- Publish the method with the number: tool, rate, duration, hardware, toolchain version.

Appendix A reports only footprint for the reference service, not latency, because its simple sequential client is the closed-loop pattern these rules exclude. Measure latency for your own service with these rules before a number appears in a decision record.

Going back

`kubectl rollout undo` returns a Deployment to its previous revision, meaning the previous Pod spec. If that spec named a tag, the bytes behind it may have changed. With digests, the previous revision is exactly the previous artifact. Rollback is also real only when database changes are backward compatible, so the old version still runs against the new schema, and when it is rehearsed.

EXAMPLE: THE RESTART STORM

Consider a team whose liveness probe queries the shared database. A slow index rebuild pushes query time past the probe timeout. Every replica fails liveness and restarts at once, the restarts add connection load, and the service goes dark. Under the contract, liveness checks only the process, readiness takes the slow replicas out of rotation, and the service degrades instead of failing.

> *The compiler proved the service is memory-safe. Only the cluster's telemetry can prove it is healthy. Every line holds only as far as someone can see it holding.*

Contract card

Field	Running under contract
Contract	Every service declares requests from measured use, runs as Guaranteed or justified Burstable, exposes startup, readiness and liveness probes with distinct meanings, drains on SIGTERM within the grace period, emits structured logs, traces and histogram metrics, and deploys by digest with a rehearsed rollback.
Threat addressed	Overcommitted nodes, surprise evictions, restart storms, dropped requests during rollouts, invisible tail latency, unsupported performance claims, rollbacks to unknown bytes.
Owner	Platform team owns the Deployment template, admission policy and telemetry pipeline; each product team owns its probes, shutdown code, resource values and latency budget.
Enforcement point	Cluster admission policy (digests, non-root, requests present, probes present); CI check on manifests; a shutdown test in the pipeline that sends SIGTERM under load.
Failure behavior	Manifests missing requests, probes or digests are rejected at admission; a release whose p99 exceeds its budget in staging does not promote.
Waiver authority and expiry	Platform lead for the service's tier; a waiver names the missing control and the compensating one and expires within one quarter (example estate policy).
Evidence the control ran	Admission logs, shutdown test results, the published load-test method and histogram per release, rollback drill records.
Evidence the outcome improved	Errors during rollouts and node drains trend down; p99 stays within budget across releases; requested resources track measured use; rollback time is known.
Review cadence	Quarterly, after every incident involving restarts, evictions or rollbacks, and when the cluster version changes.

CONTRACT CHECKLIST

- Resource requests are set from measured use and reviewed against telemetry, not written from expectation.
- Liveness never checks a shared dependency; startup and readiness probes exist and mean what they say.
- The service drains on SIGTERM, stops background tasks and exits inside the grace period, and a test proves it.
- Logs are structured, traces export over OpenTelemetry, and duration is recorded as a histogram.
- Every latency claim carries its method: fixed-rate load, warm-up, percentiles, hardware and toolchain.
- Deployments reference digests, and rollback is rehearsed each quarter.

CHAPTER TWELVE

The Operating Model



One team writing Rust is a technical choice. Twenty teams writing Rust is an operating model, whether anyone designed it or not. The contracts in this book only hold if someone owns each one, someone can grant an exception and take it back, and someone checks, on a date, that the controls still run and the outcomes still justify the choice.

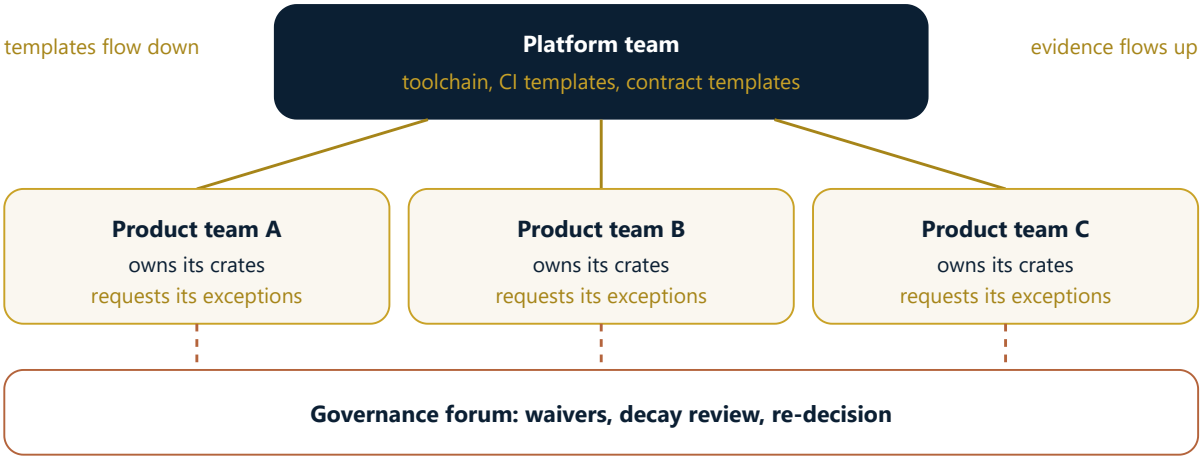
The short version for a CTO: a platform team owns the shared machinery, product teams own their code and their exceptions, every contract is a card with an owner and evidence, a small scorecard shows outcomes, and once a year the decision is made again on purpose.

Who owns what

The third tier, operations and people, is the whole of this door. The compiler cannot staff a review board, so the first design question is ownership. The split that works in most estates is one platform team and many product teams.

FIGURE 12.1

Templates flow down, exceptions and evidence flow up



Ownership is the contract under every other contract. The platform team makes the right path the easy path. Product teams own what they build and ask for what they need. The forum, which can be a standing agenda item rather than a new committee, decides exceptions and reviews decay.

- **The platform team owns** the pinned toolchain and its upgrade cadence, the CI templates that run the gates from Parts II and III, the admission policy, the image and Deployment templates from Part IV, the estate-standard crates (including the async runtime) and the contract templates.
- **Product teams own** their crates, their public interfaces, their unsafe register entries, their probes and latency budgets, and their exceptions.
- **The governance forum** grants waivers above a team's authority, runs the decay review, and prepares the annual re-decision. Keep it small: the platform lead, a security lead, and rotating product team leads.

Two tiers of governance

A governance model sized for fifty teams will crush three. Chapter Three introduced a minimum viable tier; here it is beside the full tier.

Area	Minimum viable tier	Full tier
Toolchain	rust-toolchain.toml pinned; Cargo.lock committed in every repository	Estate-wide toolchain cadence owned by the platform team, with an emergency bump path
Code gate	fmt and Clippy with warnings denied in CI	Shared lint profile, tested gates, Miri and fuzzing where unsafe or parsing code exists
Dependencies	Advisory scan on every build; license allowlist	Full admission gate: bans, sources, vetting, private mirror
Exceptions	One register file for unsafe, FFI and waivers, each with owner and expiry	Exception lifecycle with waiver authority by risk level and decay review
Artifacts	Build once, deploy by digest	Signing, SBOM, provenance, admission that rejects anything else
Runtime	Probes, SIGTERM drain, structured logs	Deployment template, OpenTelemetry pipeline, latency budgets with published method
Decision	ADR with a review date	ADR with renewal contract, scorecard and annual re-decision

Move an area from the left column to the right when the scorecard says the risk justifies it, not because the right column exists.

The Contract card, in full

Chapters Four to Eleven each ended with a card, the unit of governance in this book. Each field has a job.

Field	What it must say
Contract	One or two sentences, testable. "Every release is signed" is a contract. "We take security seriously" is not.
Threat addressed	The specific failure the contract prevents or detects. If nobody can name one, retire the contract.
Owner	A role, not a person, with a named holder recorded elsewhere.
Enforcement point	Where a machine checks it: compiler, CI job, admission policy, runtime. "Code review" alone is a tier-three control and should say so.
Failure behavior	What happens when the check fails: block, warn, revert. A gate that only warns is a report.
Waiver authority and expiry	Who can say "yes, for now", and the longest "for now" allowed.
Evidence the control ran	The record that proves the check executed, readable without asking anyone.
Evidence the outcome improved	The measure that shows the threat is actually smaller. This is the field most cards leave weak.
Review cadence	A date, plus the events that force an early review.

Keep cards as files beside the control, so both change in the same pull request. Review works in three steps. A new card is proposed by its owner and reviewed by the platform team for the enforcement point and by security for the threat. It enters as a draft and becomes active only when both evidence fields point at something that exists. At each review date, the forum reads the evidence, not the prose, and decides to keep, tighten, loosen or retire.

```
# contracts/admission-gate.card.yaml (excerpt; template in Appendix A)
contract: "No dependency with an open, unwaived advisory enters a release build."
owner: platform-security-lead
enforcement_point: ci/release-gate
failure_behavior: block
waiver: { authority: platform-security-lead, max_days: 30 }   # example estate policy
evidence_ran: "gate job log per release"
evidence_outcome: "days from advisory published to fixed in production"
review: { every: quarter, also_on: [gate-bypass, incident] }
status: active
```

A scorecard that shows outcomes, not ritual

Ritual metrics count activity, and they always look good. Outcome metrics show whether the estate is safer or cheaper to change, and they are allowed to look bad.

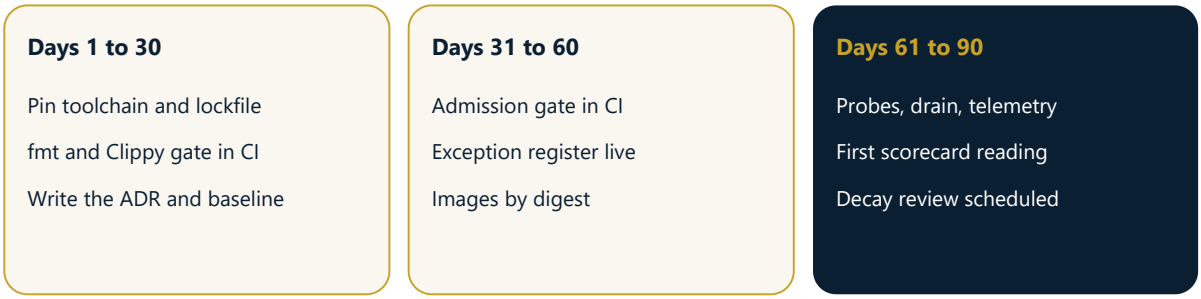
Measure	What it shows	The ritual it replaces
Time from advisory published to fixed in production	Supply-chain response, end to end	Number of scans run
Waivers past expiry, and waivers renewed more than once	Whether exceptions stay temporary	Number of waivers approved
Unsafe blocks with an owner, a stated invariant and tests under Miri where Miri supports the code, otherwise fuzzing, sanitizers or model tests	How much of the trusted boundary is actually held	Lines of unsafe code
Production incidents by tier: compiler, tooling, operations	Where failures really come from	Total incident count
Time to roll the estate to a new toolchain	Whether the pin is a control or a trap	Toolchain version reported
Releases within latency budget, by published method	Whether the runtime claim holds	A single benchmark slide
Weeks for a new engineer to merge a reviewed change	Whether the learning curve is managed	Training hours logged

The incidents-by-tier row matters most. If most incidents come from logic and operations, that is the expected shape once the compiler has removed one class of bug, and it shows where to invest next.

A 30-60-90 rollout

FIGURE 12.2

Minimum viable first, then evidence, then the full tier where it pays



Minimum viable tier first; the full tier only where the scorecard says it pays.

Ninety days to a governed baseline, as a planning assumption. Size the phases to your estate. Each phase adds controls that produce evidence the next phase uses. By day ninety the estate has its first honest scorecard and a date for its first decay review.

Start with one service that has a real owner and modest risk, and make its gates pass before they become mandatory for anyone else. In the second month, turn the dependency and exception contracts on in warning mode for a fixed period, then blocking. In the third month, add the runtime contract and take the first scorecard reading. It will be incomplete. Publish it anyway, with the gaps named.

Skills and hiring

I will be plain about this. In my experience the pool of experienced Rust engineers is smaller than the pool for Java, .NET, Go or TypeScript; check it against your own hiring market, as Chapter Two asked you to price it into the decision. The answer that works is mostly internal. Train engineers you already trust, because domain knowledge and judgment are harder to hire than syntax. Pair newcomers with experienced Rust reviewers for their first changes. Keep a short, opinionated estate guide that names the standard crates, error pattern and async runtime. Expect the learning curve to show in the scorecard's onboarding row, and manage it as a cost.

Hire for systems thinking. In my experience, an engineer who reasons well about ownership in C++ or concurrency in Go learns Rust's rules faster than a job description suggests. Do not let Rust become a status marker inside the organization. A team that keeps a service in Go or .NET because it fits better is following the decision record, not resisting it.

Decay review and the re-decision

Every control decays. Waivers are renewed by habit, a bypass for one urgent release stays, and cards stay "active" after their evidence links break. The **decay review**, once a quarter, exists to catch this. For each card it asks four questions: did the control run, is the evidence readable, did the outcome move, and are there exceptions that have outlived their reason? Cards that fail go back to draft, with an owner and a date.

Once a year, the forum prepares the **re-decision**. Chapter Two ended with a renewal contract: expected outcomes, a baseline, a review date, reversal triggers and an exit path. The re-decision reads the scorecard against that contract and answers one question honestly: does the evidence still support Rust for the scope it was chosen for? The answer can be "continue", "widen", "narrow" or "exit". Each is a legitimate outcome. A decision that cannot come out differently is not a decision; it is a habit.

> **EXAMPLE: A NARROWING THAT WAS A SUCCESS**
>
> Consider an estate that chose Rust for its data plane and later allowed it for internal tools. At the re-decision, the data plane is within its latency budget, while the tools show slow onboarding and no outcome gain. The forum narrows the scope: the tools move to the estate's main managed language at their next rewrite, and the ADR records why. That is the contract working.

Closing the book

This book began with a wish to own the bits. The three tiers are how that wish becomes something an estate can rely on. The compiler proves what it can prove, inside the trusted boundary, every time. Tooling enforces what the compiler cannot see, while someone tests that the gates still block. Operations and people own the rest.

So the line I started from needs its full form. The developer proposes. The compiler decides what it can. People decide the rest, with contracts that have owners, expiries and evidence, and a date on which they will decide again.

> *The developer proposes, the compiler decides what it can, and people decide the rest. Trust, earned at every layer.*

CONTRACT CHECKLIST

- Every contract has a card in the repository that holds its control, with a role as owner and both evidence fields filled.
- The platform team owns the toolchain, CI templates and contract templates; product teams own their crates and their exceptions.
- The estate has chosen its governance tier per area, and moves an area to the full tier only on scorecard evidence.
- The scorecard reports outcomes, including incidents by tier, and is published even when incomplete.
- A quarterly decay review and an annual re-decision against Chapter Two's renewal contract are on the calendar with named owners.

APPENDIX A

Templates and the Reference Service

Chapters 2 to 12 show short excerpts. The full files live here, ready to copy, each with the toolchain version it was checked against; the one deliberate exception, the release job of the CI template, is marked as a skeleton. Templates age: when you bump the pin, rerun each one and update its verification line, the same way the chapters ask you to renew a decision.

The second half of the appendix describes the reference service: a small HTTP service written for this book, built with these templates, and measured with a method you can rerun. It is the only source of footprint numbers in the book. Its numbers describe that service on that hardware, not Rust in general, and not your workload.

The pin: rust-toolchain.toml

Placed at the repository root, this file makes `rustup` select the same compiler on every laptop and every CI runner. Bump it on the estate cadence, never silently in a feature branch.

```
# Verified against Rust 1.100.0 stable, November 2026
[toolchain]
channel = "1.100.0"
components = ["rustfmt", "clippy"]
profile = "minimal"
# Add release targets the estate supports, for example:
# targets = ["x86_64-unknown-linux-musl", "aarch64-unknown-linux-gnu"]
```

Package manifest and release profile

The manifest states the edition and the minimum supported Rust version. The release profile trades build time for a smaller, faster binary. `panic = "abort"` ends the process on a panic instead of unwinding, so the orchestrator restarts it; choose it deliberately. `overflow-checks` is stated, not left to the default, as the Chapter Four checklist asks: this profile chooses a panic over a silent wrap. A team that keeps the release default writes `overflow-checks = false` and uses checked arithmetic for money and quotas.

The reference service promises 1.85, the same MSRV as Chapter Seven's workspace example. It is the floor its dependencies set (the locked versions declare a rust-version of 1.85 at most) and the first release with the 2024 edition. The promise is proved by `cargo +1.85.0 check`

`--locked --all-targets`, which passes; the `msrv` job in the CI template runs the same check.

```
# Verified against Rust 1.100.0 stable, November 2026
[package]
name = "refservice"
version = "0.1.0"
edition = "2024"          # stable since Rust 1.85
rust-version = "1.85"     # the MSRV this crate promises
publish = false
license = "MIT OR Apache-2.0"

[profile.release]
lto = true                # whole-program optimization; slower builds
codegen-units = 1         # better optimization; less parallel compilation
strip = true              # strip symbols from the shipped binary
panic = "abort"           # no unwinding; the process ends on panic
overflow-checks = true    # stated, not defaulted: panic instead of wrapping
```

The [lints] tables

Lints in the manifest travel with the code, so the policy is the same in the editor and in CI. In a workspace, put them under `[workspace.lints]` and add `[lints] workspace = true` to each member. There are two tables, because a service binary and a library have different panic rules (Chapter Four). Both set `unsafe_code` to `deny`, so a crate that needs unsafe can allow it in one named module; a crate that never needs unsafe tightens it to `forbid` (Chapter Five). The reference service does that.

```
# Verified against Rust 1.100.0 stable, November 2026
# Service binary
[lints.rust]
unsafe_code = "deny"               # "forbid" if the crate never needs unsafe
unsafe_op_in_unsafe_fn = "deny"    # for crates on the unsafe register
missing_debug_implementations = "warn"

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

```
# Verified against Rust 1.100.0 stable, November 2026
# Library: no panic reachable from input
[lints.rust]
unsafe_code = "deny"                # "forbid" if the crate never needs 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"
```

The admission gate: deny.toml

cargo-deny reads this file. Every entry in `ignore` needs an exception record with an expiry. The license list and the `openssl` ban are examples; your legal and platform teams own the real ones. This is the complete file the reference service uses; Chapter Eight shows an excerpt of it with a sample waiver. Duplicate versions are denied, as in Chapter Eight: a duplicate the team cannot remove yet gets a `skip` entry with a reason, so the exception is visible and has an exit. The file was checked with cargo-deny 0.20.2, which rejects unknown keys and values when it loads the file. Vulnerability advisories always fail, so there is no key for them (0.20.2 reports the old `vulnerability` key as removed). `unmaintained` takes `all`, `workspace`, `transitive` or `none`. An advisory ignore entry takes an `id` and a `reason` (a yanked version can instead be named by crate spec) and has no expiry field, so the expiry lives in the exception record. A skip names a crate with a version requirement after a colon, as in `syn:<3`. This is the documented form, not a typo: in a cargo-deny crate spec, `:` introduces a version requirement and `@` an exact version (`syn@2.0.0` means `syn:=2.0.0`), and 0.20.2 accepts both (cargo-deny docs). The `targets` list holds every target the CI build matrix ships, so platform-specific dependencies are judged too. The registry entry is the crates.io index URL that Cargo writes into `Cargo.lock`.

```
# deny.toml (complete)
# cargo-deny configuration for refservice.
# Verified with cargo-deny 0.20.2 (cargo deny check passes), Rust 1.100.0, November 2026.

[graph]
# Every target the CI template ships (Appendix A build matrix).
targets = [
    "x86_64-unknown-linux-gnu",
    "aarch64-unknown-linux-gnu",
    "x86_64-pc-windows-msvc",
    "aarch64-apple-darwin",
```

```
]
all-features = false

[advisories]
# Vulnerability advisories always fail; also fail on unmaintained crates and yanked versions.
unmaintained = "all"
yanked = "deny"
# Each ignore entry needs an exception record (Appendix A) and an expiry.
ignore = []

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

[bans]
multiple-versions = "deny"
wildcards = "deny"
# One TLS stack for the estate (example ban; the service has no TLS dependency).
deny = [{ crate = "openssl", use-instead = "rustls" }]
# Each skip needs a reason and an exit plan; remove it when upstream converges.
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 = []
```

The CI gate

GitHub Actions syntax; the same jobs translate to any CI system. Every Cargo step that resolves dependencies uses `--locked`, so CI fails rather than silently changing `Cargo.lock`. cargo-deny is installed at the exact version `deny.toml` was checked with, not taken from a floating action. Since rustup 1.28, `rustup toolchain install` with no arguments installs the toolchain that `rust-toolchain.toml` names; this was checked with rustup 1.29.3, without changing the default toolchain. The `msrv` job proves the `rust-version` promise on that exact compiler. The semver job applies to libraries only and compares with the last release tag, not with `main` (Chapter Nine), on pull requests and on the release tag itself. It calls the cargo-semver-checks CLI directly and picks the baseline per crate: the nearest earlier tag that matches `orders-client-v*`, excluding prerelease tags such as `orders-client-v2.0.0-rc.1` (git describe docs). With no earlier release tag, the job says

so and skips; when there is a baseline, a failed check fails the job. The release job runs only for a version tag and only after the gate, MSRV, semver and build jobs pass. It is a skeleton: fill in your own publish, sign and attest steps. Whatever you add, keep the rule from Chapter Ten: promote the artifact the build job produced, verified by its digest, and never rebuild to promote. Pin each action by full commit SHA in production; the tags here keep the template readable.

```
# Verified against Rust 1.100.0 stable, November 2026. Pin each action by full commit SHA.
name: ci
on:
  pull_request:
  push:
    branches: [main]
    tags: ["v*", "*-v*"]  # a release is a version tag; libraries tag per crate, e.g. orders-
client-v1.4.2
permissions:
  contents: read

jobs:
  gate:
    runs-on: ubuntu-24.04
    steps:
      - uses: actions/checkout@v7
      - name: Install the toolchain rust-toolchain.toml names (rustup 1.28 or later)
        run: rustup toolchain install
      - run: cargo fmt --all --check
      - run: cargo clippy --workspace --all-targets --locked -- -D warnings
      - run: cargo test --workspace --locked
      - name: Install the cargo-deny version deny.toml was checked with
        run: cargo install cargo-deny --locked --version 0.20.2
      - name: Admission gate (targets and features come from [graph] in deny.toml)
        run: cargo deny --locked check

  msrv:     # proves rust-version in Cargo.toml on that exact compiler
    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:   # libraries only: compares the public API with the last release tag
    runs-on: ubuntu-24.04
    steps:
      - uses: actions/checkout@v7
        with: { fetch-depth: 0 }       # history and tags, for the baseline
      - run: rustup toolchain install
      - run: cargo install cargo-semver-checks --locked   # pin --version to the one you checked
      - name: Compare with this crate's last stable release tag
        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:  # SKELETON: fill in your publish, sign and attest steps. Version tags only, after
every check
    if: startsWith(github.ref, 'refs/tags/v')
    needs: [gate, msrv, semver, build]   # a service with no public API drops semver
    runs-on: ubuntu-24.04
    steps:
      - run: echo "skeleton: add publish, sign and attest steps (Chapters Eight and Ten)"
```

The Dockerfile

Multi-stage, with cargo-chef so the dependency layer is cached until `Cargo.toml` or `Cargo.lock` change. The final stage is a distroless image with the C runtime, no shell and no package manager, running as the numeric non-root user 65532, so Kubernetes can check `runAsNonRoot` (Chapter Ten). Replace each tag with a digest before production. The base image tags and the cargo-chef version were confirmed to exist. This file is a reviewed template: the image build has not yet been run for this edition, so it is not verified and its image size is not reported. The `<digest>` placeholders do not build as written; replace each with the digest you pin.

```
# syntax=docker/dockerfile:1
# Template, reviewed November 2026; image build pending
# Replace each <digest> with the digest you pin for that tag.
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
# Dependencies only: cached until Cargo.toml or Cargo.lock change.
RUN cargo chef cook --release --locked --recipe-path recipe.json
COPY . .
RUN cargo build --release --locked --bin refservice

# glibc build, so the runtime is distroless cc. For a static musl build
# (--target x86_64-unknown-linux-musl), use gcr.io/distroless/static-debian12:nonroot
# and copy from 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"]
```

The Kubernetes Deployment

Requests are scheduling inputs, not measured use; set them from your own measurement and revisit them. The memory limit is the contract with the node. Readiness goes false on SIGTERM only because the service flips it; the reference service does so when its shutdown signal arrives. The liveness route spawns and awaits a trivial task, so a runtime that no longer schedules work fails it. The grace period must be longer than the preStop pause plus the drain. The reference service bounds its drain at 20 seconds, inside the 30-second grace period minus the 5-second pause, and then exits even if requests are still in flight. The `preStop` hook uses the built-in `sleep` action, because the distroless image has no shell (stable since Kubernetes 1.34). This Deployment is Burstable on purpose: there is no CPU limit (Chapter Eleven on throttling), and the memory limit sits above the request. For the Guaranteed class, set CPU and memory limits equal to their requests.

```
# Template, reviewed November 2026 (Kubernetes 1.34 or later for the sleep action)
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 }   # replace with measured values
            limits:   { memory: 64Mi }               # Burstable on purpose; see above
          lifecycle:
            preStop:
              sleep: { seconds: 5 }   # endpoints update while traffic still arrives
          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"] }
```

The decision record (ADR)

One record per language or platform decision. The renewal section is what keeps it a contract rather than a memo (Chapter 2).

```
# Template, reviewed November 2026
# ADR-NNN: <decision title>
Status:      proposed | accepted | renewed | adjusted | reversed
Date:        YYYY-MM-DD
Owner:       <role, not a person's name>
Approvers:   <architecture board or equivalent>

## Context
```

```
The business outcome, the workload, and the constraint that forces a choice.

## Criteria (weights agreed before scoring)
| Criterion              | Weight | Rust | Alternative A | Alternative B |
| Runtime footprint      |   <w>  |      |               |               |
| Team skills and hiring |   <w>  |      |               |               |
| Ecosystem fit          |   <w>  |      |               |               |
| Build and CI cost      |   <w>  |      |               |               |

## Alternatives considered
One fair paragraph each: where it wins, where it loses for this workload.

## Decision
What was chosen, for which scope, and what is explicitly out of scope.

## Consequences
Costs accepted (compile time, training, hiring) and contracts created.

## Renewal contract
Expected outcomes:   <measurable>
Baseline:            <value and how it was measured>
Review date:         YYYY-MM-DD
Reversal triggers:   <conditions that reopen this decision>
Exit path:           <how the estate would move off, and at what cost>
```

An unsafe register entry

One entry per `unsafe` site the estate owns. The invariant is the promise the compiler cannot check; the evidence is how the team checks it instead.

```
# Template, reviewed November 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            # a // SAFETY: comment states the invariant
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"
```

An exception record

Every exception follows one lifecycle: request, owner, waiver authority, evidence, expiry, decay review (Chapter 5). The same record covers unsafe code, FFI, build scripts, lint waivers, ignored advisories and emergency toolchain changes.

```
# Template, reviewed November 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"     # waiver authority; never the requester
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"                # no expiry, no approval
on_expiry = "gate fails again; renew with fresh evidence or remove"
```

The reference service

What it is. A small HTTP service written for this book: `refservice`, about 130 lines of Rust on the 2024 edition. It uses axum 0.8 on tokio 1 (multi-threaded runtime), logs JSON lines through `tracing`, and exposes three routes: `GET /healthz` (liveness), `GET /readyz` (readiness, which turns false when shutdown begins) and `GET /hello` (a fixed text body). It shuts down gracefully on Ctrl-C and, on Unix, SIGTERM: readiness drops, in-flight requests drain for up to 20 seconds, the process exits. Traces and metrics are omitted on purpose: the service exists to measure footprint, and a real service adds them (Chapter Eleven). It is built with the service templates above (its own crate tightens `unsafe_code` to `forbid`), and on Rust 1.100.0 it passes `cargo fmt --check`, `cargo clippy --all-targets -- -D warnings` and `cargo deny check` with cargo-deny 0.20.2 and the deny.toml above (advisories, bans, licenses and sources all ok; the only warning is that no dependency uses the allowed ISC license). It does no real work, so its numbers are a floor for a service of this shape, not a forecast for yours.

Rebuild it from this book. Create a folder with the four files below plus `rust-toolchain.toml` and `deny.toml` exactly as shown earlier in this appendix. `Cargo.lock` is generated on the first build; commit it, and build with `--locked` from then on. A fresh lockfile may resolve newer patch versions than the ones listed under the method, so record yours next to your numbers.

```
# Cargo.toml (complete)
[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 (complete)
//! refservice: the reference service for "Governing Rust by Contract".
//! GET /healthz (liveness), GET /readyz (readiness), GET /hello.
//! Graceful shutdown on Ctrl-C and, on Unix, SIGTERM, bounded by a drain deadline.
//! Logs are JSON lines. Traces and metrics are omitted: this example exists to
//! measure footprint, and a real service adds them (Chapter Eleven).

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;

/// Shorter than terminationGracePeriodSeconds (30) minus the preStop pause (5).
const DRAIN_DEADLINE: Duration = Duration::from_secs(20);

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

async fn healthz() -> (StatusCode, &'static str) {
    // Liveness exercises the async runtime: spawn and await a trivial task,
```

```
    // so a runtime that no longer schedules work fails the 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 => {},
    }

    // Stop advertising readiness first, then let in-flight requests drain.
    state.ready.store(false, Ordering::Release);
    info!("shutdown signal received, draining");
}
```

```rust
#[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(());
    });
    // The drain is bounded: once shutdown starts, stop waiting after 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: the request client used in step 4
"""Send N sequential GET /hello requests over one keep-alive connection; check each status."""
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)  # a stalled request fails the
run
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: steps 2 to 5 of the method below
# Footprint smoke measurement for refservice (Appendix A). Windows, PowerShell 5.1 or 7.
# Run from the refservice folder after: cargo +1.100.0 build --release --locked
# A run that fails (port busy, not ready, process died, client error) stops the 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) {
    $clock = [System.Diagnostics.Stopwatch]::StartNew()   # monotonic, total time
    while ($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                      # idle period, measured from readiness
        $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()
    }
}
```

The method.

This is a footprint smoke measurement on Windows, not a production benchmark. It reports binary size and memory only. It reports no latency: a sequential client that waits for each reply is the closed-loop pattern Chapter Eleven warns against, so measure latency for your own service with the Chapter Eleven method.

1. Build: `cargo +1.100.0 build --release --locked` with the release profile above, on Windows 11, target `x86_64-pc-windows-msvc`, dependencies resolved by the committed lockfile to axum 0.8.9, tokio 1.54.1, tracing 0.1.44, tracing-subscriber 0.3.23 (with its `json` feature).
2. Binary size: the file size of `refservice.exe` as reported by the file system.
3. Idle memory: start the binary with `RUST_LOG=warn`, poll `/readyz` until it returns 200 (the script fails if the port is busy, the process exits, or readiness is not reached within a total deadline of 5 seconds, timed with a monotonic stopwatch; a poll already in flight at the deadline can add up to its 1-second request timeout), wait 3 seconds, read the process working set and private bytes from the operating system (PowerShell `Get-Process`: `WorkingSet64`, `PrivateMemorySize64`).
4. Memory after use: a Python client on the same machine sends 10,000 sequential `GET /hello` requests over one keep-alive connection to 127.0.0.1 and checks each status code. Wait 2 seconds, read memory again. The requests exist only to exercise the process; they are not timed.
5. Repeat from a fresh process each time, five runs in all, and report the range. The script always stops the process, stops at the first failed run (a client error or a process that died), and only complete runs are reported. The script above does steps 2 to 5: `powershell -File measure\measure.ps1` from the service folder.

Hardware. A 16-core desktop x86_64 CPU (32 hardware threads) with 64 GB of RAM, Windows 11, measured 15 November 2026. By default, Tokio's multi-threaded runtime starts one worker per unit of parallelism available to the process, as reported by

`std::thread::available_parallelism` (Tokio documentation); here that was 32, and the process had 36 threads in every run. A container with a CPU limit, or an explicit worker count, would have fewer.

Measurement	Result	Notes
Release binary size (Windows x86_64)	1,315,840 bytes (about 1.3 MB)	Single executable with no garbage collector or virtual machine to install; it still links the system C runtime
Working set, idle 3 seconds after readiness	6.0 MB (6,080 to 6,116 KB)	Five runs; private bytes 2.5 MB (2,540 to 2,584 KB)
Working set after 10,000 sequential requests	6.1 MB (6,224 to 6,256 KB)	Five runs; private bytes 2.7 MB (2,748 to 2,788 KB); every request returned 200

The container image was not built for this edition, so its size is not reported.

> **Read these numbers narrowly.** They describe one trivial service on one machine, with the client on the same host. They are not a comparison with any other language, and they are not a capacity plan. Use the method, not the numbers: build your own service with your own pin and profile, measure it the same way under your own load, and put the result in the decision record as the baseline.

Not measured. Latency of any kind, the container image size, Linux resident memory inside a container and in a Kubernetes pod, CPU use, behavior under concurrent load, and startup time. These are the measurements a team should add for its own service before setting requests and limits.

APPENDIX B

The Three Tiers in Other Languages

The three tiers are not a Rust idea. Every language splits its guarantees between what the compiler or runtime proves, what tooling must enforce, and what operations and people still own. The line just sits in a different place. This appendix draws that line for TypeScript, Go, C# on .NET and Java, so a team that runs more than one language can write one set of contracts and fill in the enforcement point per language.

Read the tables as a map, not a ranking. A garbage collector moves memory safety into tier 1 for all four languages, and that is a real strength: the collector proves something that Rust proves with ownership. What differs is the rest: whether null is in the type system, whether data races are prevented or only detected, and how much of the supply chain the default tooling checks. Each language also has an escape hatch, as Rust has `unsafe`, and the same exception contract applies to it.

Tier 1: what the compiler or runtime proves

Language	Proved by default	Not proved (and the escape hatch)
Rust (for reference)	Memory safety and data-race freedom in safe code; types; exhaustive `match`; null is absent (`Option`)	Deadlocks, logic races, leaks, panics, overflow in release builds. Escape hatch: `unsafe`, FFI, build scripts
TypeScript	With `strict` on, type checking at compile time, including null and undefined checks (`strictNullChecks`). The type system is deliberately not sound: the TypeScript handbook names places where it accepts code it cannot prove safe, so strict checking is strong evidence, not a proof. The JavaScript runtime is memory safe through garbage collection	Types are erased: nothing is checked at run time, so data from the network is unproven until validated. Escape hatches: `any`, type assertions, non-null assertions (`!`), `@ts-ignore`
Go	Static types; memory safety through garbage collection and run-time bounds checks; unused imports and variables are compile errors	No null safety (a nil dereference panics at run time). Data races are not prevented, and the Go memory model warns that races on multiword values can corrupt memory. Escape hatches: `unsafe`, cgo
C# / .NET	Type safety and memory safety in managed code through the CLR and garbage collection; bounds checks. Nullable reference types add null-state analysis	Nullable analysis produces warnings, not errors, unless the project treats them as errors; data races are not prevented. Escape hatches: `unsafe` blocks (require `AllowUnsafeBlocks`), P/Invoke
Java	`javac` type checking, the JVM bytecode verifier, memory safety through garbage collection, bounds checks	Null is not in the type system (`NullPointerException` at run time); data races are not prevented, although the Java Memory Model defines their behavior. Escape hatches: JNI, the Foreign Function and Memory API (final since Java 22, with restricted methods), `sun.misc.Unsafe`

Tier 2: what tooling must enforce

Language	Code gate	Supply-chain gate	The pin
Rust	rustfmt, Clippy with `-D warnings`, `[lints]` table, tests, Miri	cargo-deny (advisories, licenses, bans, sources), cargo-audit, cargo-vet, cargo-semver-checks	`Cargo.lock`, `rust-toolchain.toml`, `rust-version`
TypeScript	`tsc --noEmit` with `strict` (plus options such as `noUncheckedIndexedAccess`); ESLint with typescript-eslint; run-time schema validation at every door	`npm audit`; review of install scripts; registry provenance where the registry offers it	`package-lock.json` committed and installed with `npm ci`; the Node.js version pinned (for example in `engines` and the CI image)
Go	`gofmt`, `go vet`, the race detector (`go test -race`, which finds races that tests exercise, not all races), staticcheck	`govulncheck`, which reports only vulnerabilities in code the module can reach; the checksum database (sum.golang.org)	`go.mod` and `go.sum`; the `go` and `toolchain` lines in `go.mod`
C# / .NET	`<Nullable>enable</Nullable>` with `TreatWarningsAsErrors`; .NET code analyzers (`AnalysisLevel`, `EnforceCodeStyleInBuild`)	NuGet audit during restore (on by default since the .NET 8 SDK; it reports warnings NU1901 to NU1904, so the gate must make them errors; transitive packages are audited by default for projects targeting .NET 10); package source mapping	`packages.lock.json` with locked-mode restore; `global.json` pins the SDK; Central Package Management
Java	Error Prone (compile-time checks as a `javac` plugin); SpotBugs (bytecode analysis); a nullness checker such as NullAway	Gradle dependency verification (`verification-metadata.xml`, checksums and signatures); OWASP Dependency-Check; Maven Enforcer rules	Gradle dependency locking; in Maven, explicit versions and a dependency management section (Maven has no lock file by default); a pinned JDK through toolchains

Tier 3: what operations and people own

Language	Runtime settings that become contracts	Owned by people in every language
Rust	Allocator choice, async runtime and worker threads, `panic` strategy, container memory limit	Business logic, deadlocks, timeouts and retries, capacity, the decision record and its renewal, the waiver register
TypeScript	Node.js heap size (`--max-old-space-size`), event-loop latency, the Node.js release line and its support window	
Go	`GOMEMLIMIT` and `GOGC`; `GOMAXPROCS` (container-aware by default since Go 1.25)	
C# / .NET	GC mode (server or workstation), heap hard limit, NativeAOT and trimming warnings treated as release blockers	
Java	Heap sizing (`-Xmx` or `-XX:MaxRAMPercentage`), collector choice (G1 by default, ZGC for low pause), GraalVM native image where startup and footprint matter	

USING THIS APPENDIX

- Write the contract once, in language-neutral words; fill in the enforcement point per language.
- Where a guarantee is tier 1 in one language and tier 2 in another (null, data races), the tier 2 gate must block, not warn.
- Every escape hatch (`unsafe`, any, cgo, JNI, P/Invoke) goes through the same exception lifecycle.
- Recheck this table when a language ships a major release; the line moves.

APPENDIX C

Glossary and Source Register

A contract only works when everyone reads its words the same way. This appendix fixes the meaning of the terms the book uses, then lists the public sources behind its claims, with publisher, year and address, so a reader can check every one.

Glossary of contract terms

Term	Meaning in this book
Contract	A written rule at a boundary with an owner, an enforcement point, a failure behavior and evidence. A rule without all four is a wish.
Door	Any place where something crosses into the system: code into the repository, a dependency into the build, an artifact into a registry, a workload into a cluster.
Three tiers	The split of every guarantee into what the compiler proves (tier 1), what tooling must enforce (tier 2) and what operations and people own (tier 3). The spine of every chapter.
The developer proposes, the compiler decides	The GDCD principle "the model proposes, the contract decides" applied to tier 1 only. It holds inside the trusted boundary and nowhere else.
Trusted boundary	The set of code whose correctness the safe-Rust guarantees rely on but do not check: the compiler, every `unsafe` block and the abstraction around it, dependencies, FFI, build scripts, procedural macros, the global allocator and the async runtime.
Contract card	The one-page table that ends Chapters 4 to 11: contract, threat addressed, owner, enforcement point, failure behavior, waiver authority and expiry, evidence the control ran, evidence the outcome improved, review cadence.
Enforcement point	The exact place a contract is checked: a compiler flag, a CI job, an admission controller, a probe. If you cannot name it, the contract is not enforced.
Failure behavior	What happens when the check fails: block, warn, page, roll back. Stated in advance, not decided during the incident.
Evidence the control ran	Proof that the check executed (a job log, an attestation). Different from, and weaker than, evidence the outcome improved.
Evidence the outcome improved	Proof that the business result the contract exists for actually moved (fewer advisories in production, fewer incidents of a class).

Term	Meaning in this book
The pin	Everything that fixes what a build is made of: the toolchain in `rust-toolchain.toml`, dependency versions in `Cargo.lock`, the minimum supported Rust version in `rust-version`, base images by digest.
Admission gate	The tier 2 check that decides what may enter the build or the cluster: advisories, licenses, banned crates, allowed sources, audits, signatures. It has a failure behavior and a waiver path.
Interface promise	The public API of a crate or service treated as a contract, versioned by semantic versioning and checked by an API diff in review.
Exception	Any deliberate step outside a default contract: an `unsafe` block, an FFI call, a build script, a lint waiver, an ignored advisory, an emergency toolchain change.
Exception lifecycle	The single path every exception follows: request, named owner, waiver authority, stated invariant and evidence, expiry date, decay review. An exception with no expiry is a new default nobody approved.
Unsafe register	The list of every `unsafe` site the estate owns, with owner, invariant, review and test evidence (for example Miri).
Waiver authority	The role allowed to approve an exception. Never the person who asks for it.
Decay review	The scheduled check that an exception, a contract or a decision still holds. Expired items are renewed with fresh evidence or removed.
Decision record	An architecture decision record (ADR) for the choice of language or tool: context, criteria, alternatives considered fairly, decision, consequences.
Renewal contract	The part of the decision record that keeps it alive: expected outcomes, baseline, review date, reversal triggers and exit path.
Reversal trigger	A measurable condition, agreed in advance, that reopens the decision.
Minimum viable governance	The starter set of contracts for a team without a platform team: the pin, a CI gate (fmt, Clippy, tests, cargo-deny), an unsafe policy and one decision record.
Reference service	The small HTTP service built for this book (Appendix A). The only source of footprint numbers in the book, measured with a stated method.
Estate	All the code, pipelines and running workloads an organization owns, governed as one system.

Source register

Each address was checked to load on 15 November 2026. Memory-safety statistics from these sources describe C and C++ codebases and are cited only in that scope.

Source	Publisher	Year	Scope and address
A proactive approach to more secure code	Microsoft Security Response Center	2019	About 70% of the CVEs Microsoft assigns each year are memory safety issues (C and C++ code). https://www.microsoft.com/en-us/msrc/blog/2019/07/a-proactive-approach-to-more-secure-code
Memory safety	The Chromium Projects	2020, updated	About 70% of high-severity Chromium security bugs are memory safety problems (C++ code). https://www.chromium.org/Home/chromium-security/memory-safety/
Eliminating Memory Safety Vulnerabilities at the Source	Google Security Blog	2024	Android: memory-safety share of vulnerabilities fell as new code moved to memory-safe languages. https://security.googleblog.com/2024/09/eliminating-memory-safety-vulnerabilities-Android.html
Software Memory Safety (Cybersecurity Information Sheet)	National Security Agency	2022, revised 2023	Guidance to prefer memory-safe languages. https://media.defense.gov/2022/Nov/10/2003112742/-1/-1/0/CSI_SOFTWARE_MEMORY_SAFETY.PDF
The Case for Memory Safe Roadmaps	CISA, NSA and international partners	2023	Asks manufacturers to publish memory-safe roadmaps. https://www.cisa.gov/resources-tools/resources/case-memory-safe-roadmaps
Memory Safe Languages: Reducing Vulnerabilities in Modern Software Development	CISA and NSA	2025	Joint information sheet on adopting memory-safe languages. 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	Policy case for memory-safe languages. https://bidenwhitehouse.archives.gov/wp-content/uploads/2024/02/Final-ONCD-Technical-Report.pdf

Source	Publisher	Year	Scope and address
The Rust Programming Language	The Rust Project	Current edition	Ownership, borrowing, error handling, concurrency. https://doc.rust-lang.org/book/
The Rust Reference	The Rust Project	Current	Language definition, including behavior considered undefined. https://doc.rust-lang.org/reference/
The Rustonomicon	The Rust Project	Current	Writing and reviewing `unsafe` code. https://doc.rust-lang.org/nomicon/
The Rust Edition Guide: Rust 2024	The Rust Project	2025	The 2024 edition, stable since Rust 1.85. https://doc.rust-lang.org/edition-guide/rust-2024/index.html
The Cargo Book	The Rust Project	Current	Manifests, lock files, workspaces, profiles, `[lints]`. https://doc.rust-lang.org/cargo/
The rustup book	The Rust Project	Current	Toolchains, `rust-toolchain.toml` overrides. https://rust-lang.github.io/rustup/
RustSec Advisory Database	Rust Secure Code Working Group	Current	Security advisories for crates; cargo-audit. https://rustsec.org/
RUSTSEC-2025-0052: async-std has been discontinued	RustSec Advisory Database	2025	The advisory cited in Chapter Seven. https://rustsec.org/advisories/RUSTSEC-2025-0052.html
cargo-deny	Embark Studios	Current	Advisories, licenses, bans and sources checks, configuration keys and diagnostic codes. https://embarkstudios.github.io/cargo-deny/
cargo-vet	Mozilla	Current	Recording and sharing dependency audits. https://mozilla.github.io/cargo-vet/
cargo-semver-checks	Project maintainers (open source)	Current	API diff against semantic versioning rules. https://github.com/obi1kenobi/cargo-semver-checks
cargo-chef	Project maintainers (open source)	Current	Docker layer caching for Rust dependencies. https://github.com/LukeMathWalker/cargo-chef

Source	Publisher	Year	Scope and address
Distroless container images	Google (GoogleContainerTools)	Current	Minimal base images without a shell or package manager. https://github.com/GoogleContainerTools/distroless
Resource Management for Pods and Containers	The Kubernetes Authors	Current	Requests, limits and how the scheduler uses them. https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/
Configure Liveness, Readiness and Startup Probes	The Kubernetes Authors	Current	Probe semantics. https://kubernetes.io/docs/tasks/configure-pod-container/configure-liveness-readiness-startup-probes/
Node-pressure Eviction	The Kubernetes Authors	Current	How the kubelet ranks Pods for eviction: usage over requests, priority, usage relative to requests. https://kubernetes.io/docs/concepts/scheduling-eviction/node-pressure-eviction/
Container Lifecycle Hooks; feature gate PodLifecycleSleepAction	The Kubernetes Authors	Current	The `sleep` hook action: beta and on by default in 1.30, stable in 1.34. https://kubernetes.io/docs/concepts/containers/container-lifecycle-hooks/ and https://kubernetes.io/docs/reference/command-line-tools-reference/feature-gates/
Pod Lifecycle	The Kubernetes Authors	Current	Termination, SIGTERM and the grace period. https://kubernetes.io/docs/concepts/workloads/pods/pod-lifecycle/
OpenTelemetry documentation	OpenTelemetry (CNCF)	Current	Traces, metrics, logs and their exporters. https://opentelemetry.io/docs/
OpenTelemetry Rust: status	OpenTelemetry (CNCF)	2026	Signal status for Rust (Chapter Eleven): traces beta, metrics and logs stable when checked in November 2026. https://opentelemetry.io/docs/languages/rust/
Documenting Architecture Decisions	Michael Nygard, Cognitect	2011	The original ADR format. https://cognitect.com/blog/2011/11/15/documenting-architecture-decisions
The Go Memory Model	The Go Authors	Current	Data races and their effects (Appendix B). https://go.dev/ref/mem

Source	Publisher	Year	Scope and address
Go 1.25 Release Notes	The Go Authors	2025	Container-aware `GOMAXPROCS` (Appendix B). https://go.dev/doc/go1.25
govulncheck and Go vulnerability management	The Go Authors	Current	Reachability-based vulnerability checks. https://go.dev/doc/security/vuln/
TSConfig reference: strict	Microsoft (TypeScript)	Current	Strict type-checking options. https://www.typescriptlang.org/tsconfig/#strict
TypeScript Handbook: Type Compatibility	Microsoft (TypeScript)	Current	States that the type system is not sound by design, and where. https://www.typescriptlang.org/docs/handbook/type-compatibility.html
npm audit; package-lock.json	npm, Inc. (GitHub)	Current	Advisory checks and lock files. https://docs.npmjs.com/cli/commands/npm-audit and https://docs.npmjs.com/cli/configuring-npm/package-lock-json
Nullable reference types	Microsoft Learn	Current	Null-state analysis in C#. https://learn.microsoft.com/en-us/dotnet/csharp/nullable-references
Auditing package dependencies for security vulnerabilities	Microsoft Learn (NuGet)	2025	NuGet audit defaults and warning codes. https://learn.microsoft.com/en-us/nuget/concepts/auditing-packages
Error Prone; SpotBugs	Google; SpotBugs project	Current	Java static analysis. https://errorprone.info/ and https://spotbugs.github.io/
Verifying dependencies	Gradle, Inc.	Current	Checksum and signature verification. https://docs.gradle.org/current/userguide/dependency_verification.html
The Z Garbage Collector	Oracle (Java SE 21 documentation)	2023	Low-pause collector (Appendix B). https://docs.oracle.com/en/java/javase/21/gctuning/z-garbage-collector.html

THE SERIES

GDCD in Code

GDCD in Code applies Governance-Driven Contract Design to the code door: the language, the toolchain and the supply chain that every system passes through.

Book 1 **Governing Rust by Contract**: owning the bits, from the compiler to the cluster. (This book.)

The foundation of the method is *The Contract Decides: Governing AI at Every Door*, Book 1 of the GDCD Field Guides. The same method on a cloud platform is *Governing Azure by Contract*. All books are free at blog.jcardena.com/en/ebooks.

Juan Cardena

Enterprise AI & Agentic Systems Architect · blog.jcardena.com

Download the latest edition at blog.jcardena.com/en/ebooks/governing-rust-by-contract.

GDCD in Code · Book 1 · Trust, earned at every layer.